Context Free Grammar

Context Free Grammar is formal grammar, the syntax or structure of a formal language can be described using context-free grammar (CFG), a type of formal grammar. The grammar has four tuples: (V,T,P,S).

V - It is the collection of variables or nonterminal symbols.
T - It is a set of terminals.
P - It is the production rules that consist of both terminals and nonterminals.
S - It is the Starting symbol.

A grammar is said to be the Context-free grammar if every production is in the form of :

$G \rightarrow (V \cup T)^*$, where $G \in V$

- And the left-hand side of the G, here in the example can only be a Variable, it cannot be a terminal.
- But on the right-hand side here it can be a Variable or Terminal or both combination of Variable and Terminal.

Above equation states that every production which contains any combination of the ‘V’ variable or ‘T’ terminal is said to be a context-free grammar.
For example the grammar A = { S, a,b, P,S} having production :

- Here S is the starting symbol.
- {a,b} are the terminals generally represented by small characters.
- P is variable along with S.

  $S \rightarrow aS$
  $S \rightarrow bSa$
  but
  $a \rightarrow bSa$, or
  $a \rightarrow ba$ is not a CFG as on the left-hand side there is a variable which does not follow the CFGs rule.

In the computer science field, context-free grammars are frequently used, especially in the areas of formal language theory, compiler development, and natural language processing. It is also used for explaining the syntax of programming languages and other formal languages.

Limitations of Context-Free Grammar

- CFGs are less expressive, and neither English nor programming language can be expressed using Context-Free Grammar.
- Context-Free Grammar can be ambiguous means we can generate multiple parse trees of the same input.
- For some grammar, Context-Free Grammar can be less efficient because of the exponential time complexity.
- And the less precise error reporting as CFGs error reporting system is not that precise that can give more detailed error messages and information.

Classification of Context-Free Grammar

Context **F**ree **G**rammars (CFG) can be classified on the basis of following two properties:
1) Based on number of strings it generates.

- If CFG is generating finite number of strings, then CFG is **Non-Recursive** (or the grammar is said to be Non-recursive grammar)
- If CFG can generate infinite number of strings then the grammar is said to be **Recursive** grammar

During Compilation, the parser uses the grammar of the language to make a parse tree(or derivation tree) out of the source code. The grammar used must be unambiguous. An ambiguous grammar must not be used for parsing.

2) Based on number of derivation trees.

- If there is only 1 derivation tree then the CFG is unambiguous.
- If there are more than 1 left most derivation tree or right most derivation or parse tree , then the CFG is ambiguous.

Examples of Recursive and Non-Recursive Grammars

Recursive Grammars

1) `S->SaS`

   `S->b`

The language(set of strings) generated by the above grammar is :{b, bab, babab,…}, which is infinite.

2) `S-> Aa`

   `A->Ab|c`

The language generated by the above grammar is :{ca, cba, cbba …}, which is infinite.

Note: A recursive context-free grammar that contains no useless rules necessarily produces an infinite language.

Non-Recursive Grammars

`S->Aa`

`A->b|c`

The language generated by the above grammar is :{ba, ca}, which is finite.

Types of Recursive Grammars

Based on the nature of the recursion in a recursive grammar, a recursive CFG can be again divided into the following:

- Left Recursive Grammar (having left Recursion)
- Right Recursive Grammar (having right Recursion)
- General Recursive Grammar(having general Recursion)

A) Left Recursive Grammar-

- A recursive grammar is said to be left recursive if the leftmost variable of RHS is same as variable of LHS.

OR

- A recursive grammar is said to be left recursive if it has **Left Recursion**.

Example-

$$S \rightarrow Sa / b$$

(Left Recursive Grammar)

- Left recursion is considered to be a problematic situation for Top down parsers.
- Therefore, left recursion has to be eliminated from the grammar.

Elimination of Left Recursion

Left recursion is eliminated by converting the grammar into a right recursive grammar.

If we have the left-recursive pair of productions-

$A \rightarrow A\alpha / \beta$

(Left Recursive Grammar)

where β does not begin with an A.

