**\*SYNTAX ANALYSIS**

**Introduction to Syntax Analysis**

* Syntax Analysis or Parsing is the **second phase**, i.e. after lexical analysis.
* It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not.
* It does so by building a data structure, called a **Parse tree or Syntax tree**.
* The parse tree is constructed by using the pre-defined Grammar of the language and the input string.
* If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. if not, error is reported by syntax analyzer.

We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, **this phase uses context-free grammar (CFG),** which is recognized by push-down automata.

The Grammar for a Language consists of **Production rules.**

**Example:**

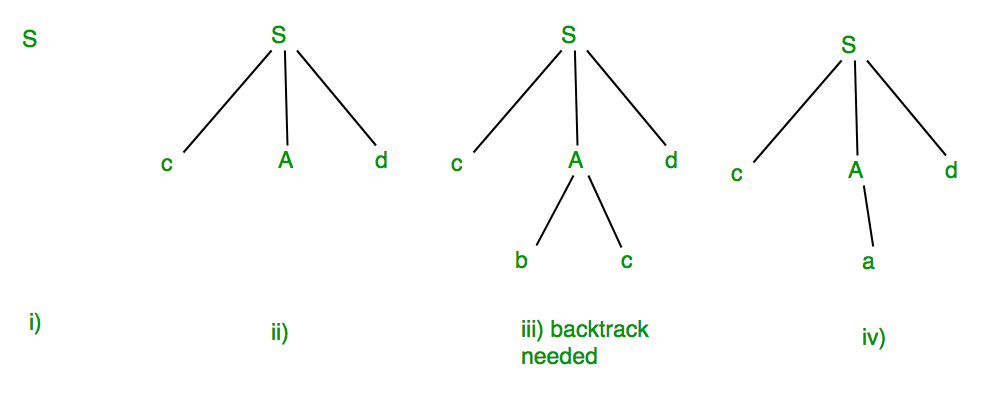
Suppose Production rules for the Grammar of a language are:

**S -> cAd**

**A -> bc | a**

And the input string is **“cad”. And “cbcd”**

Now the parser attempts to construct syntax tree from this grammar for the given input string. It uses the given production rules and applies those as needed to generate the string. To generate string “cad” it uses the rules as shown in the given diagram:



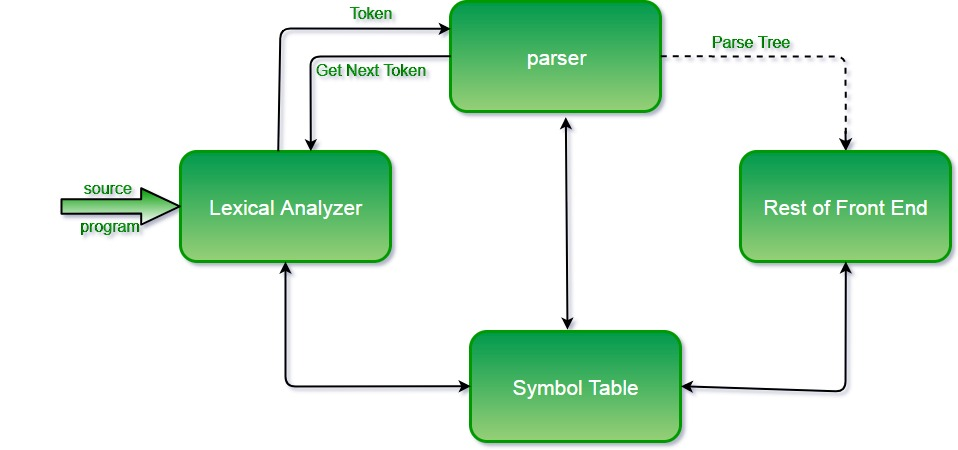
In the step iii above, the production rule A->bc was not a suitable one to apply (because the string produced is “cbcd” not “cad”), here the parser needs to backtrack, and apply the next production rule available with A which is shown in the step iv, and the string “cad” is produced.

Thus, the given input can be produced by the given grammar, therefore the input is correct in syntax. But backtrack was needed to get the correct syntax tree, which is really a complex process to implement.

There can be an easier way to solve this, which we shall see in the next topic “Concepts of FIRST and FOLLOW sets in Compiler Design”.

## **\*Role of the parser :**

In the syntax analysis phase, a compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language. This is done by a parser. The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be the grammar for the source language. It **detects and reports any syntax errors** and **produces a parse tree** from which intermediate code can be generated.



Before going to types of parsers we will discuss on some ideas about some important things required for understanding parsing.

