Context-Free Grammar (CFG)

CFG stands for context-free grammar. It is is a formal grammar which is used to generate all possible patterns of strings in a given formal language. Context-free grammar G can be defined by four tuples as:

G = (V, T, P, S)

Where,

G is the grammar, which consists of a set of the production rule. It is used to generate the string of a language.

T is the final set of a terminal symbol. It is denoted by lower case letters.

V is the final set of a non-terminal symbol. It is denoted by capital letters.

P is a set of production rules, which is used for replacing non-terminals symbols(on the left side of the production) in a string with other terminal or non-terminal symbols(on the right side of the production).

S is the start symbol which is used to derive the string. We can derive the string by repeatedly replacing a non-terminal by the right-hand side of the production until all non-terminal have been replaced by terminal symbols.

Example 1:

Construct the CFG for the language having any number of a's over the set ∑= {a}.

Solution:

As we know the regular expression for the above language is

1. r.e. = a*

Production rule for the Regular expression is as follows:

1. S → aS rule 1
2. S → ε rule 2

Now if we want to derive a string "aaaaaa", we can start with start symbols.

1. S

2. aS
3. aaS rule 1
4. aaaS rule 1
5. aaaaS rule 1
6. aaaaaS rule 1
7. aaaaaaS rule 1
8. aaaaaaε rule 2
9. aaaaaa

The r.e. = a* can generate a set of string {ε, a, aa, aaa,.....}. We can have a null string because S is a start symbol and rule 2 gives S → ε.

Example 2:

Construct a CFG for the regular expression (0+1)*

Solution:

The CFG can be given by,

1. Production rule (P):
2. S → 0S | 1S
3. S → ε

The rules are in the combination of 0's and 1's with the start symbol. Since (0+1)* indicates {ε, 0, 1, 01, 10, 00, 11, ...}. In this set, ε is a string, so in the rule, we can set the rule S → ε.

Example 3:

Construct a CFG for a language L = {wcwR | where w € (a, b)*}.

Solution:

The string that can be generated for a given language is {aacaa, bcb, abcba, bacab, abbcbba, ...}

The grammar could be:

1. S → aSa rule 1
2. S → bSb rule 2
3. S → c rule 3

Now if we want to derive a string "abbcbba", we can start with start symbols.

1. S → aSa
2. S → abSba from rule 2
3. S → abbSbba from rule 2
4. S → abbcbba from rule 3

Thus any of this kind of string can be derived from the given production rules.

Example 4:

Construct a CFG for the language L = $a^n b^{2n}$ where n>=1.

Solution:

The string that can be generated for a given language is {abb, aabbbb, aaabbbbbb....}.

The grammar could be:

1. S → aSbb | abb

Now if we want to derive a string "aabbbb", we can start with start symbols.

1. S → aSbb
2. S → aabbbb

Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing, we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

We have two options to decide which non-terminal to be placed with production rule.

1. Leftmost Derivation:

In the leftmost derivation, the input is scanned and replaced with the production rule from left to right. So in leftmost derivation, we read the input string from left to right.

Example:

Production rules:

1. E = E + E
2. E = E - E
3. E = a | b

Input

1. a - b + a

The leftmost derivation is:

1. E = E + E
2. E = E - E + E
3. E = a - E + E
4. E = a - b + E
5. E = a - b + a

2. Rightmost Derivation:

In rightmost derivation, the input is scanned and replaced with the production rule from right to left. So in rightmost derivation, we read the input string from right to left.

Example

Production rules:

1. E = E + E
2. E = E - E
3. E = a | b

Input

1. a - b + a

The rightmost derivation is:

1. E = E - E
2. E = E - E + E
3. E = E - E + a
4. E = E - b + a
5. E = a - b + a

When we use the leftmost derivation or rightmost derivation, we may get the same string. This type of derivation does not affect on getting of a string.

