UNIT-3

CONTEXT-FREE GRAMMARS AND PUSHDOWN AUTOMATA

Unit - III

CONTEXT FREE GRAMMARS

Context free grammar:

A grammar 'G' is said to be CFG if every production of the form '$\alpha \rightarrow \beta$' satisfies the following rules.

Rule 1: LHS should contain only one non-terminal.

$\therefore \alpha \in N^*$ and $|\alpha| = 1$

Rule 2: RHS can be anything, Non-terminal or terminals.

$\therefore \beta \in (N+T)^*$

→ It is denoted by 4 tuple Notation.

$G = \{V, T, P, S\}$

where, V = set of variables or non-terminals

T = set of terminals

S = starting symbol of the grammar.

P = set of finite no. of productions or rules.

Example: Consider $E \rightarrow E+T \mid T$

$T \rightarrow F$

$F \rightarrow id$

In the above CFG

V or non-terminals = $\{E, T, F\}$

Terminals $T = \{+, id\}$

$S = \{E\}$

$P = \{E \rightarrow E+T,\ E \rightarrow T,\ T \rightarrow F,\ F \rightarrow id\}$

Derivations using a Grammar:

It is a step by step process of replacing the non-terminals with the appropriate productions.

→ There are two types of derivations.

(a) left most derivation (LMD)

(b) Right most derivation (RMD)

left most derivation (LMD):

In each and every step of the derivation, left most non-terminal is replaced with the production.

Ex: consider the following productions

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Derive id+id*id using LMD.

Solution:

$E \Rightarrow E+T$ ($\because E \rightarrow E+T$)

$\Rightarrow T+T$ ($\because E \rightarrow T$)

$\rightarrow F+T$ ($\because T \rightarrow F$)

$\Rightarrow id+T$ ($\because F \rightarrow id$)

$\Rightarrow id+T*F$ ($T \rightarrow T*F$)

$\Rightarrow id+F*F$ ($\because T \rightarrow F$)

$\Rightarrow id+id*F$ ($\because F \rightarrow id$)

$\Rightarrow id+id*id$ ($\because F \rightarrow id$)

$E \overset{*}{\Rightarrow} id+id*id$.

Right most derivation:

In each and every step of the derivation right most non-terminal is replaced with the production.

Example:

Consider the following productions.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Derived id + id * id using RMD

Solution: $E \Rightarrow E+T$ ($\because E \rightarrow E+T$)

$\Rightarrow E+T*F$ ($\because T \rightarrow T*F$)

$\Rightarrow E+T*id$ ($\because F \rightarrow id$)

$\Rightarrow E+F*id$ ($\because T \rightarrow F$)

$\Rightarrow E+id*id$ ($F \rightarrow id$)

$\Rightarrow T+id*id$ ($\because E \rightarrow T$)

$\Rightarrow F+id*id$ ($\because T \rightarrow F$)

$\Rightarrow id+id*id$ ($\because F \rightarrow id$)

$E \overset{*}{\Rightarrow} id+id*id$.

The language of grammar:

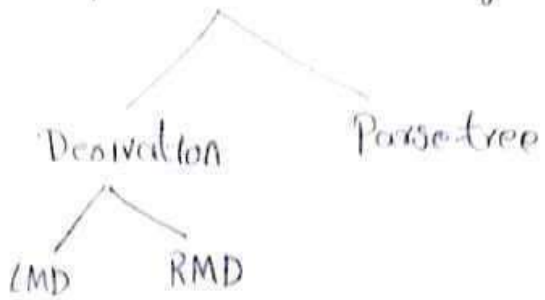
Every grammar generates a language such as regular grammar generates regular language, context free grammar generates context free language.

→ The language generated by a grammar is nothing but the strings accepted by the grammar.

→ A string can be accepted or a string is a member of a grammar can be analyzed by 2 ways

(a) Derivation (b) Parse-tree.

Acceptance of a string



Parse tree

Also known as syntax tree or derivation tree.

→ It is an hierarchial representation of a derivation

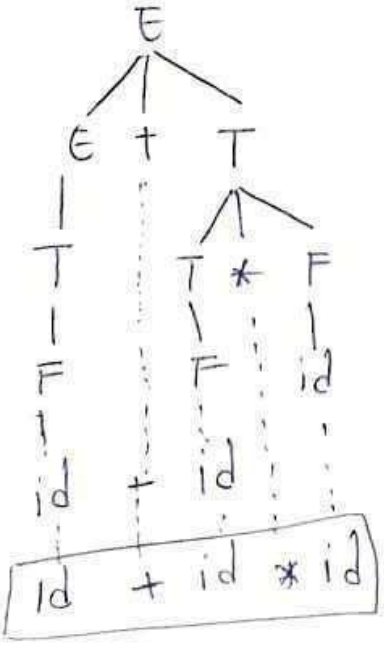
→ It consists of

- a root : starting : Non-terminal
- a leaf : Terminals
- Internals : Non-terminals.

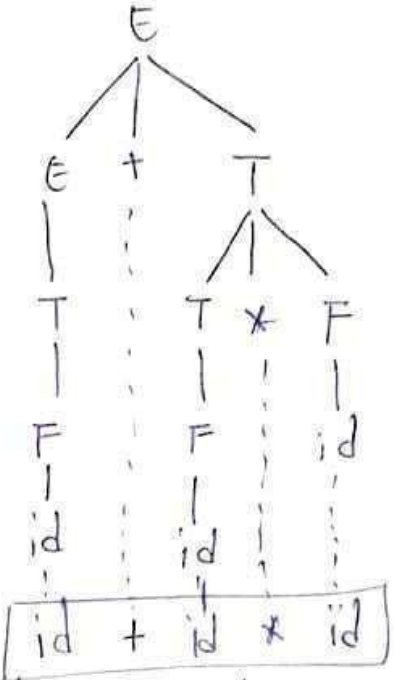
Ex: Consider a string 'w' = id + id * id from the above example in LMD / RMD

Deriving 'w' using parse tree :

LMD



RMD



We can observe that the parse tree of both LMD & RMD are same for w = id + id * id

Applications of CFG:

1. It is the most widely used grammars.
2. CFG's are used in text processing and compilers.
3. The parsers makes use of CFG to check the syntax of the source program, written in any language. Every statement of the source program is converted to its equivalent CFG and then to syntax tree by the parser.
4. CFG is also used in markup languages.
5. syntax analysis phase uses CFG to specify the syntactic rules to any programming languages.
6. semantic anlysis phase uses CFG to specify the type checking rules.

→ context free grammars were originally conceived by N. chomsky to describe natural languages.

→ CFG's are used to describe programming languages, and development of XML (Extensible Markup language), an essential part of XML is the Document type definition (DTD).

