Context-free Grammars

Marks: 20-21

Introduction :-

⇒ we have learnt effective use of regular expressions for defining regular languages. But there are certain languages which cannot be defined by regular expressions eg. we cannot write a regular expression for checking well ~~defined~~ formed parenthesis etc. so in order to cover all these things and more complicated features leads us to the idea of a context-free ~~languages~~ Grammar for the languages.

⇒ while compiling the program, it is checked syntactically first and semantically latter. For checking the correctness of the program syntactically we need CFG.

: Defination :-

⇒ A context free grammar (CFG) is defined as 4-tuples (V_N, Σ, P, S) (S: starting non-terminal)

where, V_N = set of non-terminal symbol

Σ = set of terminal symbol

P = Production rule

S = starting non-terminal symbol.

Production rule is in the form,

one non-terminal → finite string of terminals and / or non-terminal.

i.e $A \to a$ where $A \in V_N$ and $a \in (V_N \cup \Sigma)^*$

* Application

⇒ The language generated by the context-free grammars are called context-free languages are applied in parser design.

⇒ It is also useful for describing block structure in programming languages.

⇒ CFGs are useful for describing arithmetic expressions, with arbitary nesting of balanced paranthesis.

* Derivations

⇒ The production rules are used to derive certain strings. The generation of language using specific rules is called derivation.

⇒ notations : $\Rightarrow$ and $\overset{*}{\Rightarrow}$

⇒ If $\alpha \to \beta$ be a production of P in CFG and a and b are strings in $(V_N \cup \Sigma)^*$ then $a\alpha b \Rightarrow a\beta b$.

Example 1 Construct a context-free grammar G generating all integers (with sign).

Soln Let $G = (V_N, \Sigma, P, S)$ where $V_N = \{S, \langle sign\rangle, \langle digit\rangle, \langle integer\rangle\}$

$\Sigma = \{0, 1, 2, 3 \ldots 9, +, -\}$

P consists of $S \to \langle sign\rangle \langle integer\rangle$

$\langle sign\rangle \to + | -$

$\langle integer\rangle \to \langle digit\rangle \langle integer\rangle | \langle digit\rangle$

$\langle digit\rangle \to 0|1|2|3|4|5|6|7|8|9$.

L(G) is the set of all integers.

L(G) is the language of G. It is the set of terminal strings that have derivations from the start symbol.

four components in CFG:-

1. There is the finite set of symbols that form the strings of language being defined. we call this alphabet the terminal or terminal symbol.

2. There is a finite set of variables, also called non-terminal symbol. Each variable represent the language i.e set of strings.

3. One of the variable represent the language being defined. It is called start symbol. It is denoted by S.

4. There is a finite set of rules or productions that represent the recursive defination of a language. Each productions consist of

(a) A variable that is being partially defined by the production. This variable is often called the head of production.

(b) The production symbol →

(c) A strings of zero or more terminals and variables. This string is called the body of production; represents one way to form strings in the language of the variable of the head

vote:-

↳ The variable symbols are represented by capital letters.

↳ The terminal symbols are represented by lowercase letters.

↳ One of the variable is designed as a start variable, It usually occurs in the left hand side of the production rule.

Example

$S \to \varepsilon$

$S \to 0S1$

This is the CFG defining the grammar of all the strings with equal no. of 0's followed by equal no. of 1's.

Here, Two rules define the production P,

$\varepsilon, 0, 1$ are the terminal defining Σ or T,

S is the non-terminal symbol defining V.

and S is the start symbol from where production starts.

Example following is a CFG for the language $L = \{\omega c \omega^R | \omega \in (a,b)^*\}$

Solution Let G be the CFG for the language $L = \{\omega c \omega^R | \omega \in (a,b)^*\}$

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

Here, $V_N = \{S\}$

$\Sigma = \{a, b, c\}$

and P is given by

$S \to aSa$

$S \to bSb$

$S \to c$

let us check that abbcbba can be derived from given CFG

$S \Rightarrow aSa$ (use the $S \to aSa$)

$\Rightarrow abSba$ (use the $S \to bSb$)

$\Rightarrow abbSbba$ (use the $S \to bSb$)

$\Rightarrow abbcbba$ (use the $S \to c$)

Example2 Write a CFG, which generates string of balanced parenthesis.

Solution This grammar will accept the balanced left and right parenthesis. for example ( ) ( ) is acceptable. ((( ))) this is also acceptable.