Examples of Derivation:

Example 1:

Derive the string "abb" for leftmost derivation and rightmost derivation using a CFG given by,

1. S → AB | ε
2. A → aB
3. B → Sb

Solution:

Leftmost derivation:

S

AB

aB B

a Sb B

A ε bB

ab Sb

ab ε b

abb

Rightmost derivation:

S

AB

A Sb

A ε b

aB b

a Sb b

a ε bb

abb

Example 2:

Derive the string "aabbabba" for leftmost derivation and rightmost derivation using a CFG given by,

1. S → aB | bA
2. S → a | aS | bAA
3. S → b | aS | aBB

Solution:

Leftmost derivation:

1. S
2. aB S → aB
3. aaBB B → aBB
4. aabB B → b
5. aabbS B → bS
6. aabbaB S → aB
7. aabbabS B → bS
8. aabbabbA S → bA
9. aabbabba A → a

Rightmost derivation:

1. S
2. aB S → aB

3. aaBB B → aBB
4. aaBbS B → bS
5. aaBbbA S → bA
6. aaBbba A → a
7. aabSbba B → bS
8. aabbAbba S → bA
9. aabbabba A → a

Example 3:

Derive the string "00101" for leftmost derivation and rightmost derivation using a CFG given by,

1. S → A1B
2. A → 0A | ε
3. B → 0B | 1B | ε

Solution:

Leftmost derivation:

1. S
2. A1B
3. 0A1B
4. 00A1B
5. 001B
6. 0010B
7. 00101B
8. 00101

Rightmost derivation:

1. S
2. A1B
3. A10B
4. A101B
5. A101
6. 0A101
7. 00A101
8. 00101

Derivation Tree

Derivation tree is a graphical representation for the derivation of the given production rules for a given CFG. It is the simple way to show how the derivation can be done to obtain some string from a given set of production rules. The derivation tree is also called a parse tree.

Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

A parse tree contains the following properties:

1. The root node is always a node indicating start symbols.
2. The derivation is read from left to right.
3. The leaf node is always terminal nodes.
4. The interior nodes are always the non-terminal nodes.

Example 1:

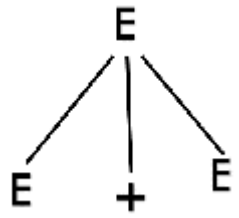
Production rules:

1. E = E + E
2. E = E * E
3. E = a | b | c

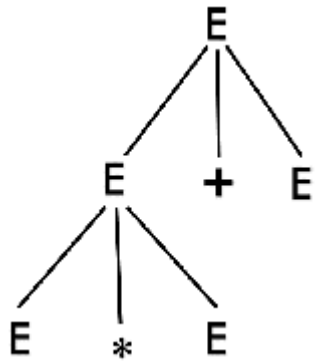
Input

1. a * b + c

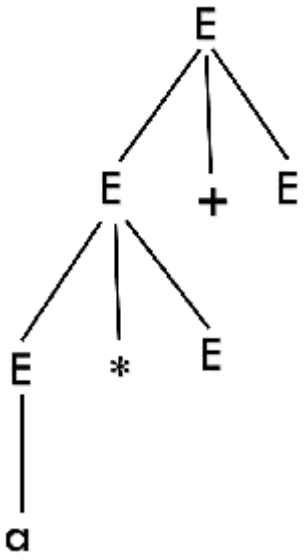
Step 1:



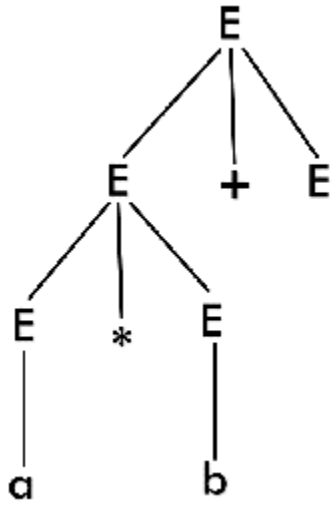
Step 2:



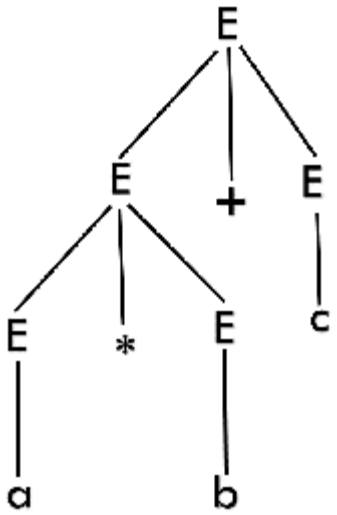
Step 2:



Step 4:



Step 5:



Note: We can draw a derivation tree step by step or directly in one step.