Passes:

1. YACC - Parser generator
2. XML and Document-Type definitions.

Parsers: Many aspects of a programming language have a structure that may be described by regular expressions. Ex: Typical languages use parentheses and/or brackets in nested and balanced fashion. We must be able to

match some left parenthesis against right parenthesis that appears immediately to its right, remove both of them, and repeat. If we eventually eliminate all the parentheses, then string was balanced and if we cannot match parentheses, then it is unbalanced.

Examples of strings of balanced parenthesis are (()), ()(), (()()) and ϵ, while)(and (() are not.

A Grammar $G_{bal} = (\{B\}, \{(,)\}, P, B)$ generates all and only the strings of balanced parentheses, where P consists of the Productions.

$$B \longrightarrow BB \mid (B) \mid \epsilon$$

- First production $B \rightarrow BB$ says concatenation of two strings of balanced parenthesis is balanced.
- Second Production $B \rightarrow (B)$, says if we replace a pair of parenthesis around a balanced string, then the result is balanced.
- Third production $B \rightarrow \epsilon$ is the basis it says that the empty string is balanced.

YACC - Parser Generator.

The generation of a parser (function that creates parse trees from source programs) has been institutionalized in the YACC command that appears in all unix systems

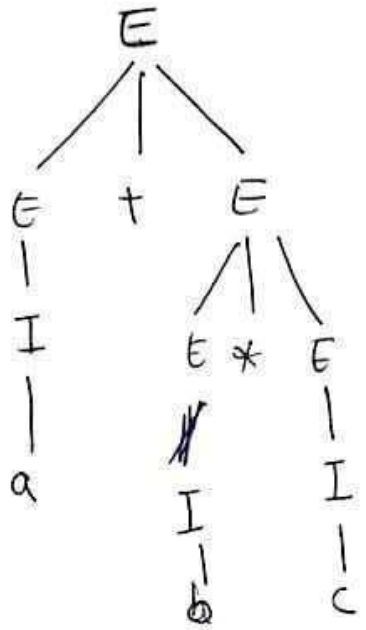
Ambiguity in Grammars & languages.

A CFG is said to be ambiguous if there exists some $w \in L(G)$ which has two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more left most or right most derivations.

Ex consider the grammar $G_1 : (V, T, P, S)$ with $V = \{E, I\}$ $T = \{a, b, c, +, *, (,)\}$ and productions:

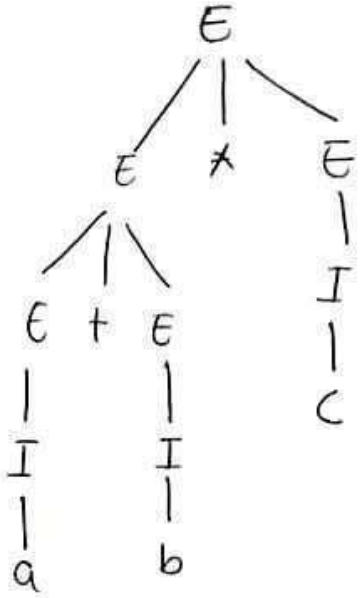
$E \rightarrow I$
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$I \rightarrow a \mid b \mid c$

consider two derivation trees for $a+b*c$



If a=5 b=6 c=7

value of tree will be 47



If a=5 b=6 c=7

value of tree will be 77

Inherently ambiguous

A context free language L is said to be inherently ambiguous if all its grammars are ambiguous.

Ex Consider the grammar for string aabbccdd

$S \rightarrow AB \mid C$

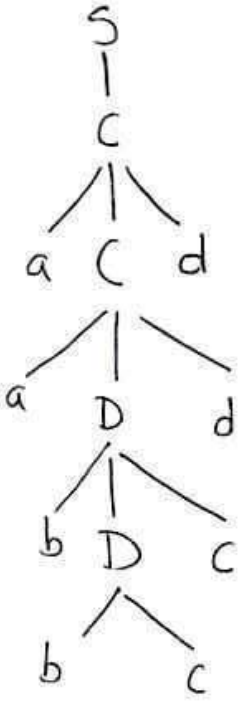
$A \rightarrow aAb \mid ab$

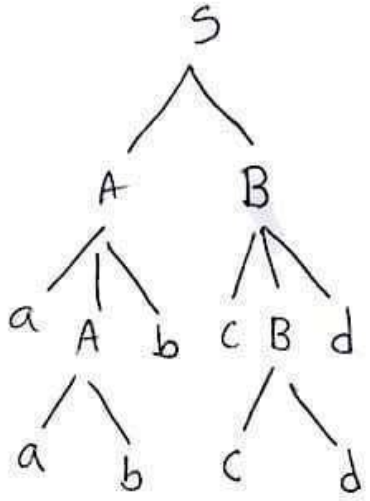
$B \rightarrow cBd \mid cd$

$C \rightarrow aCd \mid aDd$

$D \rightarrow bDc \mid bc$

Parse tree for string aabbccdd





PUSH DOWN AUTOMATA

A Finite automata with a stack memory can be viewed as push down automata. (PDA)

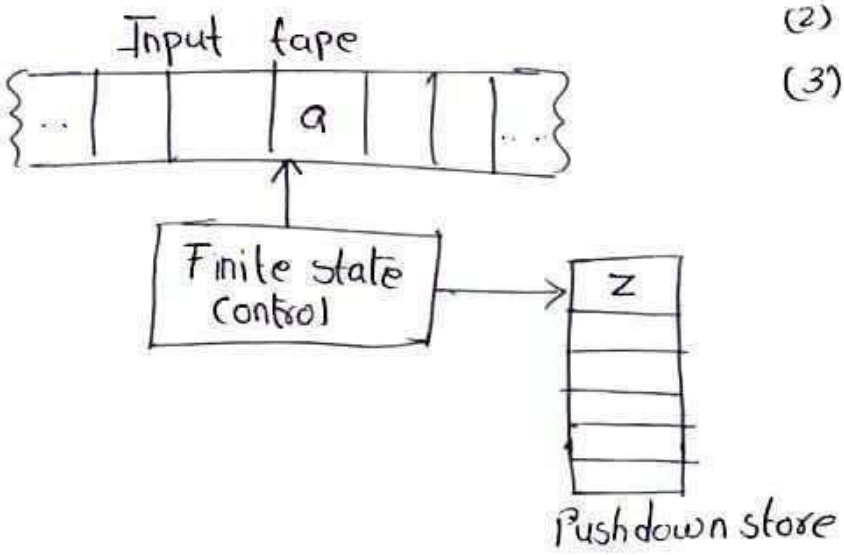
→ It can be defined as the automata or machine that processes all the context free languages.

→ FA cannot recognize every context free language and some context free languages are not regular.