# Error Handling in Compiler Design

The tasks of the **Error Handling** process are to detect each error, report it to the user, and then make some recover strategy and implement them to handle error. During this whole process processing time of program should not be slow. An **Error** is the blank entries in the symbol table.

**Types or Sources of Error –** There are **two types of error**: run-time and compile-time error:

1. A **run-time error** is an error which takes place during the execution of a program, and usually happens because of adverse system parameters or invalid input data. The lack of sufficient memory to run an application or a memory conflict with another program and logical error are example of this. Logic errors, occur when executed code does not produce the expected result. Logic errors are best handled by meticulous program debugging.
2. **Compile-time errors** rises at compile time, before execution of the program. Syntax error or missing file reference that prevents the program from successfully compiling is the example of this.

**Classification of Compile-time error –**

1. **Lexical:** This includes misspellings of identifiers, keywords or operators
2. **Syntactical**: missing semicolon or unbalanced parenthesis
3. **Semantical**: incompatible value assignment or type mismatches between operator and operand
4. **Logical**: code not reachable, infinite loop.

**Finding error or reporting an error –** Viable-prefix is the property of a parser which allows early detection of syntax errors.

* **Goal:** detection of an error as soon as possible without further consuming unnecessary input
* **How:** detect an error as soon as the prefix of the input does not match a prefix of any string in the language.
* **Example:** for (**;**), this will report an error as for have two semicolons inside braces.

**Error Recovery –**

The basic requirement for the compiler is to simply stop and issue a message, and cease compilation. There are some common recovery methods that are follows.

1. **Panic mode recovery:** This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops while recovering error. The parser discards the input symbol one at a time until one of the designated (like end, semicolon) set of synchronizing tokens (are typically the statement or expression terminators) is found. This is adequate when the presence of multiple errors in same statement is rare.

Example: Consider the erroneous expression (1 + + 2) + 3. Panic-mode recovery: Skip ahead to next integer and then continue. Bison: use the special terminal**error** to describe how much input to skip.

**E->int|E+E|(E)|error int|(error)**

1. **Phase level recovery:** Perform local correction on the input to repair the error. But error correction is difficult in this strategy. Ex: **replace a comma by semicolon.**
2. **Error productions:** Some common errors are known to the compiler designers that may occur in the code. Augmented grammars can also be used, as productions that generate erroneous constructs when these errors are encountered. Example: write **5x instead of 5\*x**
3. **Global correction:** Its aim is to make as few changes as possible while converting an incorrect input string to a valid string. This strategy is costly to implement.

To find errors first we have to known how to give input. So, specification of input can be done by CFG.

## **\*Context-Free Grammar**

In this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.

A context-free grammar has **four components:**

* A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.(CAPS)
* A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.(SMALL CASE)
* A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.
* One of the non-terminals is designated as the **start symbol (S)**; from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

### **Example**

We take the problem of palindrome language, which cannot be described by means of Regular Expression. That is, L = { w | w = wR } is not a regular language. But it can be described by means of CFG, as illustrated below:

G = ( V, Σ, P, S )

Where:

V = { Q, Z, N }

Σ = { 0, 1 }

P = { Q → Z | Q → N | Q → ℇ | Z → 0Q0 | N → 1Q1 }

S = { Q }

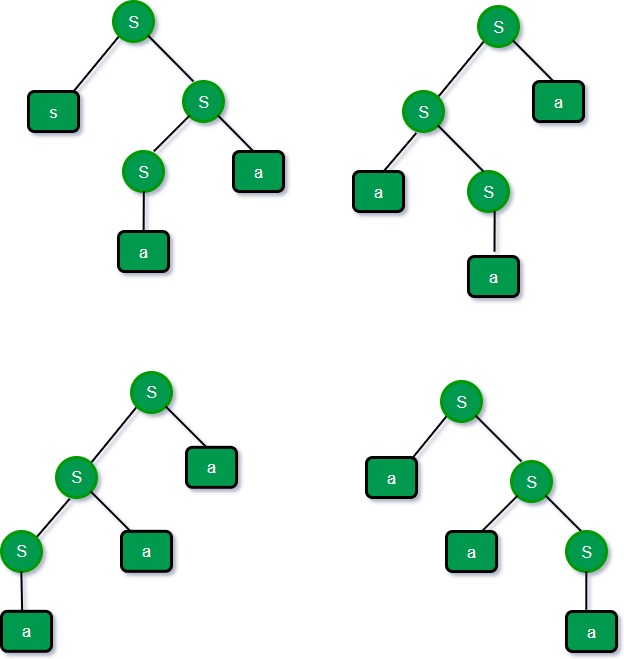
This grammar describes palindrome language, such as: 1001, 11100111, 00100, 1010101, 11111, etc.

