**Chapter 7: CONTEXT-FREE GRAMMAR**

**Topic – 1: Introduction To CFG**

**Introduction**

* **CFG** stands for **Context-Free Grammar**.
* **Formal language:** A language whose grammar needs to be **logical & non-contradictory**, unlike natural languages like Hindi, English, French etc.
* **CFG** uses **formal grammar**.
* It is used for generating all possible **strings combination**.

**CFG Symbol**

* It is defined by symbol **G**.

**G = (V, T, P, S)**

**G = Contains rules to produce a string**

**V = Set of non-terminal symbols (denoted by capital letters)**

**T = Set of terminal symbols (denoted by lowercase letters)**

**P = Set of rules to replace non-terminal symbols (left to production) with terminal or other non-terminal symbols**

**S = Start symbol for a string**

**Production Rules**

* These are rules kept as per the **nature of expression**.
* For example, **\*** may provide a null value.
* We set up such rules seeing the expressions or languages.
* Our goal is to produce **production rules** for the givenexpressions or languages.

**Terminologies**

* **Terminals:** Symbols which **can’t** be **decomposed or replaced**, as that will be against the rules of language.
* For example, any **keywords** or **notations** etc.
* **Non-terminals:** **Replaceable** & sometimes **breakable** symbols.
* For example, a **variable** or **function name** etc.

**Example**

**Ques: Construct a CFG for having any number of a’s from Σ = {a}.**

**Ans:**

**R = a\***

**Production rules (P) for R:**

**S 🡪 aS [Rule 1]**

**S 🡪 Ԑ [Rule 2]**

**On RHS we are adding each character from the forming string.**

**We start from S & first element in RE is Ԑ. So, the rule 2.**

**Let’s say we want to generate a string 'aaaaaa' using our CFG:**

**S**

**aS**

**aaS**

**aaaS**

**aaaaS**

**aaaaaS**

**aaaaaaS**

**aaaaaaԐ**

**aaaaaa**

**Example – II**

**Ques: Construct CFG for the following expression.**

**RE = (0+1)\***

**Ans:**

**Production rules (P):**

**S 🡪 0S**

**S 🡪 1S**

**S 🡪 Ԑ**

**Combining first two rules, we can write it as:**

**S 🡪 0S | 1S**

**Example – III**

**Ques: Construct CFG for the following language.**

**L = {wcwR | where w Є (a,b)\*}**

**Ans:**

**The language above says that (a,b)\* can be fit in place of w.**

**wR means reverse of w.**

**L = {aca, bcb, abcba, bacab, aacaa, bbacabb}**

**Production rules (P):**

**S 🡪 aSa**

**S 🡪 bSb**

**S 🡪 c**

**Tip!**

**🡪 Focus on the pattern of language or expression to understand what production rules to make.**

**Note!**

**🡪 Notice that the P for S’s value is actually the first expected element in the expression/language set.**

**🡪 Like we in 1st example, we expected Ԑ to be the first element in RE set.**

**🡪 And c to be so at language set in the recent question.**

**Topic – 2: Derivation**

**Introduction**

* ***Derivation*** is a **set of production rules**.
* It is highly used in **parsing**.

**Parsing Decisions**

* **Decision 1 –** Non-terminals to be replaced.
* **Decision 2 –** Production rule to replace our non-terminals.

**Leftmost Derivation**

* Input string is scanned **left to right**.
* Then we **replace** non-terminals cleverly.
* Means we choose a rule from the **production set** & then **keep making changes** to it from **left side**.
* See the example below to understand how.

**Production rules:**

**E = E + E**

**E = E – E**

**E = a | b**

**Input:**

**a – b + a**

**Leftmost derivation:**

**E = E + E**

**E = E – E + E**

**E = a – b + a**

**Rightmost Derivation**

***\*Now you know\****

**Note!**

**🡪 When deriving, we use either leftmost or rightmost derivation.**

**Warning!**

**🡪 Don’t start derivation with anything other than S.**

**🡪 Solve one-by-one, don’t right multiple Ԑ together.**

**Topic – 3: Derivation Tree**

**Introduction**

* **Graphical representation** of derivation.
* **Derivation tree** is also known as ***parse tree***.
* Follows the **operation precedence** rule.

**Parse Tree Properties**

* **Rule 1:** Root node shows **starting symbols**.
* **Rule 2:** Derivation is read from **left to right**.
* **Rule 3:** Leaf nodes are always **terminals**.
* **Rule 4:** Internal nodes are always **non-terminals**.

