Removal of ambiguity (Converting an Ambiguous grammar into Unambiguous grammar)

Last Updated : 11 Jun, 2021

Prerequisites : Context Free Grammars , Ambiguous Grammar, Difference between ambiguous and unambiguous grammar, Precedence and Associativity of operators, Recursive Grammar

In this article we are going to see the removal of ambiguity from grammar using suitable examples.

Ambiguous vs Unambiguous Grammar :

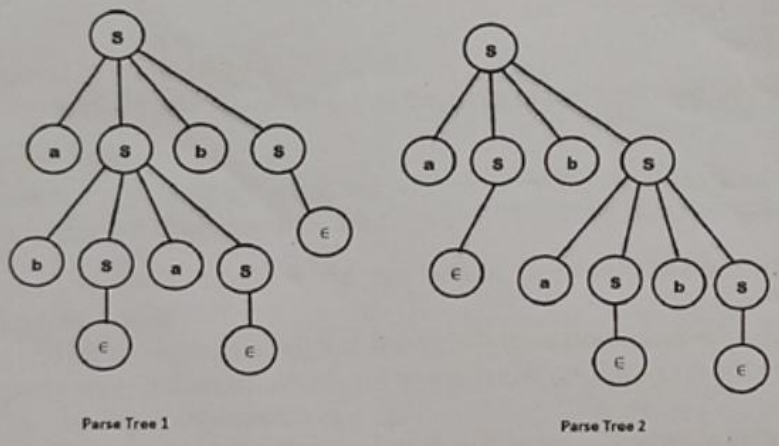
The grammars which have more than one derivation tree or parse tree are ambiguous grammars. These grammar is not parsed by any parsers.

Example-

1. Consider the production shown below –

$$S \rightarrow aSbS \mid bSaS \mid \square$$

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.



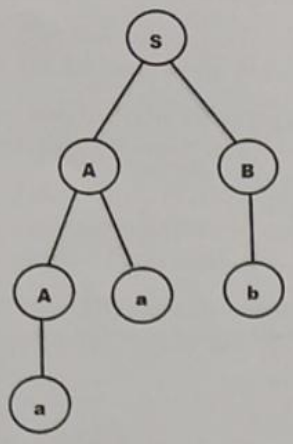
Parse Tree 1 Parse Tree 2

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

2. Consider the productions shown below –

$$S \rightarrow AB$$
$$A \rightarrow Aa \mid a$$
$$B \rightarrow b$$

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.

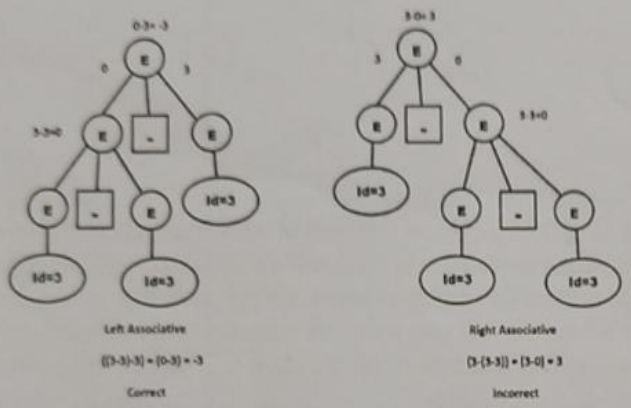
Example 1 – Consider the ambiguous grammar

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.

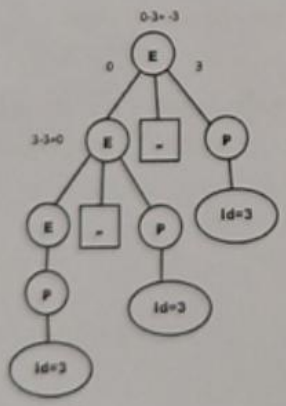
Parse Tree – The parse tree which grows on the left side of the root will be the correct parse tree in order to make the grammar unambiguous.



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 :