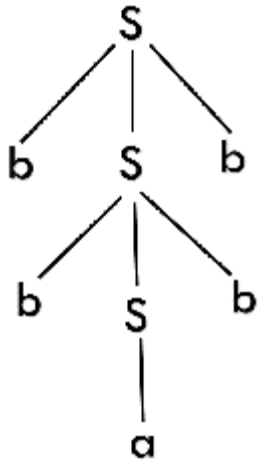
Example 2:

Draw a derivation tree for the string "bab" from the CFG given by

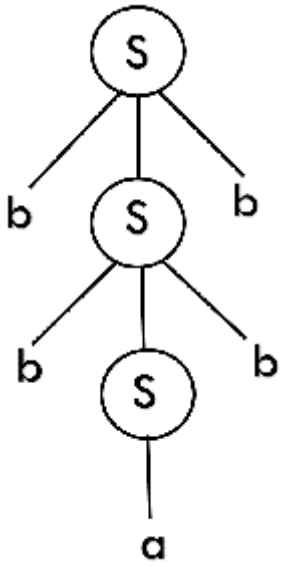
1. S → bSb | a | b

Solution:

Now, the derivation tree for the string "bbabb" is as follows:



The above tree is a derivation tree drawn for deriving a string bbabb. By simply reading the leaf nodes, we can obtain the desired string. The same tree can also be denoted by,



Example 3:

Construct a derivation tree for the string aabbabba for the CFG given by,

1. S → aB | bA
2. A → a | aS | bAA
3. B → b | bS | aBB

Solution:

To draw a tree, we will first try to obtain derivation for the string aabbabba

S

aB

a aBB

aa bS B

aab bA B

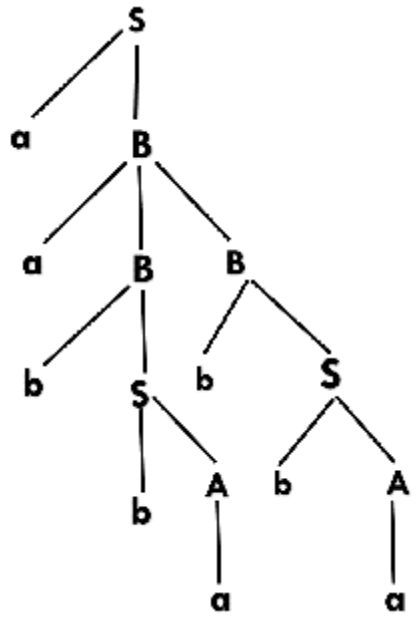
aabb a B

aabba bS

aabbab bA

aabbabb a

Now, the derivation tree is as follows:



Example 4:

Show the derivation tree for string "aabbbb" with the following grammar.

1. S → AB | ε
2. A → aB
3. B → Sb

Solution:

