Context **F**ree **G**rammars (CFGs) are classified based on:

- Number of Derivation trees
- Number of strings

Depending on Number of Derivation trees, CFGs are sub-divided into 2 types:

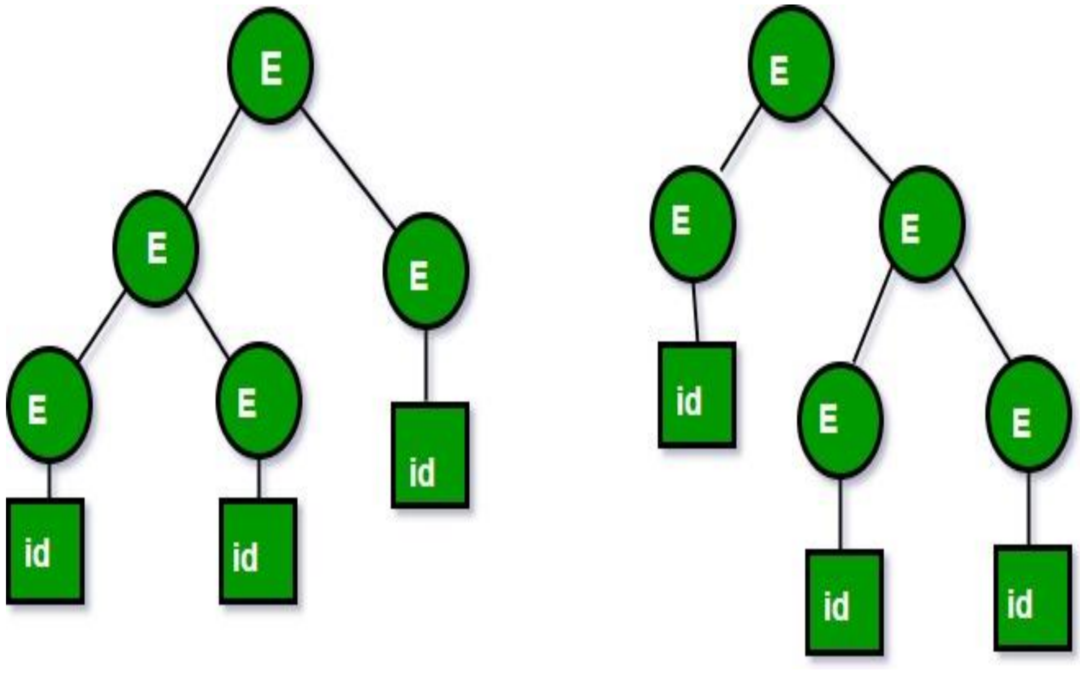
- Ambiguous grammars
- Unambiguous grammars

Ambiguous grammar:

A CFG is said to ambiguous if there exists more than one derivation tree for the given input string i.e., more than one **L**eft **M**ost **D**erivation **T**ree (LMDT) or **R**ight **M**ost **D**erivation **T**ree (RMDT). A context free grammar is called ambiguous if there exists more than one LMD or more than one RMD for a string which is generated by grammar.

Definition: G = (V,T,P,S) is a CFG is said to be ambiguous if and only if there exist a string in T* that has more than on parse tree.
where V is a finite set of variables.
T is a finite set of terminals.
P is a finite set of productions of the form, A -> α, where A is a variable and α ∈ (V ∪ T)* S is a designated variable called the start symbol.

For Example:
1. Let us consider this grammar: **E -> E+E|id**
We can create 2 parse tree from this grammar to obtain a string **id+id+id** :
The following are the 2 parse trees generated by left most derivation:



Both the above parse trees are derived from same grammar rules but both parse trees are different. Hence the grammar is ambiguous.