Then, we can eliminate left recursion by replacing the pair of productions with-

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \in$

(Right Recursive Grammar)

This right recursive grammar functions same as left recursive grammar.

B) Right Recursive Grammar-

- A recursive grammar is said to be right recursive if the rightmost variable of RHS is same as variable of LHS.

OR

- A recursive grammar is said to be right recursive if it has right recursion.

Example-

$$S \rightarrow aS / b$$

(Right Recursive Grammar)

3. General Recursion-

The recursion which is neither left recursion nor right recursion is called as general recursion.

Example-

$S \rightarrow aSb / \in$

Note-01:

The grammar which is either left recursive or right recursive is always unambiguous.
Examples-

- S → aS / b (Unambiguous Grammar)
- S → Sa / b (Unambiguous Grammar)

Note-02:

The grammar which is both left recursive and right recursive is always ambiguous.
Example-

E → E + E / E – E / E x E / id
(Ambiguous Grammar)

Note-03:

Left recursive grammar is not suitable for Top down parsers.

- This is because it makes the parser enter into an infinite loop.
- To avoid this situation, it is converted into its equivalent right recursive grammar.
- This is done by eliminating left recursion from the left recursive grammar.

Note-04:

The conversion of left recursive grammar into right recursive grammar and vice-versa is decidable.

PRACTICE PROBLEMS BASED ON LEFT RECURSION ELIMINATION-

Problem-01:

Consider the following grammar and eliminate left recursion-

A → ABd / Aa / a
B → Be / b

Solution-

The grammar after eliminating left recursion is-

A → aA'
A' → BdA' / aA' / ∈
B → bB'
B' → eB' / ∈

Problem-02:

Consider the following grammar and eliminate left recursion-

E → E + E / E x E / a

Solution-

The grammar after eliminating left recursion is-

E → aA
A → +EA / xEA / ∈

Ambiguous vs Unambiguous Grammar :

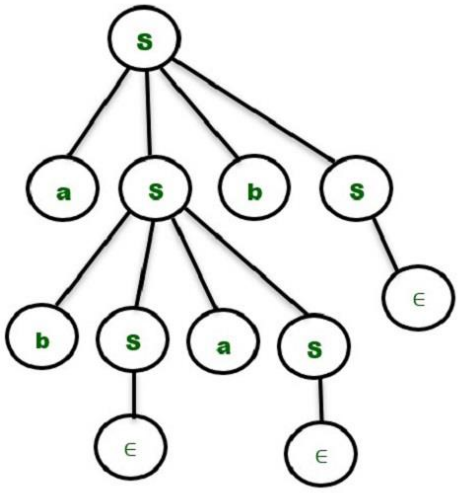
The grammars which have more than one derivation tree or **parse tree** are ambiguous grammars. These grammar is not parsed by any **parsers**.

Example-

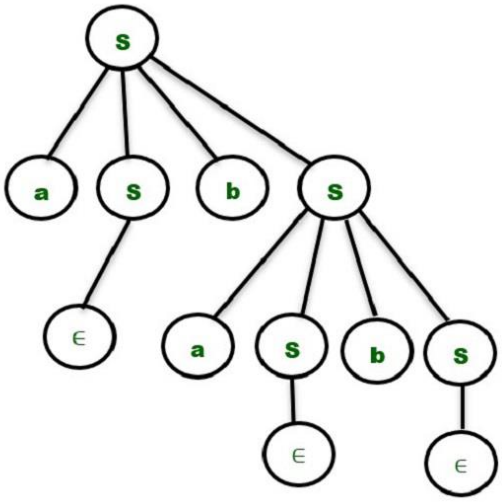
1. Consider the production shown below –

S->aSbS | bSaS | ∈

Say, we want to generate the string **"abab"** from the above grammar. We can observe that the given string can be derived using two parse trees. So, the above grammar is ambiguous.



Parse Tree 1



Parse Tree 2

The grammars which have only one derivation tree or parse tree are called unambiguous grammars.