To draw a tree we will first try to obtain derivation for the string aabbbb

S

AB

aB B

a Sb B

a AB bB

a aB BbB

aa Sb BbB

aa ε bBbB

aab Sb bB

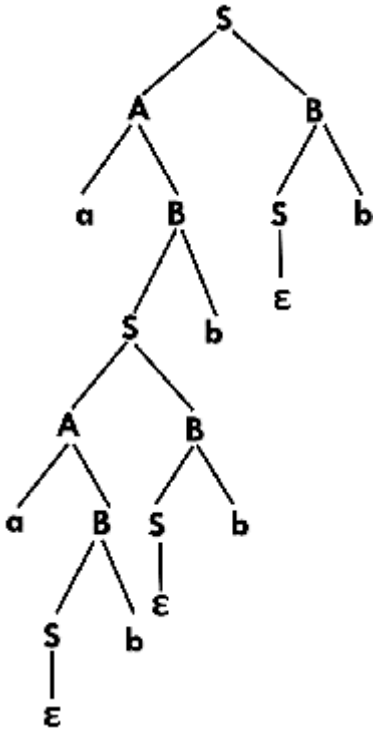
aab ε bbB

aabbb Sb

aabbb ε b

aabbbb

Now, the derivation tree for the string "aabbbb" is as follows:



Ambiguity in Grammar

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string. If the grammar is not ambiguous, then it is called unambiguous.

If the grammar has ambiguity, then it is not good for compiler construction. No method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing the whole grammar without ambiguity.

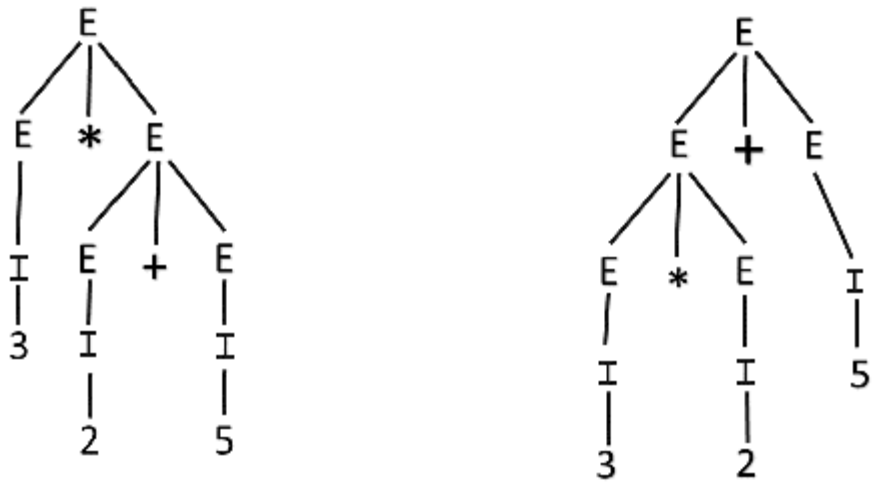
Example 1:

Let us consider a grammar G with the production rule

1. E → I
2. E → E + E
3. E → E * E
4. E → (E)
5. I → ε | 0 | 1 | 2 | ... | 9

Solution:

For the string "3 * 2 + 5", the above grammar can generate two parse trees by leftmost derivation:



Since there are two parse trees for a single string "3 * 2 + 5", the grammar G is ambiguous.

Example 2:

Check whether the given grammar G is ambiguous or not.

1. E → E + E
2. E → E - E
3. E → id

Solution:

From the above grammar String "id + id - id" can be derived in 2 ways:

First Leftmost derivation

1. E → E + E
2. → id + E
3. → id + E - E
4. → id + id - E
5. → id + id- id

Second Leftmost derivation

1. E → E - E
2. → E + E - E
3. → id + E - E
4. → id + id - E
5. → id + id - id

Since there are two leftmost derivation for a single string "id + id - id", the grammar G is ambiguous.

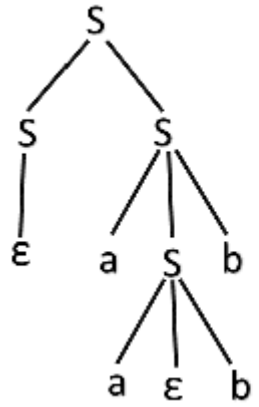
Example 3:

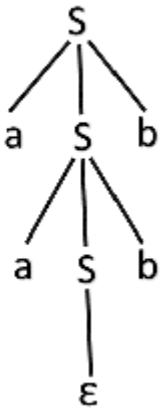
Check whether the given grammar G is ambiguous or not.