**Example**

**Production rules:**

**E = E + E**

**E = E \* E**

**E = a | b | c**

**Input:**

**a \* b + b**

**Derivation:**

**E = E + E**

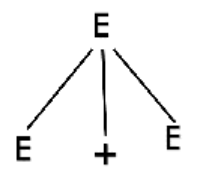
**E = E \* E + E**

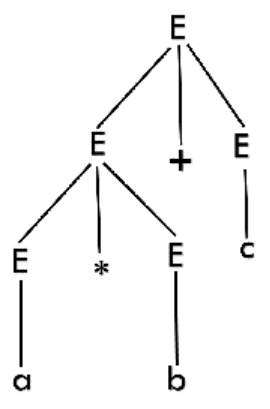
**E = a \* E + E**

**E = a \* b + E**

**E = a \* b + c**

**Solution:**

****

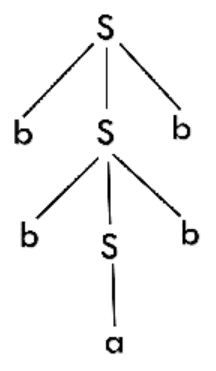
****

**Example - II**

**Ques: Draw derivation tree for string "bbabb" for CFG:**

**S 🡪 bSb | a | b**

**Ans:**

****

**Warning!**

**🡪 Don’t make any other type of derivation other than leftmost derivation.**

**🡪 And solve CGF from right side.**

**Topic – 4: Ambiguity In Grammar**

**Introduction**

* A grammar is said to be **ambiguous** if we get **different** parse trees for its **leftmost** & **rightmost** derivation.
* Compiler **can’t** be built with ambiguous grammar.
* We do leftmost & rightmost derivation both to see if they are same.
* If no input string is given to us, we test it with **random string(s)**.

**Example**

**Production rules (P):**

**E 🡪 I**

**E 🡪 E + E**

**E 🡪 E \* E**

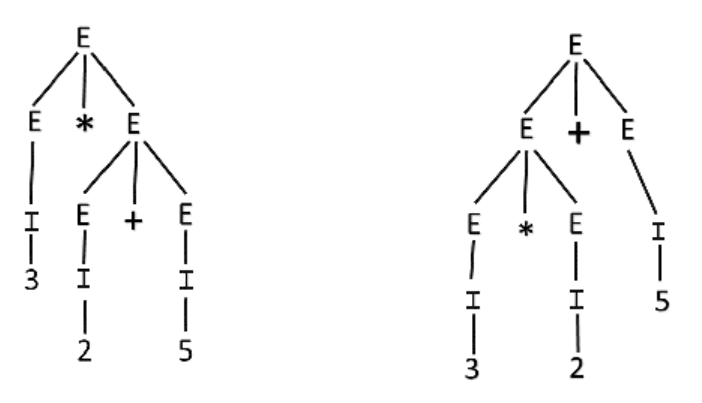
**E 🡪 (E)**

**I 🡪 Ԑ | 0 | 1 | 2 | … | 9**

**Sample input:**

**3 \* 2 + 5**

**Test results:**

****

**Example – II**

**Ques: Check ambiguity for given G with P:**

**S 🡪 aSb | SS**

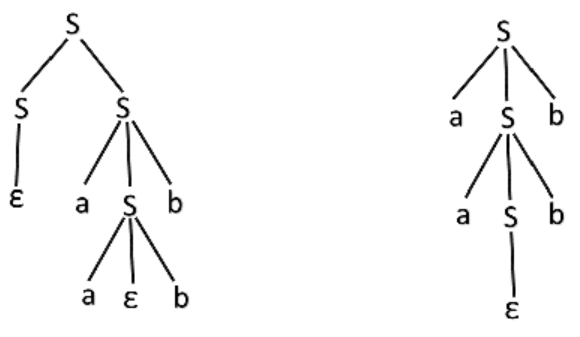
**S 🡪 Ԑ**

**Ans:**

**Sample Input:**

**aabb**

**Test results:**

****

**Topic – 5: Unambiguous Grammar**

**Introduction**

**Left associative operators – +, -, \*, /**

**Right associative operator – ^**

**Balancing Sides**

* If **left associative operators** are used, then we use **left recursion** to production rule.
* We basically **balance** both the **left & right side** of the terminal symbol by doing so.

**X 🡪 Xa**

**Note!**

**🡪 We add the non-terminal on right side in case of left associative operators.**

* If **right associative operators** are used, then we use **right recursion** to production rule.

**X 🡪 aX**

**Steps Involved**

* **Step 1:** Make **left recursion** for **leftmost associate** operators.
* **Step 2:** Make **right recursion** for **rightmost associate** operator.
* **Step 3:** Check which parts of each production rule is causing **unambiguity**.
* **Step 4:** Remove these parts of those rules.