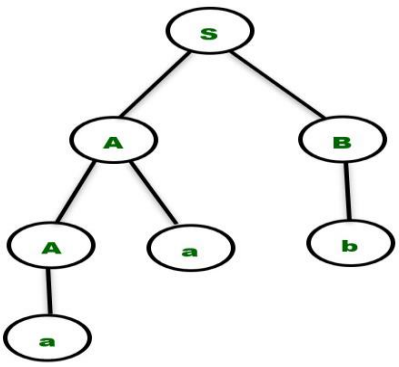
2. Consider the productions shown below –

S -> AB

A -> Aa | a

B -> b

For the string "aab" we have only one Parse Tree for the above grammar as shown below.



It is important to note that there are **no direct algorithms** to find whether grammar is ambiguous or not. We need to build the parse tree for a given input string that belongs to the language produced by the grammar and then decide whether the grammar is ambiguous or unambiguous based on the number of parse trees obtained as discussed above.

Note – The string has to be chosen carefully because there may be some strings available in the language produced by the unambiguous grammar which has only one parse tree.

Removal of Ambiguity :

We can remove ambiguity solely on the basis of the following two properties –

1. Precedence –

If different operators are used, we will consider the precedence of the operators. The three important characteristics are :

1. The level at which the production is present denotes the priority of the operator used.
2. The production at **higher levels** will have **operators with less priority**. In the parse tree, the nodes which are at top levels or close to the root node will contain the lower priority operators.
3. The production at **lower levels** will have operators with **higher priority**. In the parse tree, the nodes which are at lower levels or close to the leaf nodes will contain the higher priority operators.

2. Associativity –

If the same precedence operators are in production, then we will have to consider the associativity.

- If the associativity is left to right, then we have to prompt a left recursion in the production. The parse tree will also be left recursive and grow on the left side.
  +, -, *, / are left associative operators.
- If the associativity is right to left, then we have to prompt the right recursion in the productions. The parse tree will also be right recursive and grow on the right side.
  ^ is a right associative operator.

Example 1 – Consider the ambiguous grammar
E -> E-E | id

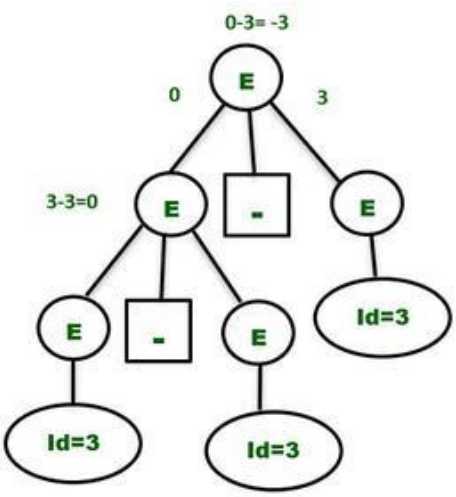
The language in the grammar will contain { id, id-id, id-id-id, ….}

Say, we want to derive the string **id-id-id.** Let’s consider a single value of id=3 to get more insights. The result should be :

3-3-3 *=-3*

Since the same priority operators, we need to consider associativity which is left to right.

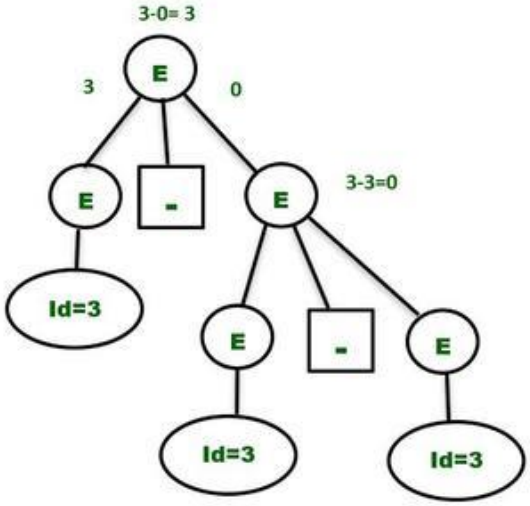
Parse Tree – The parse tree which grows on the left side of the root will be the correct parse tree in order to make the grammar unambiguous.