1. S → aSb | SS
2. S → ε

Solution:

For the string "aabb" the above grammar can generate two parse trees





Since there are two parse trees for a single string "aabb", the grammar G is ambiguous.

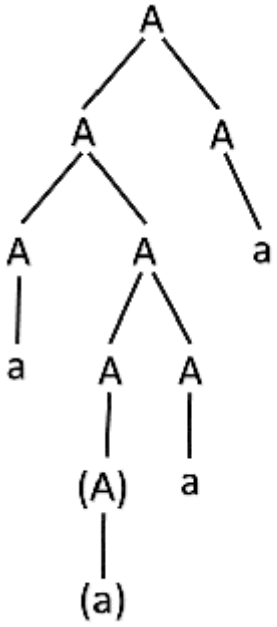
Example 4:

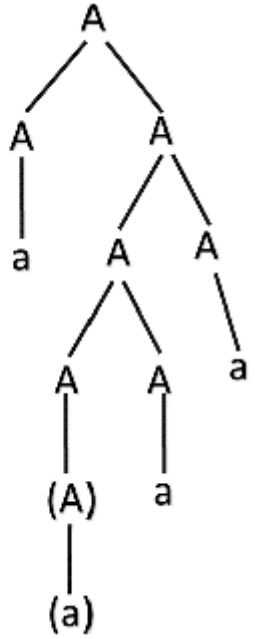
Check whether the given grammar G is ambiguous or not.

1. A → AA
2. A → (A)
3. A → a

Solution:

For the string "a(a)aa" the above grammar can generate two parse trees:





Since there are two parse trees for a single string "a(a)aa", the grammar G is ambiguous.

Unambiguous Grammar

A grammar can be unambiguous if the grammar does not contain ambiguity that means if it does not contain more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string.

To convert ambiguous grammar to unambiguous grammar, we will apply the following rules:

1. If the left associative operators (+, -, *, /) are used in the production rule, then apply left recursion in the production rule. Left recursion means that the leftmost symbol on the right side is the same as the non-terminal on the left side. For example,

1. X → Xa

2. If the right associative operates(^) is used in the production rule then apply right recursion in the production rule. Right recursion means that the rightmost symbol on the left side is the same as the non-terminal on the right side. For example,

1. X → aX

Example 1:

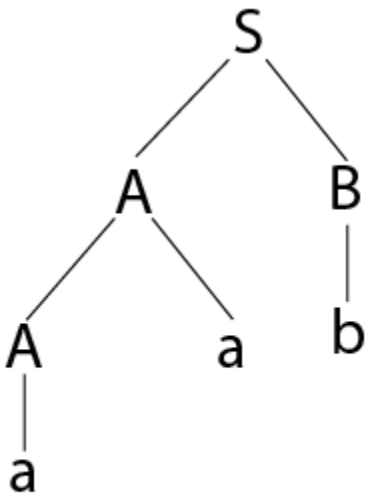
Consider a grammar G is given as follows:

1. S → AB | aaB
2. A → a | Aa
3. B → b

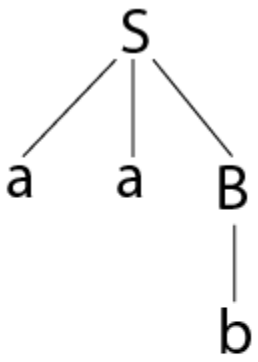
Determine whether the grammar G is ambiguous or not. If G is ambiguous, construct an unambiguous grammar equivalent to G.

Solution:

Let us derive the string "aab"



Parse tree 1



Parse tree 2

As there are two different parse tree for deriving the same string, the given grammar is ambiguous.

Unambiguous grammar will be:

1. S → AB
2. A → Aa | a
3. B → b

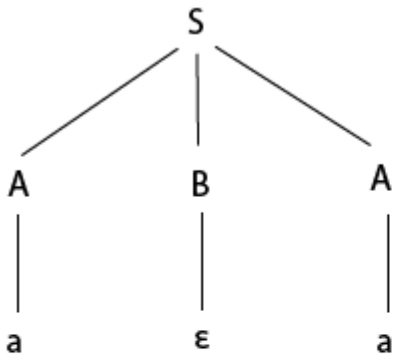
Example 2:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

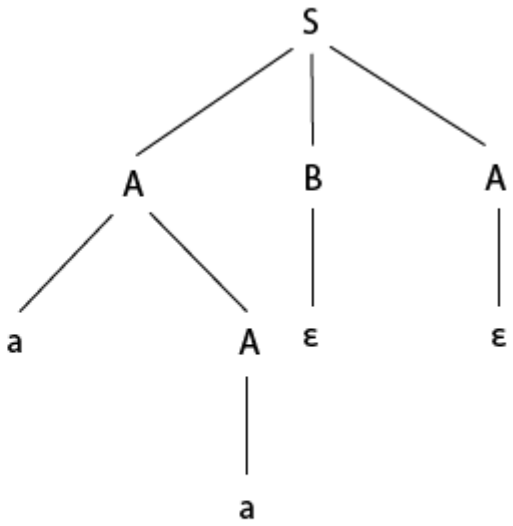
1. S → ABA
2. A → aA | ε
3. B → bB | ε

Solution:

The given grammar is ambiguous because we can derive two different parse tree for string aa.



Parse tree 1



Parse tree 2

The unambiguous grammar is:

1. S → aXY | bYZ | ε
2. Z → aZ | a
3. X → aXY | a | ε
4. Y → bYZ | b | ε

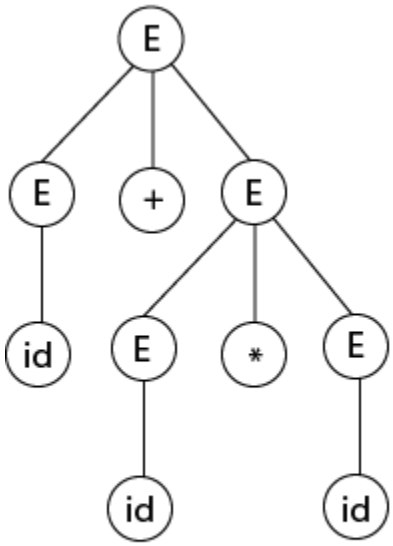
Example 3:

Show that the given grammar is ambiguous. Also, find an equivalent unambiguous grammar.

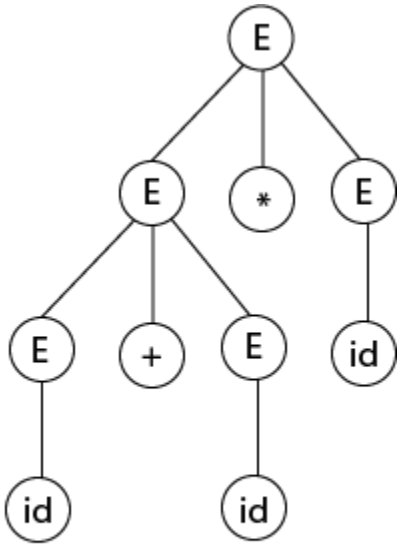
1. E → E + E
2. E → E * E
3. E → id

Solution:

Let us derive the string "id + id * id"



Parse tree 1



Parse tree 2

As there are two different parse tree for deriving the same string, the given grammar is ambiguous.

Unambiguous grammar will be:

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → id

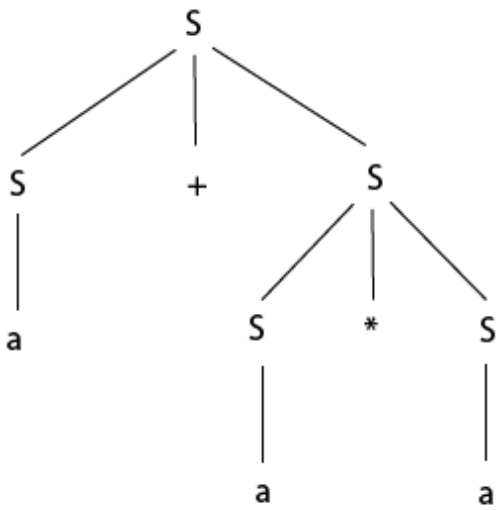
Example 4:

Check that the given grammar is ambiguous or not. Also, find an equivalent unambiguous grammar.