**Ambiguity**  
A grammar that produces more than one parse tree for some sentence is said to be ambiguous.  
Eg- consider a grammar

**S -> aS | Sa | a**

Now for string “**aaa”** we will have **4 parse trees**, hence ambiguous

NOTE: small mistake in first parse tree replace **s** with **a**.



**Definition**− A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P, S)** where

* **N** is a set of non-terminal symbols.
* **T** is a set of terminals where **N ∩ T = NULL.**
* **P** is a set of rules, **P: N → (N ∪ T) \***, i.e., the left-hand side of the production rule **P** does have any right context or left context.
* **S** is the start symbol.

**Example**

* The grammar ({A}, {a, b, c}, P, A), **P: A → aA, A → abc.**
* The grammar ({S, a, b}, {a, b}, P, S), **P: S → aSa, S → bSb, S → ε**
* The grammar ({S, F}, {0, 1}, P, S), **P: S → 00S | 11F, F → 00F | ε**

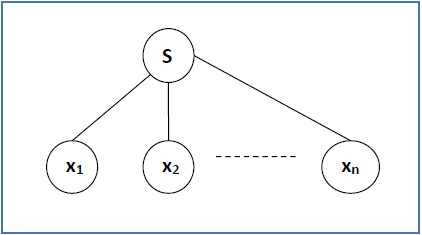
## **Generation of Derivation Tree**

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

### **Representation Technique**

* **Root vertex** − Must be labeled by the start symbol.
* **Vertex** − Labeled by a non-terminal symbol.
* **Leaves** − Labeled by a terminal symbol or ε.

If S → x1x2 …… xn is a production rule in a CFG, then the parse tree / derivation tree will be as follows –



There are **two different approaches** to draw a derivation tree −

* 1. **Top-down Approach −**
* Starts with the starting symbol **S**
* Goes down to tree leaves using productions
  1. **Bottom-up Approach −**
* Starts from tree leaves
* Proceeds upward to the root which is the starting symbol **S**

### **Derivation or Yield of a Tree**

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

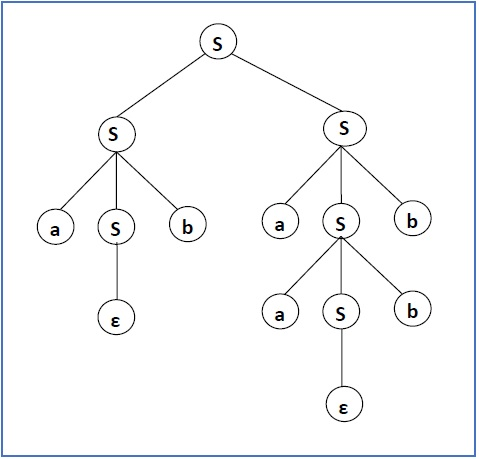
**Example**

Let a CFG {N,T,P,S} be

N = {S}, T = {a, b}, Starting symbol = S, **P = S → SS | aSb | ε**

One derivation from the above CFG is “abaabb”

S → SS → aSbS → abS → abaSb → abaaSbb → abaabb



### **Sentential Form and Partial Derivation Tree**

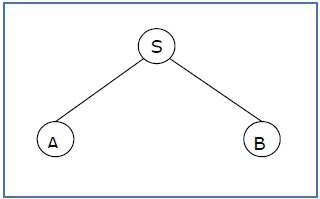
A partial derivation tree is a sub-tree of a derivation tree/parse tree such that either all of its children are in the sub-tree or none of them are in the sub-tree.

**Example**

If in any CFG the productions are −

S → AB, A → aaA | ε, B → Bb| ε

the partial derivation tree can be the following −



If a partial derivation tree contains the root S, it is called a **sentential form**. The above sub-tree is also in sentential form.

### **Leftmost and Rightmost Derivation of a String**

* **Leftmost derivation** − A leftmost derivation is obtained by applying production to the leftmost variable in each step.
* **Rightmost derivation** − A rightmost derivation is obtained by applying production to the rightmost variable in each step.

**Example**

Let any set of production rules in a CFG be

X → X+X | X\*X |X| a

over an alphabet {a}.