→ some extra features are added to F.A inorder to accept any type of CFL.

→ The PDA performs transitions by observing symbols at top of the stack, present state and on the input symbol.

PDA consists of three components
(1) Input tape
(2) control unit
(3) stack structure



→ Input tape consists of linear configuration of cells each of which contains a character from an alphabet. The tape can be moved one cell at a time to the left.

→ The stack is also a sequential structure that has a first element.

→ The control unit contains both tape heads

Formal definition of PDA

Formally PDA is defined as a seven tuple machine.

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$$

where $Q \rightarrow$ Finite set of states

$\Sigma \rightarrow$ set of input symbols

$\delta \rightarrow Q \times \Sigma \times \Gamma \rightarrow Q \times \Gamma^*$

$\Gamma \rightarrow$ Finite set of symbols to push on the stack called "stack alphabet"

$q_0 \rightarrow$ Initial state of control unit $\in Q$

$Z \rightarrow$ stack start symbol

$F \rightarrow$ set of final states $F \subseteq Q$.

The arguments of δ are current state of control unit, the current input symbol and current stack symbol on the top of the stack.

Instantaneous description of a PDA:

Instantantaneous description of a PDA is a triplet (q, w, u)

where q = current state of the automaton

w = unread part of the input string.

u = stack contents (written as a string, with leftmost symbol on top of the stack).

Let the symbol '$\vdash$' denote move of the PDA and suppose that $\delta(q_1, a, x) = \{(q_2, y), \ldots\}$ then following is possible.

$$(q_1, aw, xz) \vdash (q_2, w, yz)$$

This notation tells that in moving from state q_1 to state q_2, an 'a' is consumed from the input string 'aw' and the x at the top of the stack 'xz' is replaced with y, leaving yz on the stack.

Example:

Construct a push down automata for a language

$$L = \{a^n b^n \mid n \geq 0\}$$

Solution: Firstly note down the basic strings generated by the following language.

$L = \{\epsilon, ab, aabb, aaabbb, \ldots\}$

→ which means the input symbol 'a' is followed by the input symbol 'b' and no. of input symbols 'a' is equal to the no. of input symbols 'b'.

→ The main idea here is to push all the a's in the stack until the control reaches 'b' in the string, then pop 'a' for each 'b'.

→ Repeat the above step until the stack is empty and we reach the end point of the string 'ϵ'.

Note: If we reach the end point of the string and still find 'a' in the stack then it means that number of

'a's are more than b's.

∴ Not acceptable / Rejected.

→ If the stack is empty and there are still b's left in the string, then it means that b's are more than a's.

∴ Not acceptable / Rejected.

→ consider the string generated by the language

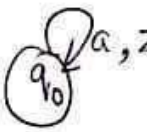
string | a | a | a | b | b | b | ∈ |

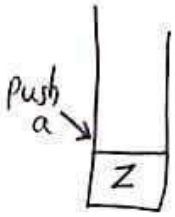
step1 : Read the first input symbol

a	a	a	b	b	b	∈
↑						

current state	i/p	Top of stack	next state	operation
q_0	a	Z	q_0	push

$(q_0, a, Z) \vdash \{q_0, aZ\}$

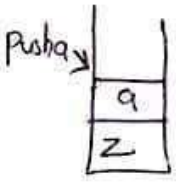


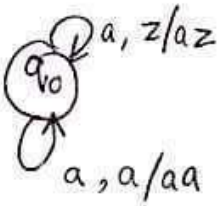


step2:

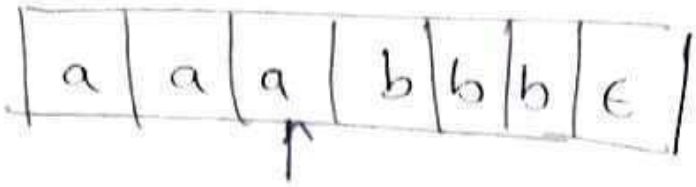
a	a	a	b	b	b	∈
	↑					

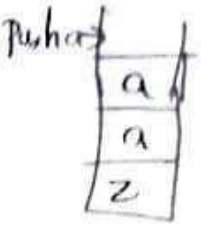
$(q_0, a, a) = \{q_0, aa\}$





step 3 ;

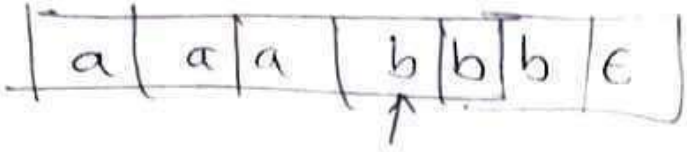


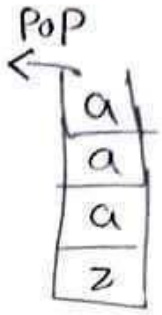


$(q_0, a, a) = \{q_0, aa\}$

repeat step 2

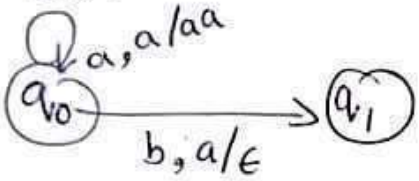
step 3 :



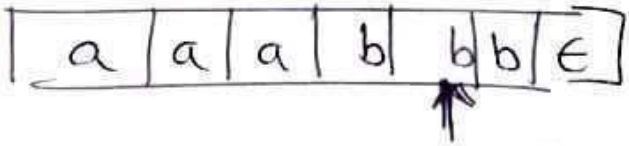


$(q_0, b, a) \vdash \{q_1, \epsilon\}$

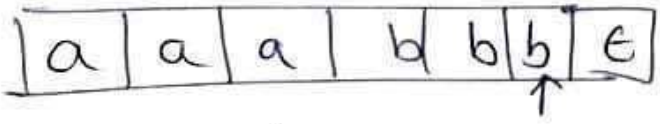
→ Pop operation
→ change the state

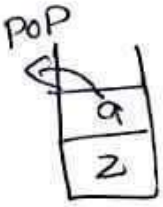


step 4:

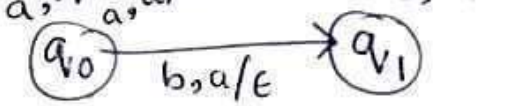


$(q_1, b, a) \vdash \{q_1, \epsilon\}$

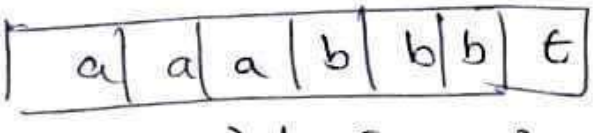




repeat the step 4



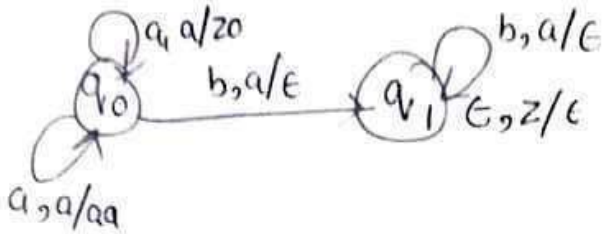
step 5 :