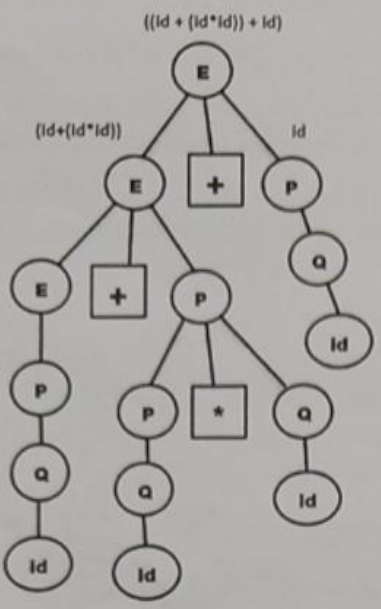
E -> E – P | P
P -> id

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



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

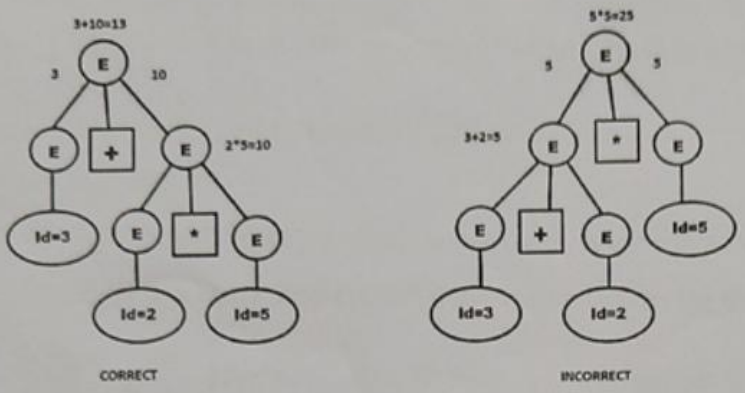
Also, there are some ambiguous grammars which can't be converted into unambiguous grammars.

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

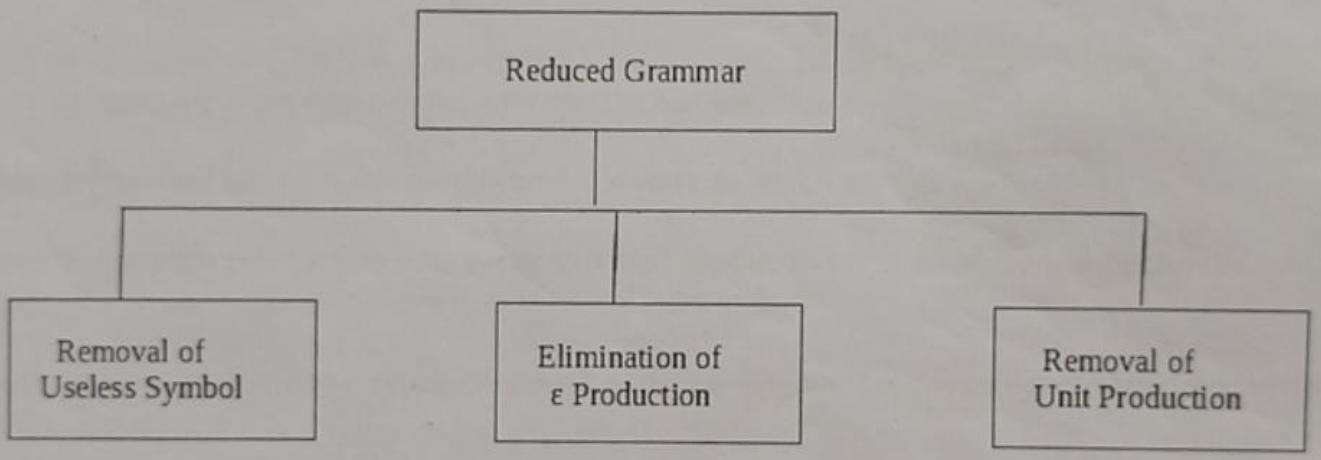
E is used for doing addition operations and **P** is used to perform multiplication operations. They are independent and will maintain the precedence order in the parse tree.
The parse tree for the string **" id+id*id+id "** will be –

Simplification of CFG

As we have seen, various languages can efficiently be represented by a context-free grammar. All the grammar are not always optimized that means the grammar may consist of some extra symbols(non-terminal). Having extra symbols, unnecessary increase the length of grammar. Simplification of grammar means reduction of grammar by removing useless symbols. The properties of reduced grammar are given below:

1. Each variable (i.e. non-terminal) and each terminal of G appears in the derivation of some word in L.
2. There should not be any production as $X \rightarrow Y$ where X and Y are non-terminal.
3. If ε is not in the language L then there need not to be the production $X \rightarrow \varepsilon$.

Let us study the reduction process in detail./p>



Removal of Useless Symbols

A symbol can be useless if it does not appear on the right-hand side of the production rule and does not take part in the derivation of any string. That symbol is known as a useless symbol. Similarly, a variable can be useless if it does not take part in the derivation of any string. That variable is known as a useless variable.

For Example:

1. $T \rightarrow aaB \mid abA \mid aaT$

2. $A \rightarrow aA$
3. $B \rightarrow ab \mid b$
4. $C \rightarrow ad$

In the above example, the variable 'C' will never occur in the derivation of any string, so the production $C \rightarrow ad$ is useless. So we will eliminate it, and the other productions are written in such a way that variable C can never reach from the starting variable 'T'.