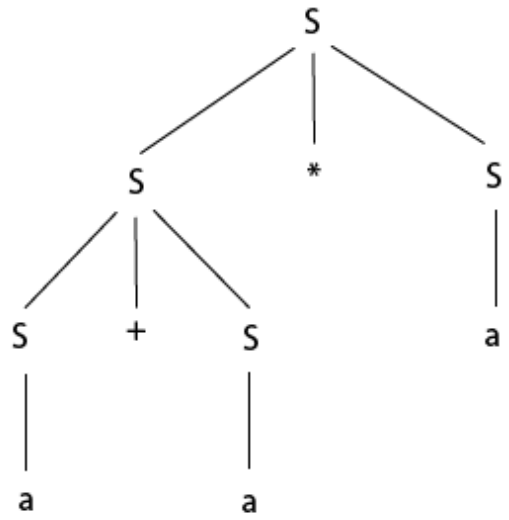
1. S → S + S
2. S → S * S
3. S → S ^ S
4. S → a

Solution:

The given grammar is ambiguous because the derivation of string aab can be represented by the following string:



Parse tree 1



Parse tree 2

Unambiguous grammar will be:

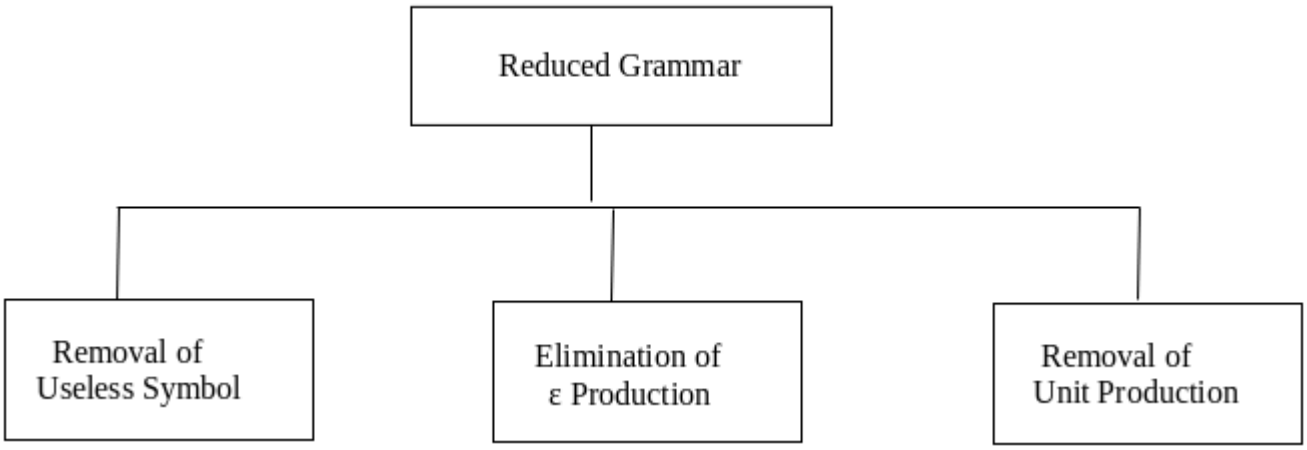
1. S → S + A |
2. A → A * B | B
3. B → C ^ B | C
4. C → a

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 → 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 → ε.