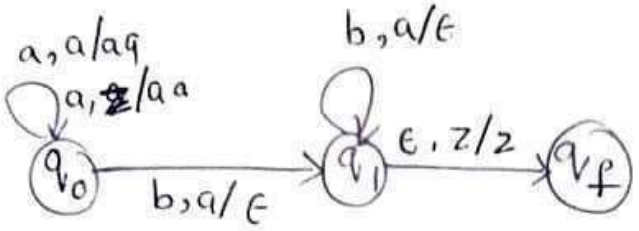


$(q_1, \epsilon, z) \vdash \{q_1, \epsilon\}$

∴ Stack is empty reached the end of the string.

∴ The final PDA (acceptance by empty stack)





Acceptance by final state

- Introduce a new state from q_1 to q_f

$$(q_1, \epsilon, Z_0) \vdash (q_f, z)$$

Instantaneous description is

$$\delta(q_0, a, z) \vdash (q_0, az)$$

$$\delta(q_0, a, a) \vdash (q_0, aa)$$

$$\delta(q_0, b, a) \vdash (q_1, \epsilon)$$

$$\delta(q_1, b, a) \vdash (q_1, \epsilon)$$

$$\delta(q_1, \epsilon, z_0) \vdash (q_1, \epsilon)$$

$$\delta(q_1, \epsilon, z_0) \vdash (q_f, z_0)$$

The simulation of the string is as follows

$$\delta(q_0, aaabbb, z_0) \Rightarrow \delta(q_0, aabbb, az_0)$$

$$\Rightarrow \delta(q_0, abbb, aaz_0)$$

$\Rightarrow \delta(q_0, bbb, aaaz_0)$

$\Rightarrow \delta(q_1, bbb, aaaz_0)$

$\Rightarrow \delta(q_1, bb, aaz_0)$

$\Rightarrow \delta(q_1, b, az_0)$

$\Rightarrow \delta(q_1, \epsilon, z_0)$

$\Rightarrow \delta(q_2, \epsilon)$ Accepted.

The Languages of PDA:

There are two ways to define the language of a PDA $P = (Q, \Sigma, T, \delta, q_0, z_0, F)$ $(L(P) \subseteq \Sigma^*)$ because there are two notions of acceptance.

(i) Acceptance by final state.

(ii) Acceptance by Empty stack.

Acceptance by final state:

A Push down automata is defined by $P=(Q, \Sigma, T, \delta, q_0, z_0, F)$ accepts a given language by generating the strings $w \in \Sigma^*$, by entering the final state such that

$(q_0, w, z_0) \Rightarrow (q_f, \epsilon, \alpha)$ where $q_f \subseteq F$ and

$\alpha \subset T^*$

ie, A PDA can accept strings through final state based on the following conditions.

(a) If string is completely traversed (i.e, it comes to an end)

(b) If the PDA has visited its final state

i.e. $\delta(q_i, \epsilon, z_0) = (q_f, z_0)$

Acceptance by Empty stack:

A PDA is defined by $P = (Q, \Sigma, T, \delta, q_0, z_0, F)$ accepts a given language by generating the set of strings $w \in \Sigma^*$ by making its stack empty such that

$$(q_0, w, z_0) \Rightarrow (q_0, \epsilon, \epsilon)$$

→ A PDA can accept strings by empty stack based on the following conditions:

(a) If the string is completely traversed (i.e, it comes to one end)

(b) If the PDA has its stack empty.

Thus if a string of PDA are accepted by empty stack then the final state should be of type.

$$\delta(q_i, \epsilon, z_0) \Rightarrow (q_f, \epsilon).$$

Example 2 PDA for

$L = \{a^n b^m c^n \mid n \geq 1, m \geq 1\}$

Sol: Strings generated by this language is

$L = \{abc, aabcc, abbbc, \ldots\}$

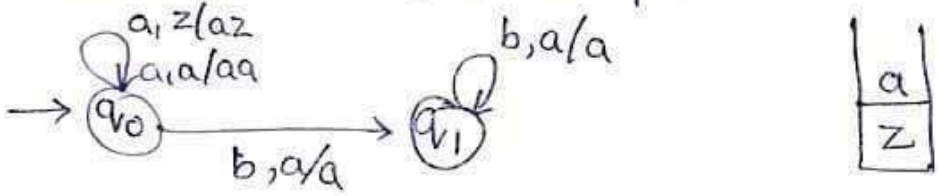
ie. equal no. of a's & equal number of c's and any number of b's between a's & c's.

→ The concept here is all a's are pushed on to the stack, when b will come 'no operation' and for each 'c' pop one 'a' then stack will be empty.

steps are as follows.

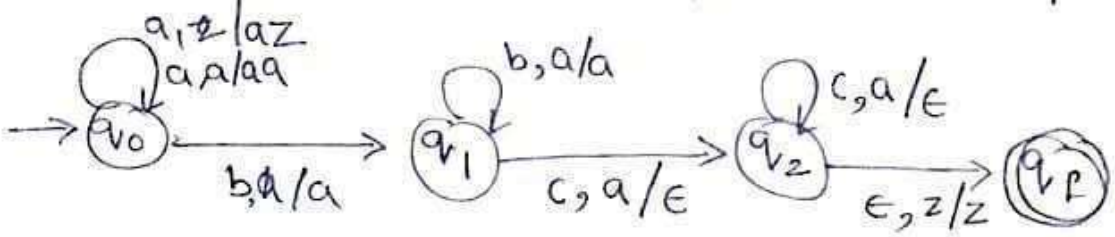
(i) First Push 'a' on to the stack.

(ii) when b is read as input.



Here any number of b's may be read, but no operation is performed.

(iii) when 'c' is read then pop 'a' from top of stack ie.,



For each 'c' pop 'a' then stack will be empty then reaches final state. This language is deterministic PDA (DPDA).

Consider any string 'aabbbcc' generated by given language

$(q_0, aabbbcc, z) \vdash (q_0, abbbcc, az)$ (: 'a' pushed on to stack)

$\vdash (q_0, bbbcc, aaz)$ (second 'a' is also pushed on to stack)

$\vdash (q_0, bbcc, aaz)$ (: when 'b' is read stack top remains same as no operation is performed)

$\vdash (q_1, bcc, aaz)$ (: No operation)

$\vdash (q_1, cc, aaz)$

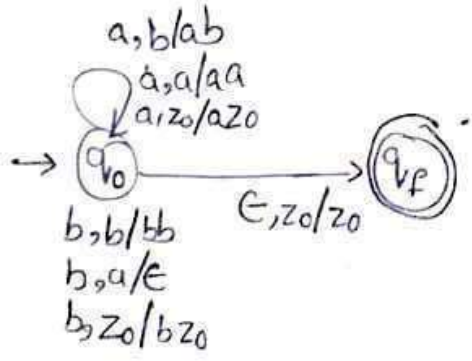
$\vdash (q_2, c, az)$ (: when 'c' is read change state and pop 'a')

$\vdash (q_2, \epsilon, z)$

There fore when ϵ is read then state reaches the acceptance / final state.

Example3: PDA for $L = \{w \in (a,b)^* \mid na = nb\}$

Soln. $L = \{\epsilon, ab, ba, abab, baba, \ldots\}$



Push down Automata.

(1) DPDA (Deterministic push down automata)

(2) NPDA (Non-deterministic " ")

DPDA :

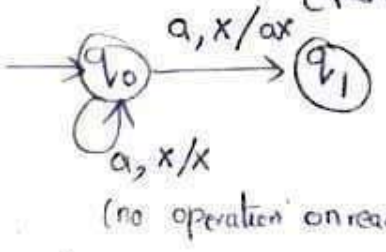
(1) Center symbol is known (ie. up to which element we need to perform push and at which element we need to pop Ex: $a^n b^n$, wcw^R.

(2). There is only one moves in every situation

$$\delta(q_0, a, Z_0) \Rightarrow (q_0, aZ_0)$$

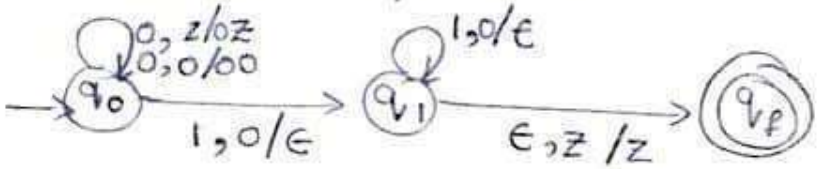
If M is deterministic then it should satisfy, for any $q \in Q$, $a \in \Sigma$ and $X \in T$ then (q, a, X) has atmost one transition, and $(q, \epsilon, X) \neq \emptyset$ then $(q, a, X) \neq \emptyset$ for every $a \in \Sigma$.

Ex:



Here two operations are performed for same input symbol for same state, this is not allowed in DPDA.

→ The grammar $L = \{0^n 1^n / n \geq 1\}$, ther string 0011 has the PDA



The ID's here are

$(q_0, 0, z) \vdash (q_0, 0z)$

$(q_0, 0, 0) \vdash (q_0, 00)$

$(q_0, 1, 0) \vdash (q_1, \epsilon)$

$(q_1, 1, 0) \vdash (q_1, \epsilon)$

$(q_1, \epsilon, z) \vdash (q_f, z)$

This is a deterministic PDA because in first two steps of ID input is same and stack top is different

NPDA

(1) Center is not known

i. ww^R suppose aaaa is string, we donot know for which we need to push or pop.

(2) There are multiple moves in every situation

$\delta(q_0, a, a) \vdash (q_0, aa)$ (or) (q_1, a) (or) (q_1, ϵ)

Examples of DPDA & NPDA are

① Construct PDA for $L = \{ww^R \mid w \in (a,b)^*\}$

Solution: Language $L = \{aa, bb, abba, abaaba, \ldots\}$

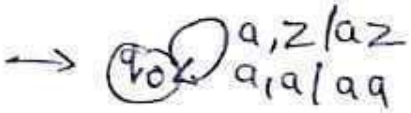
$\underset{w}{a}\ \underset{w^R}{a}$ $\underset{w}{ab}\ \underset{w^R}{ba}$ re

b	b
a	a
ab	ba
ba	ab
abb	bba
w	w^R

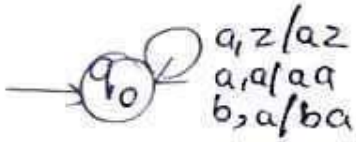
→ The concept here is for each symbol of 'w' we need to perform push operation and for w^R perform pop operation, then stack will be empty.

→ some times we are not clear where to stop push operation and start pop operation, so this cannot be a deterministic PDA.

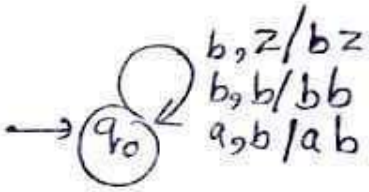
→ when ~~we~~ we read input a then



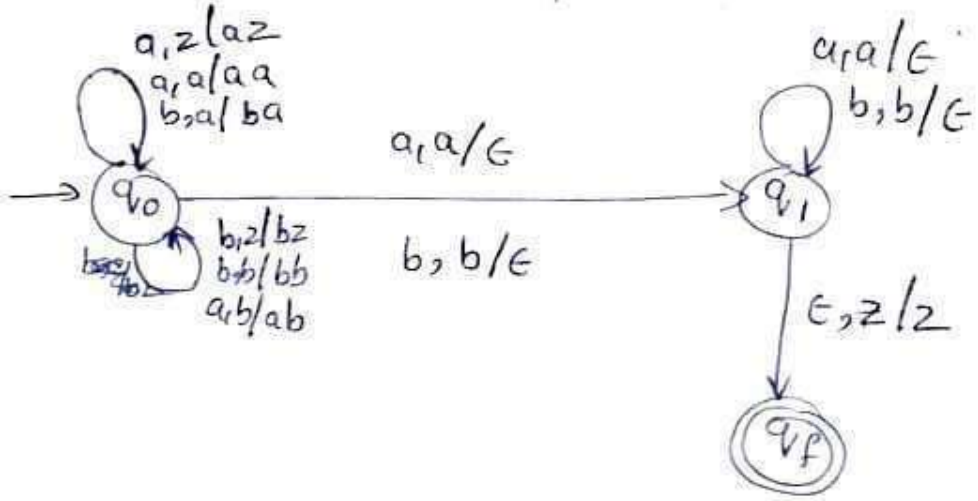
→ when we read input b then (ie. $w = aab$)



→ Suppos 'b' is read on empty stack (ie $w = bba$)



→ To go to q_1 state



This is a Non deterministic PDA because for

$(q_0, a, a) \Rightarrow (q_0, aa)$ – push operation

$(q_0, a, a) \Rightarrow (q_1, G)$ – pop operation

Here for same i/p & same stack top different operations are performed (that's why this is 'NPDA'.

→ This is a CFL because only NPDA can have CFL language, but not DPDA.