Production $A \rightarrow aA$ is also useless because there is no way to terminate it. If it never terminates, then it can never produce a string. Hence this production can never take part in any derivation.

To remove this useless production $A \rightarrow aA$, we will first find all the variables which will never lead to a terminal string such as variable 'A'. Then we will remove all the productions in which the variable 'B' occurs.

Elimination of ε Production

The productions of type $S \rightarrow \varepsilon$ are called ε productions. These type of productions can only be removed from those grammars that do not generate ε.

Step 1: First find out all nullable non-terminal variable which derives ε.

Step 2: For each production $A \rightarrow a$, construct all production $A \rightarrow x$, where x is obtained from a by removing one or more non-terminal from step 1.

Step 3: Now combine the result of step 2 with the original production and remove ε productions.

Example:

Remove the production from the following CFG by preserving the meaning of it.

1. $S \rightarrow XYX$
2. $X \rightarrow 0X \mid \varepsilon$
3. $Y \rightarrow 1Y \mid \varepsilon$

Solution:

Now, while removing ε production, we are deleting the rule $X \rightarrow \varepsilon$ and $Y \rightarrow \varepsilon$. To preserve the meaning of CFG we are actually placing ε at the right-hand side whenever X and Y have appeared.

Let us take

1. $S \rightarrow XYX$

If the first X at right-hand side is ε. Then

1. $S \rightarrow YX$

Similarly if the last X in R.H.S. = ε. Then

1. $S \rightarrow XY$

If Y = ε then

1. $S \rightarrow XX$

If Y and X are ε then,

1. $S \rightarrow X$

If both X are replaced by ε

1. $S \rightarrow Y$

Now,

1. $S \rightarrow XY \mid YX \mid XX \mid X \mid Y$

Now let us consider

1. $X \rightarrow 0X$

If we place ε at right-hand side for X then,

1. $X \rightarrow 0$
2. $X \rightarrow 0X \mid 0$

Similarly Y → 1Y | 1

Collectively we can rewrite the CFG with removed ε production as

1. S → XY | YX | XX | X | Y
2. X → 0X | 0
3. Y → 1Y | 1

Removing Unit Productions

The unit productions are the productions in which one non-terminal gives another non-terminal. Use the following steps to remove unit production:

Step 1: To remove X → Y, add production X → a to the grammar rule whenever Y → a occurs in the grammar.

Step 2: Now delete X → Y from the grammar.

Step 3: Repeat step 1 and step 2 until all unit productions are removed.

For example:

1. S → 0A | 1B | C
2. A → 0S | 00
3. B → 1 | A
4. C → 01

Solution:

S → C is a unit production. But while removing S → C we have to consider what C gives. So, we can add a rule to S.

1. S → 0A | 1B | 01

Similarly, B → A is also a unit production so we can modify it as

1. B → 1 | 0S | 00

Thus finally we can write CFG without unit production as

Chomsky's Normal Form (CNF)

CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:

- Start symbol generating ε. For example, A → ε.
- A non-terminal generating two non-terminals. For example, S → AB.
- A non-terminal generating a terminal. For example, S → a.

For example:

1. G1 = {S → AB, S → c, A → a, B → b}
2. G2 = {S → aA, A → a, B → c}

The production rules of Grammar G1 satisfy the rules specified for CNF, so the grammar G1 is in CNF. However, the production rule of Grammar G2 does not satisfy the rules specified for CNF as S → aZ contains terminal followed by non-terminal. So the grammar G2 is not in CNF.

Steps for converting CFG into CNF

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as:

1. S1 → S

Where S1 is the new start symbol.

Step 2: In the grammar, remove the null, unit and useless productions. You can refer to the Simplification of CFG.

Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production S → aA can be decomposed as:

1. S → RA
2. R → a

Step 4: Eliminate RHS with more than two non-terminals. For example, S → ASB can be decomposed as:

1. S → RS
2. R → AS

Example:

Convert the given CFG to CNF. Consider the given grammar G1:

1. S → a | aA | B
2. A → aBB | ε
3. B → Aa | b

Solution:

Step 1: We will create a new production S1 → S, as the start symbol S appears on the RHS. The grammar will be:

1. S1 → S
2. S → a | aA | B
3. A → aBB | ε
4. B → Aa | b

Step 2: As grammar G1 contains A → ε null production, its removal from the grammar yields:

1. S1 → S
2. S → a | aA | B
3. A → aBB
4. B → Aa | b | a

Now, as grammar G1 contains Unit production S → B, its removal yield:

1. S1 → S
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Also remove the unit production S1 → S, its removal from the grammar yields: