6) What is context Free Grammar and design of context Free Grammar (CFG)?

Context - Free Grammar.

A Context - Free Grammar (CFG) is a formal grammar used to describe the syntax of programming languag -es, mathematical expressions, and many formal language -s. Is called "context - free" because production rules can be applied regardless of the surrounding symbols (i.e. independent of context).

Defination of CFG.

A CFG is defined as a 4 - tuple

$$G = (V, \Sigma, P, S)$$

where

$V \rightarrow$ Finite set of variables (non-Terminals)

$\Sigma \rightarrow$ Finite set of terminals (input alphabet, actual symbols)

$P \rightarrow$ Finite set of production rules of the form $A \rightarrow a$

where $A \in V$ and $a \in (V \cup \Sigma)^*$.

$S \rightarrow$ Starting element (or) initial element (or) root ; $(S \in \Sigma)$

→ Design of CFG.

Designing a CFG means writing rules to generate all strings of a language.

Steps:

1. Identify the structure of the language (recursive, hierarical)
2. Define non-terminals for major syntatic categories (e.g. Expressions, Term, Factor)
3. Write production rules to describe valid constructions
4. Ensure no invalid strings are generated.

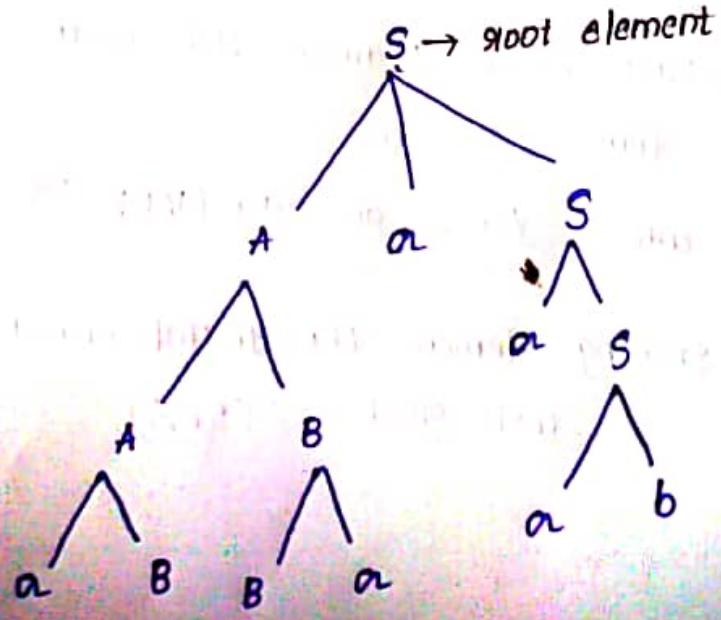
Example:- consider the expression like aboaabaaaab

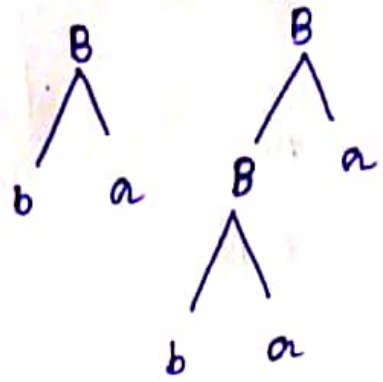
using production rule

$S \rightarrow Aas/as/ab$

$A \rightarrow AB/aB/Ab$

$B \rightarrow Ba/ba.$





∴ the Grammar from left to right is

abaabaaaab.

7) write about Parse Trees.

Parse Tree:

A parse tree also called a Derivation Tree is a tree representation of the derivation of a string according to a context-free grammar (CFG)

→ It shows how the start symbol expands using production rules until we get the string.

properties of a Parse Tree.

1. Root node → the start symbol of the grammar.
2. Internal nodes → Non-terminals (variables) of the grammar.
3. Leaf nodes → Terminals of ε (empty strings).
4. Children of a node → Correspond to the symbols on the right-hand side of a production rule.