Let us design the CFG for this,

Let CFG be,

$G_1 = (V_N, \Sigma, P, S)$ where

V_N = set of non-terminal = $\{S\}$

$\Sigma = \{(,)\}$

and set of production P is given by

$S \rightarrow SS$

$S \rightarrow (S)$

$S \rightarrow \varepsilon$

Now we see what this grammar generates

$S \Rightarrow SS$

$\Rightarrow (S)S$

$\Rightarrow (S)SS$

$\Rightarrow (S)(S)S$

$\Rightarrow (S)(S)(S)$

$\Rightarrow ()()()$

Thus above grammar will always generates balanced pair of parenthesis.

Example write a CFG, which generates palindrome for binary numbers.

Solution Grammar will generate palindrome for binary numbers, that is 00, 010, 11, 101 ...

Let CFG be $G_1 = (V_N, \Sigma, P, S)$

V_N = set of non-terminals = $\{S\}$

Σ = set of terminals = $\{0, 1\}$

and P is the Production Rule defined as

$$S \to 0S0 \mid 1S1$$

$$S \to 0 \mid 1 \mid \varepsilon$$

Obviously this grammar generates palindrome for binary numbers it can be seen by the following derivation.

$$S \Rightarrow 0S0$$
$$\Rightarrow 01S10$$
$$\Rightarrow 010S010$$
$$\Rightarrow 0101010 \text{ which is palindrome.}$$

Example write a CFG for a regular expressions.

$$r = 0^*1(0+1)^*$$

Solution Let us analyse regular expression

$$r = 0^*1(0+1)^*$$

Clearly regular expression is the set of strings which starts with any number of 0's followed by one and end with any combination of 0's and 1's.

Let CFG be $G = (V_N, \Sigma, P, S)$

where,

V_N = set of non-terminal symbol

$= \{S, A, B\}$

$\Sigma = \{0, 1\}$

and productions P are defined as

$S \rightarrow A1B$

$A \rightarrow 0A \mid \varepsilon$

$B \rightarrow 0B \mid 1B \mid \varepsilon$

Let us see the derivation of the string 00101

$S \Rightarrow A1B$

$\Rightarrow 0A10B$

$\Rightarrow 00A101B$

$\Rightarrow 00101$

Clearly, G is the CFG for regular expression r.

Example Write a CFG which generates strings having equal number of a's and b's.

Solution Let CFG be $G = (V_N, \Sigma, P, S)$

$V_N = \{S\}$

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

where P is defined as,

$S \rightarrow aSbS \mid bSaS \mid \varepsilon$

Let us derive a string $\omega = bbabaabbaa$

$S \Rightarrow b\underline{S}aS$

$\Rightarrow bb\underline{S}aSaS$

$\Rightarrow bbaSb\underline{S}aSaS$

$\Rightarrow bbaSba\underline{S}bSaSaS$

$\Rightarrow bba\underline{S}baaSbSbSaSaS$

$\Rightarrow bbabaaS\underline{b}SbSaSaS$

$\Rightarrow bbabaaba\underline{S}bSaSaS$

$bbabaabb\underline{S}aSaS$

$\Rightarrow$ ~~$bbabaabab\underline{S}aSaS$~~

$\Rightarrow bbabaabbaSaS$

$\Rightarrow bbabaabbaaS$

$\Rightarrow bbabaabbaa$

Example Design a CFG for the language $L = \{a^n b^m : n \neq m\}$

solution If $n \neq m$ then there are only two cases are possible

Case 1. $n > m$

let us say language L on condition $n > m$ is

L_1 and $L_2 = \{a^n b^m : n > m\}$

Let G_L be the CFG for the language L_1,

$G_L = (V_n^L, \Sigma^L, P^L, S^L)$

$V_N^L = \{S_1, A, S\}$

$\Sigma^L = \{a, b\}$

Then production P_L are defined as.

$$S^1 \rightarrow AS_1$$

$$S_1 \rightarrow aS_1b \mid \varepsilon$$

$$A \rightarrow aA \mid a$$

Case 2 $\quad n<m$

let us say L on condition $n<m$ is L_2

$$L_2 = \{0^n 1^m : n<m\}$$

and let CFG for G_2 be $G_{12} = (V_N^2, \Sigma^2, P^2, S)$

$$V_N^2 = \{S^2, S_2, B\}$$

$$\Sigma^2 = \{a, b\}$$

P^2 are defined as.

$$S^2 \rightarrow S_2B$$

$$S_2 \rightarrow aS_2b \mid \varepsilon$$

$$B \rightarrow bB \mid b$$

By combining G_L and G_2 we can write the CFG for L as

$$S \rightarrow S^1 \mid S^2$$

* Derivation

⇒ CFG generates string according to the following process called derivation.

ⓐ We start by writing down the start symbol of the CFGs.

ⓑ At each step of the derivation, we may replace any non-terminal symbol of the string generated so far by the right hand side of any production that has the symbol on the left.

ⓒ The process ends when it is impossible to apply a step of type described in ⓑ.

⇒ If the string at the end of this process consists entirely of terminals, it is a string in the language generated by the grammar called yield.

⇒ There are two approaches of derivation

1. Body to head approach (Bottom up)
2. Head to Body (Top down)

1. Body to head :-

⇒ Here, we take strings known to be in the language of each of the variables of the body, concatenate them in the proper order with any terminals appearing in the body, the resulting string is the language of the variable in the head.

Consider grammar,

$S \rightarrow S+S$

$S \rightarrow S|S$

$S \rightarrow (S)$

$S \rightarrow S-S$

$S \rightarrow S*S$

$S \rightarrow a$

Here given a+(a*a)|a-a

now, applying body to head approach,

S.N	String inferred	Variable	Production	string used.
1.	a	S	S→a	
2.	a*a	S	S→S*S	string ①
3.	(a*a)	S	S→(S)	string ②
4.	(a*a)|a	S	S→S|S	string ③ and ①
5.	(a*a)|a-a	S	S→S-S	string ④ and ①
6.	a+(a*a)|a-a	S	S→S+S	string ⑤ and ①

Thus in this approach we start with any terminal appearing in the body and use the available rules from body to head.

2. Head to Body :-

Here, we used production from head to body. we expand the start symbol using a production, whose head is the start symbol. Here we expand the resulting string until all strings of terminal are obtained. Here we have two approaches

(a) Left most Derivation :-

Here, leftmost symbol (variable) is replaced first

(b) Right most Derivation :-

Here, rightmost symbol is replaced first.

Example Consider the grammar G1 with production

$S \to ass \mid b$

Now, we have

$S \Rightarrow ass$

$\Rightarrow aasss$

$\Rightarrow aabss$

$\Rightarrow aabasss$

$\Rightarrow aababss$

$\Rightarrow aababbs$

$\Rightarrow aababbb$

The sequence followed is left-most derivation

Now right-most derivation:

$S \Rightarrow ass$

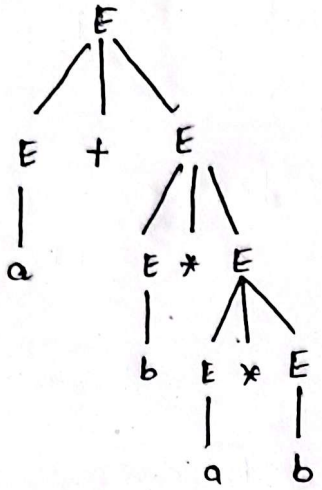
$\Rightarrow asb$

$\Rightarrow aassb$

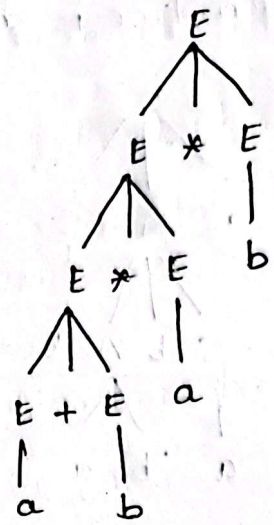
$\Rightarrow aasasbb$

$\Rightarrow aasabbb$

$\Rightarrow aababbb$



LMD parse tree



RMD parse tree

Example Consider the grammar G.