For string abba

$(q_0, w, z) \vdash (q_0, abba, z)$

(q_0, bba, az)

(q_0, ba, baz)

(q_1, a, az)

(q_1, ϵ, z)

Example

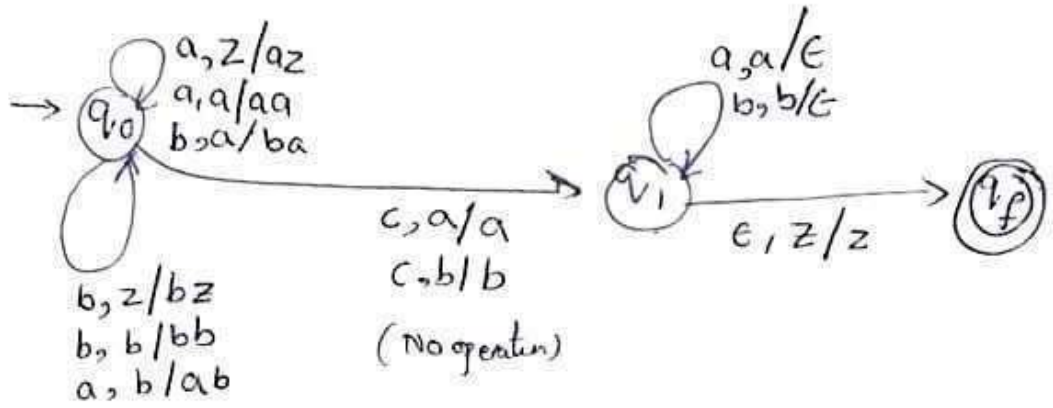
$L = \{ wcw^R \mid w \in (a,b)^* \text{ and } c \text{ is input alphabet} \}$

Solution consider strings aca

bcb

abcba

w c w^R



This is deterministic PDA.

Consider string abcba

$(q_0, abcba, Z) \vdash (q_0, bcba, aZ)$

$\vdash (q_0, cba, baZ)$

$\vdash (q_1, ba, baZ)$

$\vdash (q_1, a, aZ)$

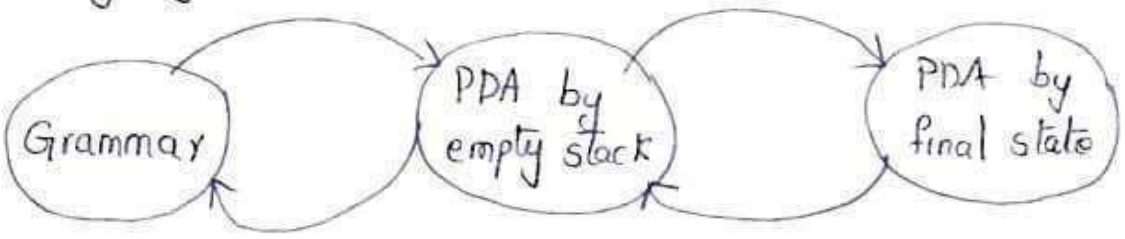
$\vdash (q_1, \epsilon, Z)$

$\vdash (q_f, \epsilon Z)$

→ This language is deterministic context free language.

Equivalence of CFG & PDA:

Languages defined by PDA's are exactly the context free language.



Equivalence of three ways of defining the CFL's

The goal is to prove that the following three classes of languages.

1. The context free languages ie, the languages defined by CFG's.
2. The languages that are accepted by final state by some PDA.
3. The languages that are accepted by empty stack by some PDA.

Conversion from grammars to Push down automata:

$G = \{V, T, P, S\}$ $\quad M = \{Q, \Sigma, T, \delta, q_0, Z_0, F\}$

$A \rightarrow \alpha$

$A \in V$ & $\alpha \in (VUT)^*$

Then Push down automata P that accepts $L(G)$ by empty stack as follows:

$P = (\{q\}, T, VUT, \delta, q, S)$

$\{q\}$ – no of states

T – i/p symbol is replaced by T.

T – is replaced by VUT

where transition function δ is defined by:

1. For each variable A,

$$\delta(q, \epsilon, A) = \{(q, \beta) \mid A \to \beta \text{ is a production of } P\}$$

(i.e., push operation)

2. For each terminal a,

$$\delta(q, a, a) = \{(q, \epsilon)\} \quad \text{(i.e., pop operation).}$$

Example.

① construct a PDA equivalent to the grammar

$$S \to aAA$$
$$A \to aS \mid bS \mid a.$$

Solution: First identify set of Variables & Terminals

$$T = \{a, b\}$$
$$V = \{S, A\}$$

For every variable

$$\delta(q, \epsilon, A) = \{(q, aAA)\}$$

$$\delta(q, \epsilon, S) = \{(q, aS), (q, bS), (q, a)\}$$

For each terminal

$$\delta(q, a, a) = \{(q, \epsilon)\}$$

$$\delta(q, b, b) = \{(q, \epsilon)\}$$

These four transition functions are used for PDA

Consider string aabb is acceptable or not

$$\delta(q, aabb, S) \vdash$$

Ex 2 Construct a PDA for the following CFG and test whether "abbabb" is in N(P).

$G = (\{S, A\}, \{a, b\}, R, S)$

$R = \{S \to AA \mid a,\ A \to SA \mid b\}$

Soln Identify set of Variables & Terminals

$V = \{S, A\}$ & $T = \{a, b\}$

For each Variable

$\delta(q, \epsilon, S) = \{(q, AA), (q, a)\}$

$\delta(q, \epsilon, A) = \{(q, SA), (q, b)\}$

For each terminal

$\delta(q, a, a) = \{(q, \epsilon)\}$

$\delta(q, b, b) = \{(q, \epsilon)\}$

$\delta(q, abbabb, S) \vdash (q, abbabb, AA)$

$\vdash (q, abbabb, SAA)$

$\vdash (q, \underline{a}bbabb, \underline{a}AA)$ $\vdash (q, \underline{a}bb, \underline{a}AA)$

$\vdash (q, bbabb, AA)$ $\vdash (q, bb, AA)$

$\vdash (q, \underline{b}babb, \underline{b}A)$ $\vdash (q, \underline{b}b, \underline{b}A)$

$\vdash (q, babb, A)$ $\vdash (q, b, A)$

$\vdash (q, babb, SA)$ $\vdash (q, \underline{b}, \underline{b})$

$\vdash (q, babb, AAA)$ $\vdash (q, \epsilon)$

$\vdash (q, \underline{b}abb, \underline{b}AA)$ Finally stack is empty, string is accepted.

$\vdash (q, abb, AA)$

$\vdash (q, abb, SAA)$

Conversion of PDA to CFG:

To convert the PDA to CFG, we use the following rules:

R1: The productions from startsymbol S are given by

$S \longrightarrow [q_0, Z_0, q]$ for some state q in Q.

R2: Each move that pops a symbol from stack, with transitions.

$$\delta(q, a, Z_i) = (q, \epsilon)$$

includes a production as

$[q\ Z_i\ q_1] \longrightarrow a$ for q_1 in Q

R3: Each move that doesnot pop any symbol from stack with transition as

(i) $\delta(q, a, Z) = (q_1, Z_1 Z_2 Z_3 \ldots)$

includes a production as

$[q_1\ Z_0\ q_m] \longrightarrow a\,[q_1 Z_1 q_2]\,[q_2 Z_2 q_3]\,[q_3 Z_3 q_4] \ldots [q_{m-1} Z_m q_m]$

for each q_i in Q wher $1 \le i \le m$

(ii) $\delta(q, a, Z) = (q_1, \epsilon)$
then $[q\ Z\ q_1] \longrightarrow a$

(iii) $\delta(q, \epsilon, Z) \longrightarrow (q_1, \epsilon)$
then $[q\ Z\ q_1] \longrightarrow \epsilon$.

After defining the rules apply simplification of grammar to get reduced grammar.

Example: convert the following PDA to CFG.

$\delta(q_0, a, Z_0) = (q_0, aZ_0)$
$\delta(q_0, a, a) = (q_0, aa)$
$\delta(q_0, b, a) = (q_1, a)$
$\delta(q_1, b, a) = (q_1, a)$
$\delta(q_1, a, a) = (q_1, \epsilon)$
$\delta(q_1, \epsilon, Z_0) = (q_1, \epsilon)$

Soln: To convert to CFG first note set of variable, $V = \{S\}$ and seven tuple notation of PDA ie.

$P = (Q, \Sigma, T, \delta, q_0, Z_0, F)$

$P = (\{q_0, q_1\}, \{a, b\}, \{a, Z_0\}, \delta, q_0, Z_0, F)$

$V = \{S, [q_0, a, q_0], [q_0 Z_0 q_0], [q_0 a q_1], [q_0 Z_0 q_1],$

$[q_1 a q_0], [q_1 Z_0 q_0] [q_1 a q_1] [q_1 Z q_1]\}$

These are variables of CFG

Now writing productions for states by considering RI.

(i) $S \longrightarrow [q_0 Z_0 q_0]$

$S \longrightarrow [q_0 Z_0 q_1]$

} $[q_0, Z_0]$ is common then followed by state $\{q_0, q_1\}$

(ii) $\delta(q_0, a, Z_0) = (q_0, aZ_0)$ (∵ As there are two symbol in stack $2^2 = 4$ productions are possible)

As 2^2 → power is 2 (in right side has two derivations).

$[q_0 Z_0 q_0] \longrightarrow a [q_0 q_0] [q_0 Z_0 q_0]$

$[q_0 Z_0 q_0] \longrightarrow a [q_0 a q_1] [q_1 Z_0 q_0]$

$[q_0 Z_0 q_1] \longrightarrow a [q_0 q q_0] [q_0 Z_0 q_1]$

$[q_0 Z_0 q_1] \longrightarrow a [q_0 a q_1] [q_1 Z_0 q_1]$

states are written alternatively

(iii) $\delta(q_0, a, a) = (q_0, aa)$

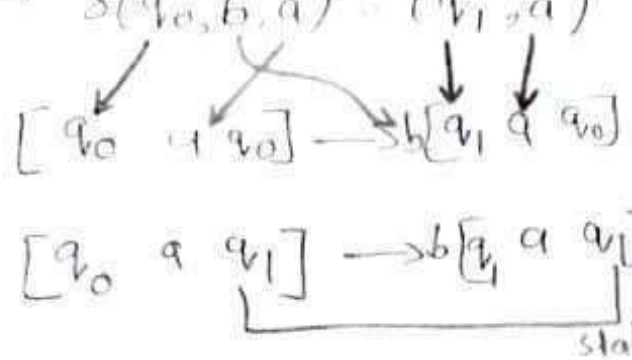
$[q_0 a q_0] \longrightarrow a [q_0 a q_0] [q_0 a q_0]$

$[q_0 a q_0] \longrightarrow a [q_0 a q_1] [q_1 a q_0]$

$[q_0 a q_1] \longrightarrow a [q_0 a q_0] [q_0 a q_1]$

$[q_0 a q_1] \longrightarrow a [q_0 a q_1] [q_1 a q_1]$

(iii) $\delta(q_0, b, a) = (q_1, a)$ (stack contains only one symbol)



two productions are possible
$2^1 - 2$ and power is 1 then right side is only one terminal

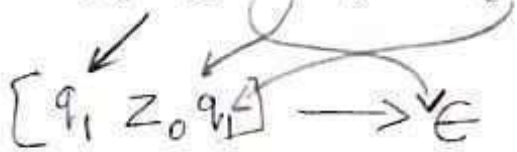
(iv) $\delta(q_1, b, a) = (q_1, a)$

$[q_1\ a\ q_0] \longrightarrow b\ [q_1\ a\ q_0]$

$[q_1\ a\ q_1] \longrightarrow b\ [q_1\ a\ q_1]$

(v) $\delta(q_1, a, a) = (q_1, \epsilon)$ [∵ stack is empty]

(vi) $\delta(q_1, \epsilon, z_0) = (q_1, \epsilon)$



These are the set of productions for CFG1

Ex 2 Convert the following PDA in to CFG.

$P = (\{q, p\}, \{0, 1\}, \{X, Z\}, \delta, q, Z)$

$\delta(q, 1, Z) = (q, XZ)$

$\delta(q, 1, X) = (q, XX)$

$\delta(q, \epsilon, X) = (q, \epsilon)$

$\delta(q, 0, X) = (p, X)$

$\delta(p, 1, X) = (p, \epsilon)$

$\delta(p, 0, Z) = (q, Z)$

FORMAL LANGUAGES AND AUTOMATA THEORY 10CS56

The Language of a Grammar

If G(V,T,P,S) is a CFG, the language of G, denoted by L(G), is the set of terminal strings that have derivations from the start symbol.

L(G) = {w in T | S → w}

Sentential Forms

Derivations from the start symbol produce strings that have a special role called "sentential forms". That is if G = (V, T, P, S) is a CFG, then any string In (V T)* such that S → is a sentential form. If S → , then is a left – sentential form, and if S → , then is a right – sentential form. Note that the language L(G) is those sentential forms that are in T*; that is they consist solely of terminals.

For example, E * (I + E) is a sentential form, since there is a derivation E → E * E → E * (E) → E * (E + E) → E * (I + E)
However this derivation is neither leftmost nor rightmost, since at the last step, the middle E is replaced.
As an example of a left - sentential form, consider a * E, with the leftmost derivation. E → E * E → I * E → a * E
Additionally, the derivation
E → E * E → E * (E) → E * (E +
E) Shows that
E * (E + E) is a right - sentential form.