2. Let us now consider the following grammar:

Set of alphabets ∑ = {0,…,9, +, *, (,)}

E -> I

E -> E + E

E -> E * E

E -> (E)

I -> ε | 0 | 1 | … | 9

From the above grammar String **3*2+5** can be derived in 2 ways:

I) First leftmost derivation	II) Second leftmost derivation
E=>E*E	E=>E+E
=>I*E	=>E*E+E
=>3*E+E	=>I*E+E
=>3*I+E	=>3*E+E
=>3*2+E	=>3*I+E
=>3*2+I	=>3*2+I
=>3*2+5	=>3*2+5

Following are some examples of ambiguous grammars:

- S-> aS |Sa| Є
- E-> E +E | E*E| id
- A -> AA | (A) | a
- S -> SS|AB , A -> Aa|a , B -> Bb|b

$\boxed{\text{Homework}}$

Inherently ambiguous Language:

Let L be a Context Free Language (CFL). If every Context Free Grammar G with Language L = L(G) is ambiguous, then L is said to be inherently ambiguous Language. Ambiguity is a property of grammar not languages. Ambiguous grammar is unlikely to be useful for a programming language, because two parse trees structures (or more) for the same string(program) implies two different meanings (executable programs) for the program. An inherently ambiguous language would be absolutely unsuitable as a programming language, because we would not have any way of fixing a unique structure for all its programs.

Note : Ambiguity of a grammar is undecidable, i.e. there is no particular algorithm for removing the ambiguity of a grammar, but we can remove ambiguity by:

Disambiguate the grammar i.e., rewriting the grammar such that there is only one derivation or parse tree possible for a string of the language which the grammar represents.

Converting Ambiguous Grammar into Unambiguous Grammar-

- Causes such as left recursion, common prefixes etc. makes the grammar ambiguous.
- The removal of these causes may convert the grammar into unambiguous grammar.
- However, it is not always compulsory.

NOTE

It is not always possible to convert an ambiguous grammar into an unambiguous grammar.

Removing Ambiguity By Precedence & Associativity Rules-

An ambiguous grammar may be converted into an unambiguous grammar by implementing-

- Precedence Constraints
- Associativity Constraints

These constraints are implemented using the following rules-

Rule-01:

The precedence constraint is implemented using the following rules-

- The level at which the production is present defines the priority of the operator contained in it.
- The higher the level of the production, the lower the priority of operator.
- The lower the level of the production, the higher the priority of operator.

Rule-02:

The associativity constraint is implemented using the following rules-

- If the operator is left associative, induce left recursion in its production.
- If the operator is right associative, induce right recursion in its production.

PROBLEMS BASED ON CONVERSION INTO UNAMBIGUOUS GRAMMAR-

Problem-01:

Convert the following ambiguous grammar into unambiguous grammar-

$$R \rightarrow R + R \mid R . R \mid R^* \mid a / b$$

where ‘*’ is kleene closure, ‘+’ is union and ‘.’ is concatenation.

Solution-

To convert the given grammar into its corresponding unambiguous grammar, we implement the precedence and associativity constraints.

We have-

- Given grammar consists of the following operators-

+ , **.** , *****

- Given grammar consists of the following operands-

a , **b**

The priority/precedence order is-

(**a** , **b**) > ***** > **.** > **+**

where-

- **.** operator is left associative
- **+** operator is left associative

Using the precedence and associativity rules, we write the corresponding unambiguous grammar as-

E → E + T | T

T → T.F | F

F → F* | G

G → a | b

Unambiguous Grammar

OR

E → E + T | T

T → T . F | F

F → F* | a | b

Unambiguous Grammar

Problem-02:

Convert the following ambiguous grammar into unambiguous grammar-

bexp → bexp or bexp | bexp and bexp | not bexp | T | F

where bexp represents Boolean expression, T represents True and F represents False.

Solution-

To convert the given grammar into its corresponding unambiguous grammar, we implement the precedence and associativity constraints.

We have-

- Given grammar consists of the following operators-

  or , **and** , **not**

- Given grammar consists of the following operands-

  T , **F**

The priority/precedence order is-

(T , F) > **not** > **and** > **or**

where-

- **and** operator is left associative
- **or** operator is left associative

Using the precedence and associativity rules, we write the corresponding unambiguous grammar as-

bexp → bexp or M | M

M → M and N | N

N → not N | G

G → T | F

Unambiguous Grammar

OR

bexp → bexp or M | M

M → M and N | N

N → not N | T | F