Left Associative

((3-3)-3) = (0-3) = -3

Correct



Right Associative

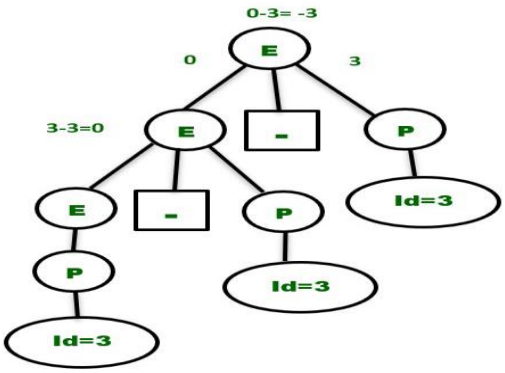
(3-(3-3)) = (3-0) = 3

Incorrect

So, to make the above grammar unambiguous, simply make the grammar **Left Recursive** by replacing the left most non-terminal E in the right side of the production with another random variable, say **P.** The grammar becomes :

E -> E – P | P
P -> id

The above grammar is now unambiguous and will contain only one Parse Tree for the above expression as shown below –



Similarly, the unambiguous grammar for the expression : **2^3^2** will be –

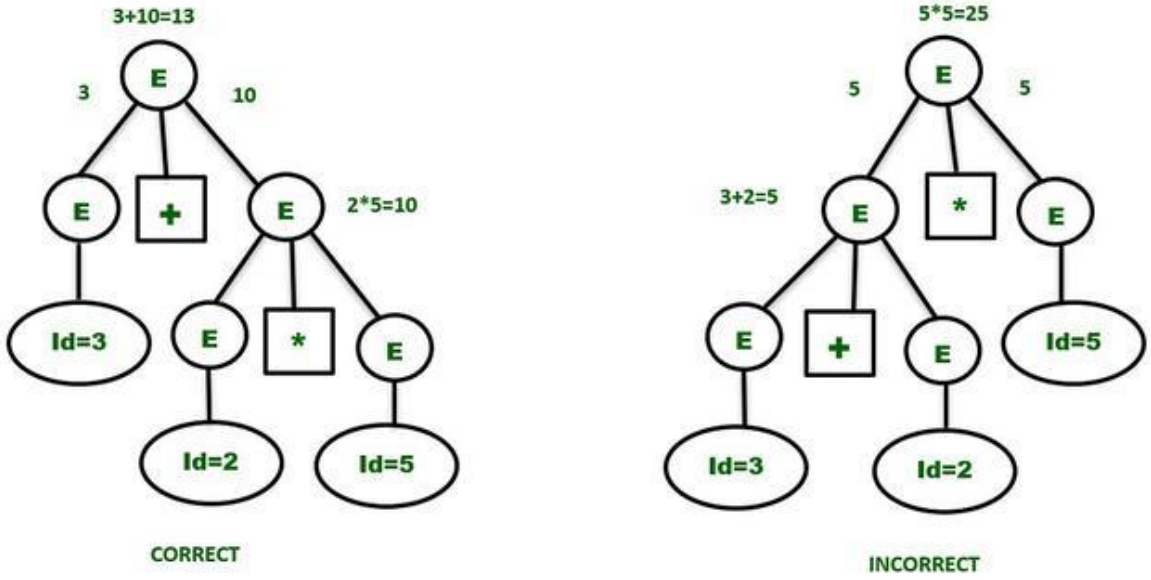
E -> P ^ E | P // Right Recursive as ^ is right associative.
P -> id

Example 2 – Consider the grammar shown below, which has two different operators :

E -> E + E | E * E | id

Clearly, the above grammar is ambiguous as we can draw two parse trees for the string **"id+id*id"** as shown below. Consider the expression :
3 + 2 * 5 // "*" has more priority than "+"