**Example**

**Ambiguous rules:**

**S 🡪 AB | aaB**

**A 🡪 a | Aa**

**B 🡪 b**

**Filtered unambiguous rules:**

**S 🡪 AB**

**A 🡪 Aa | a**

**B 🡪 b**

**Note!**

**🡪 Removing ambiguity is more of a hit & trial method.**

**🡪 It generally consumes a lot of time & might not be much time efficient.**

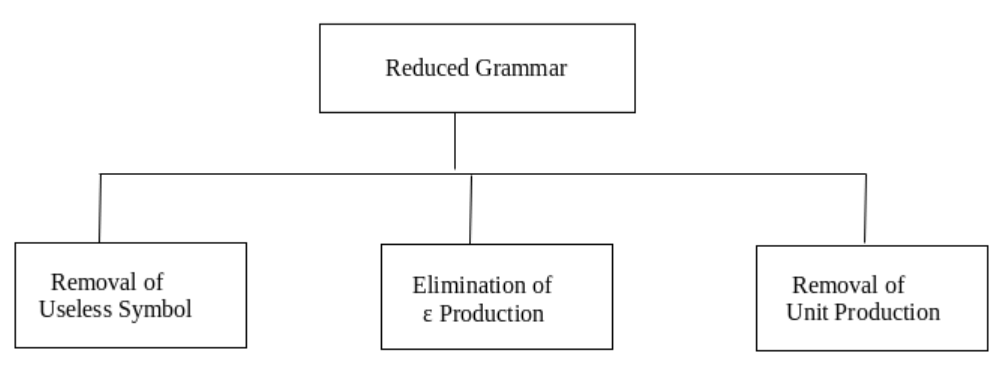
**Topic – 6: Simplification Of CFG**

**Introduction**

* Grammars can be **simplified/optimized** by eliminating **useless symbols**.

**Reduced Grammar Properties**

* **Property 1:** Each terminal & non-terminal must appear **at least once** when deriving a string.
* **Property 2:** There should be no **one-to-one** production rules among non-terminals like **X 🡪 Y** (**X**, **Y** are **non-terminals**).
* **Property 3:** There must be no **Ԑ** production until it is part of a language.



**Example**

**Given productions:**

**T 🡪 aaB | abA | aaT**

**A 🡪 aA**

**B 🡪 ab | b**

**C 🡪 ad**

**We will remove 'C' as it will never occur under any condition.**

**We will also remove 'A' as it creates endless recursion.**

**Adding Ԑ as an option to A can’t be done as one of the property we saw is about elimination of Ԑ production.**

**We have to remove 'A' related terminals too to ensure safe grammar.**

**Edited Productions**

**T 🡪 aaB | aaT**

**B 🡪 ab | b**

**Elimination Of Ԑ Production**

* **Step 1:** Remove all **non-terminals** that somehow lead to **Ԑ only**.
* **Step 2:** Convert (**A 🡪 a**) into (**A 🡪 x**) where **'a'** led to **Ԑ** earlier.

**Example – II**

**Given production rules:**

**S 🡪 XYX**

**X 🡪 0X | Ԑ**

**Y 🡪 1Y | Ԑ**

**For S 🡪 XYX, keeping one non-terminal or multiple as Ԑ gives:**

**YX, XY, XX, X, Y**

**So, our P becomes:**

**S 🡪 YX | XY | XX | X | Y**

**X 🡪 0X**

**Y 🡪 1Y**

**X & Y generate recursive strings, but we can’t remove them as they are in S & whole grammar destroys as we do that. So, final P:**

**S 🡪 YX | XY | XX | X | Y**

**X 🡪 0X | 0**

**Y 🡪 1Y | 1**

**Removing Unit Productions**

* **Step 1:** Remove terminals which produce **just non-terminals** or **unit production**.
* **Step 2:** **Replace** those terminals/units in other terminals with the non-terminals they produced.

**Example – III**

**Given P:**

**S 🡪 0A | 1B | C**

**A 🡪 0S | 00**

**B 🡪 1 | A**

**C 🡪 01**

**We see, 'C' produces only non-terminals. So, our new P:**

**S 🡪 0A | 1B | 01**

**A 🡪 0S | 00**

**B 🡪 1 | A**

**We should also remove 'A' from 'B' as it’s a unit production.**

**Final P:**

**S 🡪 0A | 1B | 01**

**A 🡪 0S | 00**

**B 🡪 1 | 0S | 00**