(5) Reading leaves left to right → Gives the desired string (sentence) in the language.

Types of Derivations in parse tree:

→ Parsing is divided into 2 steps.

(1) Top Down Parsing

(2) Bottom up parsing

→ The Parser is used for syntax Analysis.

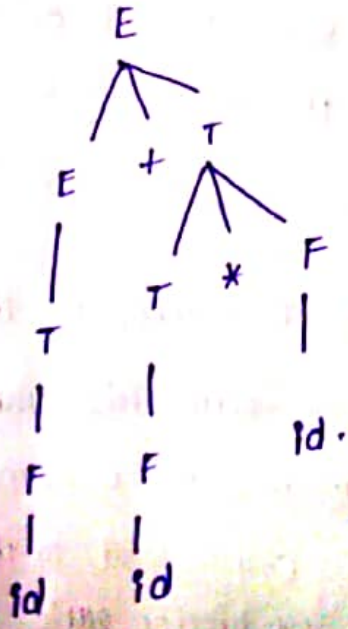
Example of Parse Tree.

Generating a string id + id * id using to the [string] production rule

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E) / id.$



∴ The generated string is id + id * id

Example of top down and Bottomup Parsing.

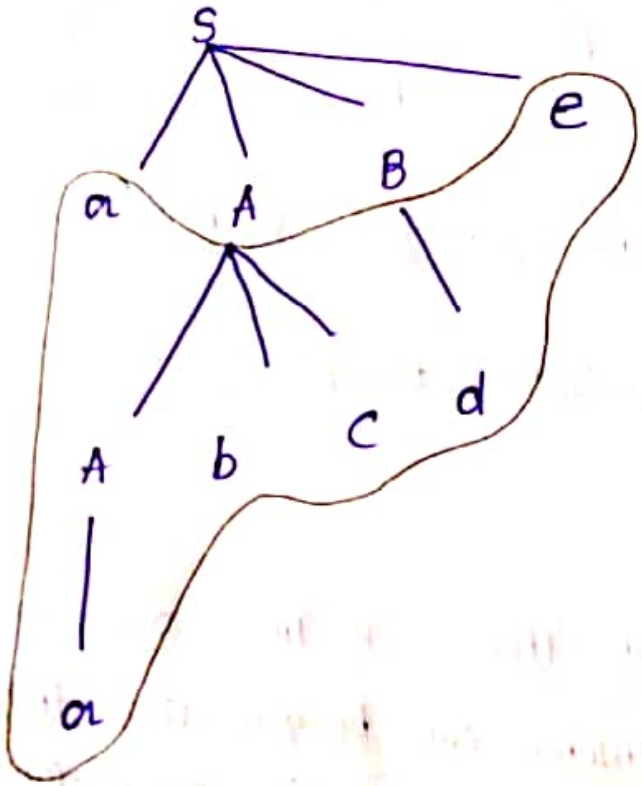
Generate a string aabcde according to the given production rule