The leftmost derivation for the string **"a+a\*a"** may be −

**X → X+X**

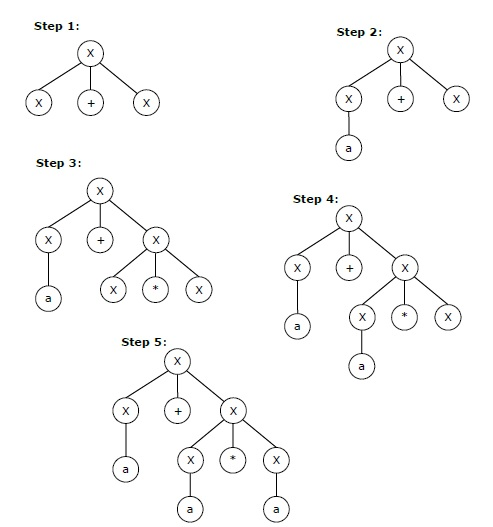
**→ a+X**

**→ a + X\*X**

**→ a+a\*X**

**→ a+a\*a**

The stepwise derivation of the above string is shown as below −



**The rightmost derivation** for the above string **"a+a\*a"** may be

**X → X\*X**

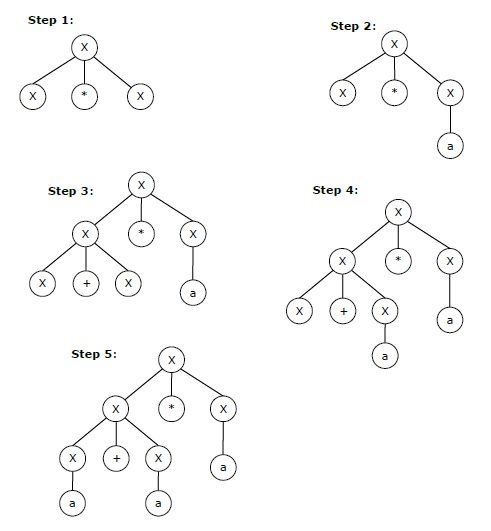
**→ X\*a**

**→ X+X\*a**

**→ X+a\*a**

**→ a+a\*a**

The stepwise derivation of the above string is shown as below –



## **Left and Right Recursive Grammars**

In a context-free grammar **G**, if there is a production in the form **X → Xa** where **X** is a non-terminal and **‘a’** is a string of terminals, it is called a **left recursive production**. The grammar having a left recursive production is called a **left recursive grammar**.

And if in a context-free grammar **G**, if there is a production is in the form **X → aX** where **X** is a non-terminal and **‘a’** is a string of terminals, it is called a **right recursive production**. The grammar having a right recursive production is called a **right recursive grammar**.

# \*Ambiguity in Context-Free Grammars

**DEF**: “If a context free grammar **G** has more than one derivation tree for some string **w ∈ L(G)**, it is called an **ambiguous grammar**”.

There exist multiple right-most or left-most derivations for some string generated from that grammar.

## **Problem**

Check whether the grammar G with production rules −

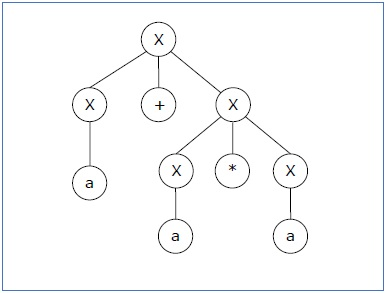
X → X+X | X\*X |X| a is ambiguous or not.

## **Solution**

Let’s find out the derivation tree for the string "a+a\*a". It has two leftmost derivations.

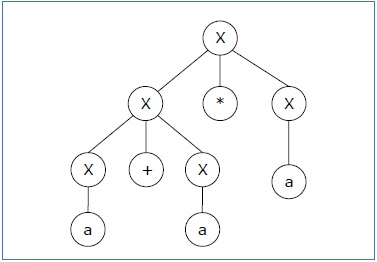
**Derivation 1** − X → X+X → a +X → a+ X\*X → a+a\*X → a+a\*a

**Parse tree 1** −



**Derivation 2** − X → X\*X → X+X\*X → a+ X\*X → a+a\*X → a+a\*a

**Parse tree 2** −



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

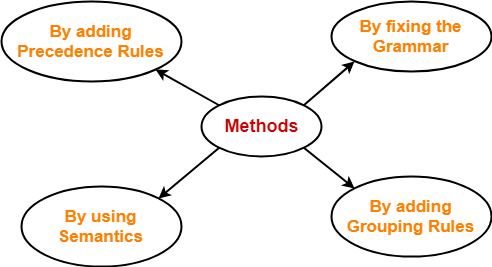
## **\*Converting Ambiguous Grammar Into Unambiguous Grammar**