The correct answer is : (3+(2*5))=13



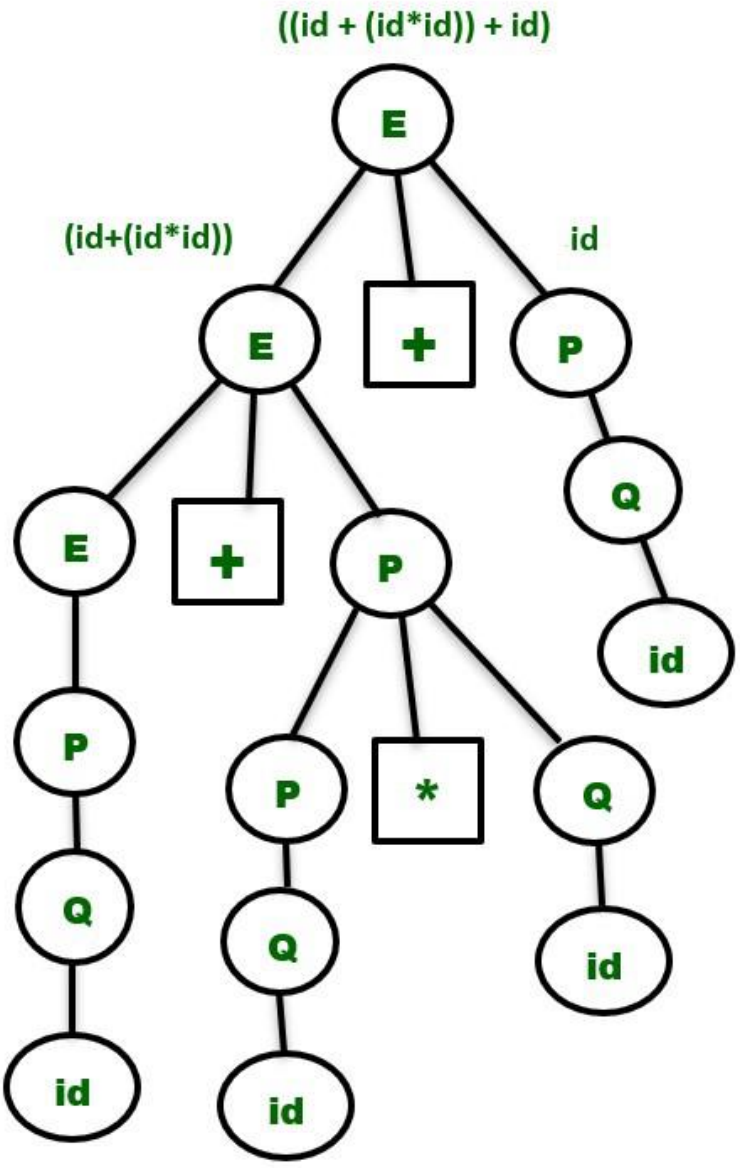
The "+" having the least priority has to be at the upper level and has to wait for the result produced by the "*" operator which is at the lower level. So, the first parse tree is the correct one and gives the same result as expected.

The unambiguous grammar will contain the productions having the highest priority operator **("*" in the example)** at the lower level and vice versa. The associativity of both the operators are **Left to Right**. So, the unambiguous grammar has to be **left recursive.** The grammar will be :

E -> E + P *// + is at higher level and left associative*
E -> P
P -> P * Q *// * is at lower level and left associative*
P -> Q
Q -> id
(or)

E -> E + P | P
P -> P * Q | Q
Q -> id

E is used for doing addition operations and **P** is used to perform multiplication operations. They are independent and will maintain the precedence order in the parse tree.
The parse tree for the string **" id+id*id+id "** will be –



Note : It is very important to note that while converting an ambiguous grammar to an unambiguous grammar, we shouldn't change the original language provided by the ambiguous grammar. So, the non-terminals in the ambiguous grammar have to be replaced with other variables in such a way that we get the same language as it was derived before and also maintain the precedence and associativity rule simultaneously.

This is the reason we wrote the production **E -> P** and **P -> Q** and **Q -> id** after replacing them in the above example, because the language contains the strings **{ id, id+id }** as well.
Similarly, the unambiguous grammar for an expression having the operators **-,*,^** is :

E -> E – P | P *// Minus operator is at higher level due to least priority and left associative.*
P -> P * Q | Q *// Multiplication operator has more priority than – and lesser than ^ and left* ***associative.***
Q -> R ^ Q | R *// Exponent operator is at lower level due to highest priority and right associative.*
R -> id

Also, there are some ambiguous grammars which can’t be converted into unambiguous grammars.