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 → aaB | abA | aaT
2. A → aA
3. B → ab | b
4. C → ad

In the above example, the variable 'C' will never occur in the derivation of any string, so the production C → 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 → 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 → 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 → ε 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 → a, construct all production A → 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 → XYX
2. X → 0X | ε
3. Y → 1Y | ε

Solution:

Now, while removing ε production, we are deleting the rule X → ε and Y → ε. 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 → XYX

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

1. S → YX

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

1. S → XY

If Y = ε then

1. S → XX

If Y and X are ε then,

1. S → X

   If both X are replaced by ε

1. S → Y

   Now,

1. S → XY | YX | XX | X | Y

   Now let us consider

1. X → 0X

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

1. X → 0
2. X → 0X | 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:

S → 0A | 1B | C

A → 0S | 00

B → 1 | A

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.

S → 0A | 1B | 01

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

B → 1 | 0S | 00

Thus finally we can write CFG without unit production as

S → 0A | 1B | 01
A → 0S | 00
B → 1 | 0S | 00
C → 01

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 → RB
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:

1. S0 → a | aA | Aa | b
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Step 3: In the production rule S0 → aA | Aa, S → aA | Aa, A → aBB and B → Aa, terminal a exists on RHS with non-terminals. So we will replace terminal a with X:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → XBB
4. B → AX | b | a
5. X → a

Step 4: In the production rule A → XBB, RHS has more than two symbols, removing it from grammar yield:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → RB
4. B → AX | b | a
5. X → a
6. R → XB

Hence, for the given grammar, this is the required CNF.

Greibach Normal Form (GNF)

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

- A start symbol generating ε. For example, S → ε.
- A non-terminal generating a terminal. For example, A → a.
- A non-terminal generating a terminal which is followed by any number of non-terminals. For example, S → aASB.

For example:

1. G1 = {S → aAB | aB, A → aA| a, B → bB | b}
2. G2 = {S → aAB | aB, A → aA | ε, B → bB | ε}

The production rules of Grammar G1 satisfy the rules specified for GNF, so the grammar G1 is in GNF. However, the production rule of Grammar G2 does not satisfy the rules specified for GNF as A → ε and B → ε contains ε(only start symbol can generate ε). So the grammar G2 is not in GNF.

Steps for converting CFG into GNF

Step 1: Convert the grammar into CNF.

If the given grammar is not in CNF, convert it into CNF. You can refer the following topic to convert the CFG into CNF: Chomsky normal form

Step 2: If the grammar exists left recursion, eliminate it.

If the context free grammar contains left recursion, eliminate it. You can refer the following topic to eliminate left recursion: Left Recursion

Step 3: In the grammar, convert the given production rule into GNF form.

If any production rule in the grammar is not in GNF form, convert it.

Example:

1. S → XB | AA

2. A → a | SA
3. B → b
4. X → a

Solution:

As the given grammar G is already in CNF and there is no left recursion, so we can skip step 1 and step 2 and directly go to step 3.

The production rule A → SA is not in GNF, so we substitute S → XB | AA in the production rule A → SA as:

1. S → XB | AA
2. A → a | XBA | AAA
3. B → b
4. X → a

The production rule S → XB and A → XBA is not in GNF, so we substitute X → a in the production rule S → XB and A → XBA as:

1. S → aB | AA
2. A → a | aBA | AAA
3. B → b
4. X → a

Now we will remove left recursion (A → AAA), we get:

1. S → aB | AA
2. A → aC | aBAC
3. C → AAC | ε
4. B → b
5. X → a

Now we will remove null production C → ε, we get:

1. S → aB | AA
2. A → aC | aBAC | a | aBA
3. C → AAC | AA
4. B → b
5. X → a

The production rule S → AA is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule S → AA as:

1. S → aB | aCA | aBACA | aA | aBAA
2. A → aC | aBAC | a | aBA
3. C → AAC
4. C → aCA | aBACA | aA | aBAA
5. B → b
6. X → a

The production rule C → AAC is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule C → AAC as:

1. S → aB | aCA | aBACA | aA | aBAA
2. A → aC | aBAC | a | aBA
3. C → aCAC | aBACAC | aAC | aBAAC
4. C → aCA | aBACA | aA | aBAA
5. B → b
6. X → a

Hence, this is the GNF form for the grammar G.