$S \rightarrow A1B$

$A \rightarrow 0A|\varepsilon$

$B \rightarrow 0B|1B|\varepsilon$

1. Construct the parse tree for 00101.

LMD

$S \rightarrow \underline{A}1B$

$\Rightarrow 0\underline{A}1B$

$\Rightarrow 00\underline{A}1B$

$\Rightarrow 00\varepsilon 1B$

$\Rightarrow 001\underline{B}$

$\Rightarrow 00101\varepsilon$

$\Rightarrow 00101$

RMD

$S \rightarrow A1\underline{B}$

$\Rightarrow A10\underline{B}$

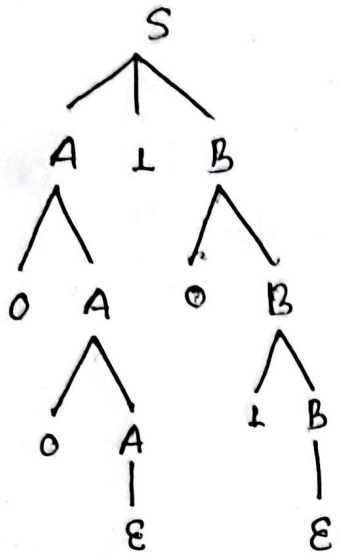
$\Rightarrow A101\underline{B}$

$\Rightarrow \underline{A}101\varepsilon$

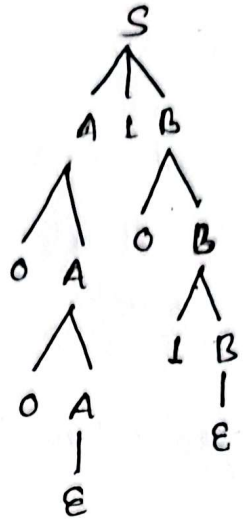
$\Rightarrow 0\underline{A}101$

$\Rightarrow 00\underline{A}101$

$\Rightarrow 00101$



LMD parse tree



RMD parse tree

* Ambiguity in Grammar:-

⇒ A grammar $G = (V, \Sigma, P, S)$ is said to be ambiguous if there is a string $\omega \in L(G)$ for ~~each~~ which we can derive two or more distinct derivation tree at S and yielding ω. In other words, a grammar is ambiguous if there exist more than one leftmost derivation or more than one rightmost derivation for the same string.

Example

$S \to AB \mid aaB$

$A \to a \mid Aa$

$B \to b$

for any string aab;
we have two leftmost derivation as.

$S \to \underline{A}B$

$\to AaB$

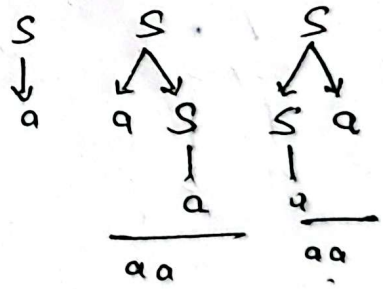
$\to aaB$

$\to aab$

Also, $S \to aaB$

$\to aab$

$S \to aS \mid Sa \mid a$



Here, we see there are two ways to generate string aa. Thus ambiguous.

for example

$G = (\{S\}, \{a, b, +, *\}, P, S)$ where P consist

$S \to S+S \mid S*S \mid a \mid b$, there are two derivation trees.

for a+a*b.

Soln ~~two have two left most derivation tree~~

$S \to S+S$

$\to a+S$

$\to a+S*S$

$\to a+a*S$

$\to a+a*b$

Also,

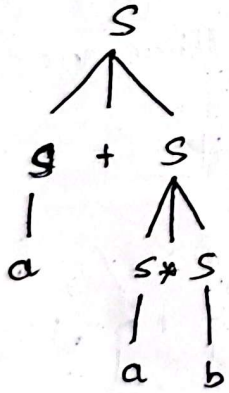
$S \to S*S$

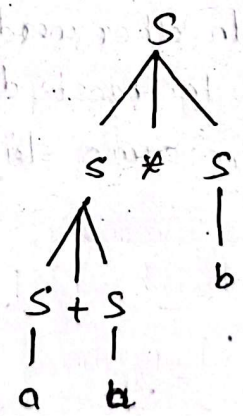
$\to S*b$

$\to S+S*b$

$\to S+a*b$

$\to a+a*b$





Thus, the given grammar is ambiguous.

* Removal of Ambiguity

⇒ In programming Languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible.

⇒ often we can achieve this by rewriting the grammar in an equivalent unambiguous form.

Example. Grammar

$E \rightarrow I$

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$I \rightarrow a|b|c$

is ambiguous and the ambiguity is due to the precedence of operator. If we correct the precedence then ambiguity may be removed.

Here, two causes of ambiguity in the grammar.

1. The precedence of operators is not respected.

2. The sequence of identical operators can group either from left or from the right. We would see two different parse trees for $E+E+E$ Since addition is associative, it doesnot mater whether we group from the left or the right, but to remove ambiguity we must take one.

The unambiguous grammar is

$E \to T$

$T \to F$

$F \to I$

$E \to E+T$

$T \to T*F$

$F \to (E)$

$I \to a|b|c$

The sole parse tree for a+b*c is