$S \rightarrow a\,A\,B\,e$

$A \rightarrow Abc \mid a$

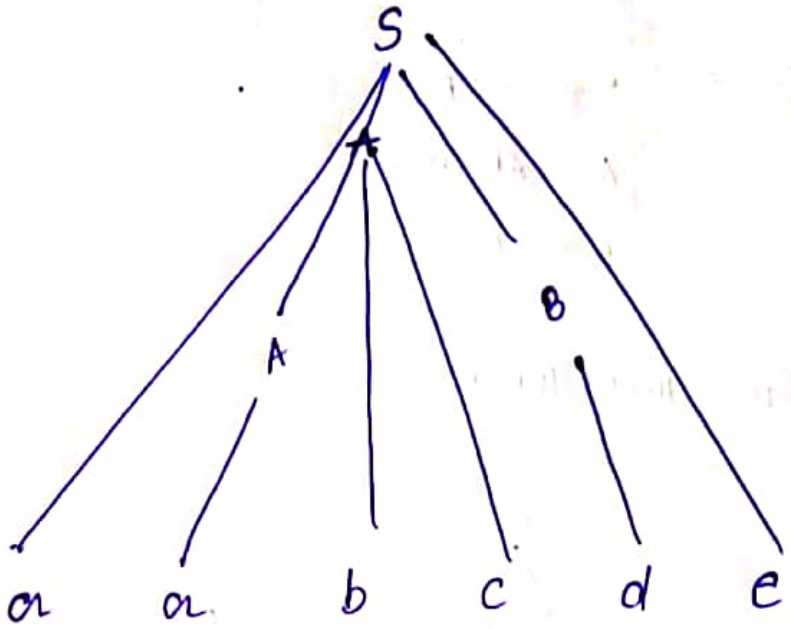
$B \rightarrow d$

top down parser



∴ the generated string is aabcde

Bottomup Parser.



∴ the generated string is aabcde.

8) write about derivation trees.

. Derivation Tree:

A derivation tree is a tree diagram that shows how a string (sentence) can be generated from a context-free Grammar (CFG) by applying production rules step by step.

→ It is used in syntax analysis (compiler design) to check whether a string belongs to given language.

Elements of a Derivation Tree.

1. Root node → the start symbol of the grammar.

2. Internal Nodes → Represent non-terminals (variable)
3. leaf Nodes → Represent terminals or ε
4. Edges → Show how a production rule is applied (from left-hand side - right hand side).

Types of Derivations.

1, leftmost Derivation - Always expand the leftmost non-terminal first.

2, Rightmost Derivation - Always expand the rightmost non-terminal first.

→ Based on the product length and point view have to choose.

→ Based on constrain we can develop a leftmost tree without unambiguous.

Construct CFG using left Most String w = aabaaa using production and Right Most derivation also.

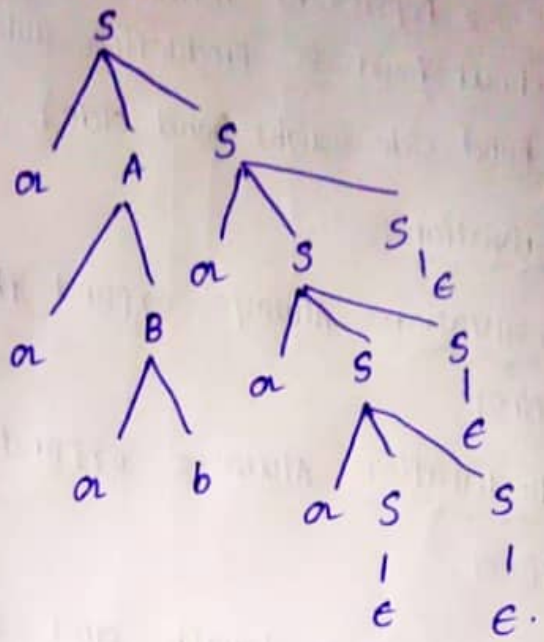
S → aAS/aSS/ε

A → aB/ab

B → bS/Aab

aAS
aabS
aabaSS
aabaSSS
aabaaSS
aabaaaS
aabaaaε

=) "aabaaa".



Right most derivation:

S =) a AS

=) a A aSS

=) a AaSaSS → {ε}

=) aAaSaS → {ε}

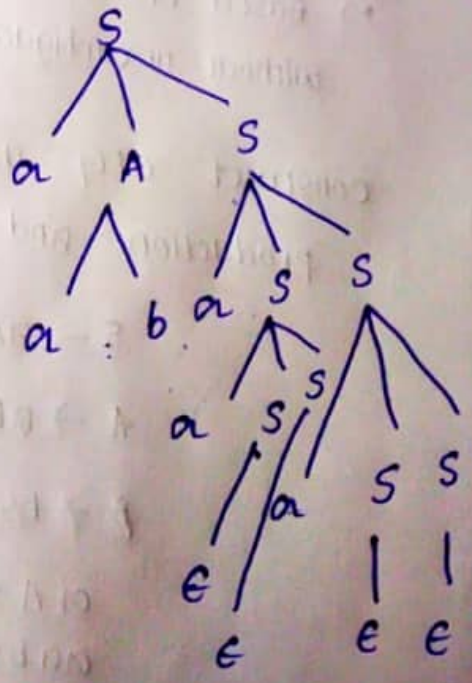
=) aAaSa

=) aAaaSSa → {ε}

=) aAaaSa → {ε}

=) aAaaa

=) 'aabaaa'.