: Applications of Context – Free Grammars

- Parsers
- The YACC Parser Generator
- Markup Languages
- XML and Document typr definitions

The YACC Parser Generator

E→ E+E | E*E |
(E)|id %{ #include
<stdio.h> %}
%token ID id
%%
Exp : id { $$ = $1 ; printf ("result is %d\n", $1);}
| Exp „+" Exp {$$ = $1 + $3;}
| Exp „*" Exp {$$ = $1 * $3; }
| „(„ Exp „)" {$$ = $2; }
;

```
%%

int main (void) {
return yyparse ( );
}
void yyerror (char *s) {
fprintf (stderr, "%s\n", s);
}
%{
#include "y.tab.h"
%}
%%
[0-9]+          {yylval.ID = atoi(yytext); return id;}
[ \t \n]          ;
[+ * ( )]               {return yytext[0];}
.                       {ECHO; yyerror ("unexpected character");}
%%
```

Example 2:

```
%{
 #include <stdio.h>
%}
%start line
%token <a_number> number
%type <a_number> exp term factor
%%
line : exp ';' {printf ("result is %d\n", $1);}
;
exp : term {$$ = $1;}
        | exp '+' term {$$ = $1 + $3;}
        | exp '-' term {$$ = $1 - $3;}
term : factor {$$ = $1;}
        | term '*' factor {$$ = $1 * $3;}
        | term '/' factor {$$ = $1 / $3;}
;
factor : number {$$ = $1;}
| '(' exp ')' {$$ = $2;}
;
 %%
 int main (void) {
return yyparse ( );
}
void yyerror (char *s) {
fprintf (stderr, "%s\n", s);
}
%{
#include "y.tab.h"
%}
%%
```

```
[0-9]+ {yylval.a_number = atoi(yytext); return number;}
[ \t\n] ;
[-+*/();] {return yytext[0];}
.              {ECHO; yyerror ("unexpected character");}
%%
```

Markup Languages

Functions

- Creating links between documents
- Describing the format of the document

Example

The Things I *hate*

1. Moldy bread
2. People who drive too slow In the fast lane

HTML Source

```
<P> The things I
<EM>hate</EM>: <OL>
<LI> Moldy bread
<LI>People who drive too
slow In the fast lane
</OL>
```

HTML Grammar

- Char — a | A | ...
- Text — e | Char Text
- Doc — e | Element Doc
- Element — Text | <EM> Doc </EM>| <p> Doc | <OL> List </OL>| ...

5. List-Item — <LI> Doc
6. List — e | List-Item List — Start symbol

XML and Document type definitions.

1. A→E1,E2.
   A→BC
   B→E1
   C→E2
2. A→E1 | E2.
   A→E1
   A→E2
3. A→(E1)*
   A→BA
   A→ ∈
   B→E1
4. A→(E1)+
   A→BA
   A→B
   B→E1
5. A→(E1)?
   A→ ∈
   A→E1

4.4:Ambiguity

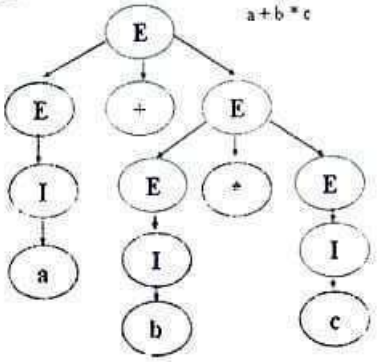
A context - free grammar G is said to be ambiguous if there exists some w L(G) which has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more left most or rightmost derivations.

Ex:-

Consider the grammar G=(V,T,E,P) with V={E,I}, T={a,b,c,+,*,(,)}, and productions. E→I,
E→E+E,
E→E*E,
E→(E),
I→a|b|c

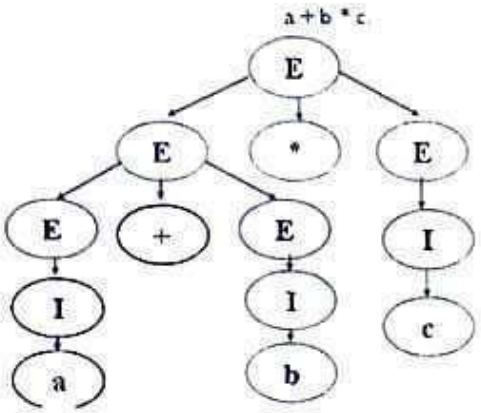
Consider two derivation trees for a + b * c.

Tree I
Let a=5, b=6, c=7
The value for Tree I will be 47



Tree II

Let a=5, b=6, c=7
The value for Tree II will be 77



Now unambiguous grammar for the above
Example:

E→T, T→F, F→I, E→E+T, T→T*F,
F→(E), I→a|b|c

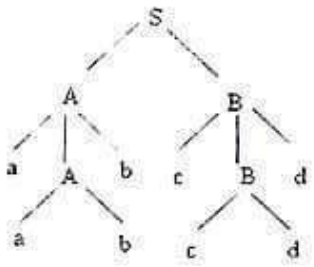
Inherent Ambiguity

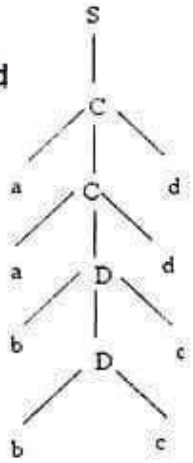
A CFL L is said to be inherently ambiguous if all its grammars are ambiguous Example:
Condider the Grammar for string

aabbccdd S→AB | C
A→ aAb | ab
B→cBd | cd
C→ aCd | aDd
D->bDc | bc

Parse tree for string aabbccdd

Parse tree for string aabbccdd





ASSIGNMENT QUESTIONS

1) The following grammar generates the language of RE
   $0^*1(0+1)^*$
   $S \rightarrow A|B$
   $A \rightarrow 0A|$
   $B \rightarrow 0B|1B|$
   Give leftmost and rightmost derivations of the following strings
   a) 00101 b) 1001 c) 00011

2) Consider the grammar
   $S \rightarrow aS|aSbS|$
   Show that deviation for the string aab is ambiguous

3) The following grammar generates the language of RE
   $0^*1(0+1)^*$
   $S \rightarrow A|B$
   $A \rightarrow 0A|$
   $B \rightarrow 0B|1B|$
   Give leftmost and rightmost derivations of the following strings
   a) 00101 b) 1001 c) 00011

4) Consider the grammar
   $S \rightarrow aS|aSbS|S$

   Show that deviation for the string aab is ambiguous