9) what is ambiguity write down case-1 ambiguity and removing solutions for that.

Ambiguity in context-free Grammar (CFG)

A grammar is said to be ambiguous if there exist at least one string in the language that can have more than one leftmost derivation, rightmost derivation, or parse tree

→ the string must be generated by one derivation tree

case 1: Ambiguity in Arithmetic Expressions.

Let is take a grammar for arithmetic expressions.

Grammar G1:

$E \rightarrow E+E \mid E*E \mid (E) \mid id.$

Example string:

id + id * id.

this string can be parsed in two ways.

1. (id + id) * id → addition first, then multiplication
2. (id + (id * id)) → multiplication first, then addition

So, two different parse tree exist → Grammar is ambiguous.

→ To avoid ambiguity we have to follow 2 rules.

1. Associate
2. Precedence.

- Multiplication (*) has higher precedence than addition (+).
- Both operators are usually left-associate.

We rewrite grammar as:

Unambiguous Grammar (G2):

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Explanation:

E handles addition

T handles multiplication

F handles operands

Now, for the same string

id + id * id.

It can only be parsed as:

id + (id * id)

because * has higher precedence.

thus, ambiguity is removed.

10) Pushdown automata with example.

Defination: A Pushdown Automaton (PDA) is an abstract machine like a Finite Automation (FA), but with an extra memory component called a stack.

- FA has only finite memory.

- PDA has an unbounded stack allowing it to recognize Context - free languages.
- So, PDA is useful in parsing programming languages, handling nested structure (like paienthese, recursion, etc).

Formal Defination

A PDA is a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where :

Q : Finite set of states

Σ : Input alphabet

Γ : Stack alphabet

δ : Transition function

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to P(Q \times \Gamma^*)$$

→ Takes current state, input symbol, and top of stack
→ Returns new state and stack changes

- q_0: start state
- Z_0: Initial stack symbol
- F: set of final states

Working of PDA.

1, PDA reads input from left to right

2, It use stack operations

- push
- pop
- No operation

- Ends in final state
- Stack becomes Empty.

Example 1: PDA for $L = \{a^n b^n \mid n \geq 0\}$

This is the classic balanced language (same number of a's followed by b's).

- For every a read → push into stack
- For every b read → pop from stack.
- If stack becomes empty exactly when input is finished → string is accepted

PDA construction:

States $Q = \{q_0, q_1, q_2\}$

Input alphabet $\Sigma = \{a, b\}$

Stack alphabet $\Gamma = \{A, Z_0\}$ (A = symbol for a, Z_0 = bottom marker)

Start state : q_0

Initial stack symbol : Z_0

Final state : q_f

Transitions δ:

1, on reading a: Push A

$$\delta(q_0, a, Z_0) = (q_0, AZ_0),$$

$$\delta(q_0, a, A) = (q_0, AA)$$

2, on reading b: Pop A

$$\delta(q_0, b, A) = (q_0, \epsilon)$$

3, When input is finished & stack has only Z_0: move to final state

$$\delta(q_0, \epsilon, Z_0) = (q_f, Z_0)$$

Trace Example:

Input: aaabbb

- Read a: push A → stack = A
- Read a: push A → stack = AA
- Read a: push A → stack = AAA
- Read b: POP A → stack = AA
- Read b: POP A → stack = A
- Read b: POP A → stack = ε
- End of input tempty stack → Accept