* 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. |

## **Methods To Remove Ambiguity-**

The ambiguity from the grammar may be removed using the following methods-



* By fixing the grammar
* By adding grouping rules
* By using semantics and choosing the parse that makes the most sense
* By adding the precedence rules or other context sensitive parsing rules

## **Removing Ambiguity By Precedence & Associativity Rules-**

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

* **Precedence Constraints a+b\*c**
* **Associativity Constraints a+b+c**

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 → R + R / R . R / R\* / a / b**

where \* is kleen closure 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 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-

**R → R + R / R . R / R\* / a / b**

**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 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**

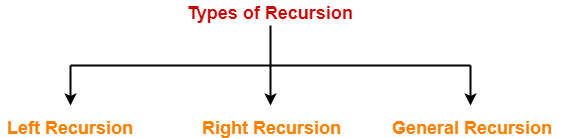
**N → not N / T / F**

**Unambiguous Grammar**

# [Left Recursion | Left Recursion Elimination](https://www.gatevidyalay.com/left-recursion-left-recursion-elimination/)

## **Recursion-**

Recursion can be classified into following three types-



1. Left Recursion
2. Right Recursion
3. General Recursion

## **1. Left Recursion-**

* A production of grammar is said to have **left recursion** if the leftmost variable of its RHS is same as variable of its LHS.
* A grammar containing a production having left recursion is called as Left Recursive Grammar.

### **Example-**

S → Sa / ∈

(**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**→**Aα / β**  (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**→ **βA’**  **A’**→ **αA’ / ∈**  (Right Recursive Grammar)    This right recursive grammar functions same as left recursive grammar. |

## **2. Right Recursion-**

* A production of grammar is said to have **right recursion** if the rightmost variable of its RHS is same as variable of its LHS.
* A grammar containing a production having right recursion is called as Right Recursive Grammar.

### **Example-**

S → aS / ∈

(**Right Recursive Grammar**)

* Right recursion does not create any problem for the Top down parsers.
* Therefore, there is no need of eliminating right recursion from the grammar.

## **3. General Recursion-**

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

### **Example-**

S → aSb / ∈

## **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 / ∈**

## **Problem-03:**

Consider the following grammar and eliminate left recursion-

**E → E + T / T**

**T → T x F / F**

**F → id**

## **Solution-**

The grammar after eliminating left recursion is-

**E → TE’**

**E’ → +TE’ / ∈**

**T → FT’**

**T’ → xFT’ / ∈**

**F → id**

## **Problem-04:**

Consider the following grammar and eliminate left recursion-

S → (L) / a

L → L , S / S

## **Solution-**

The grammar after eliminating left recursion is-

S → (L) / a

L → SL’

L’ → ,SL’ / ∈

## **Problem-05:**

Consider the following grammar and eliminate left recursion-

S → S0S1S / 01

## **Solution-**

The grammar after eliminating left recursion is-

S → 01A

A → 0S1SA / ∈

## **Problem-06:**

Consider the following grammar and eliminate left recursion-

S → A

A → Ad / Ae / aB / ac

B → bBc / f

## **Solution-**

The grammar after eliminating left recursion is-

S → A

A → aBA’ / acA’

A’ → dA’ / eA’ / ∈

B → bBc / f

## **Problem-07:**

Consider the following grammar and eliminate left recursion-

A → AAα / β

## **Solution-**

The grammar after eliminating left recursion is-

A → βA’

A’ → AαA’ / ∈

## **Problem-08:**

Consider the following grammar and eliminate left recursion-

A → Ba / Aa / c

B → Bb / Ab / d

## **Solution-**

This is a case of indirect left recursion.

### **Step-01:**

First let us eliminate left recursion from A → Ba / Aa / c

Eliminating left recursion from here, we get-

A → BaA’ / cA’

A’ → aA’ / ∈

Now, given grammar becomes-

A → BaA’ / cA’

A’ → aA’ / ∈

B → Bb / Ab / d

### **Step-02:**

Substituting the productions of A in B → Ab, we get the following grammar-

A → BaA’ / cA’

A’ → aA’ / ∈

B → Bb / BaA’b / cA’b / d

### **Step-03:**

Now, eliminating left recursion from the productions of B, we get the following grammar-

A → BaA’ / cA’

A’ → aA’ / ∈

B → cA’bB’ / dB’

B’ → bB’ / aA’bB’ / ∈

This is the final grammar after eliminating left recursion.

## **Problem-09:**

Consider the following grammar and eliminate left recursion-

X → XSb / Sa / b

S → Sb / Xa / a

## **Solution-**

This is a case of indirect left recursion.

### **Step-01:**

First let us eliminate left recursion from X → XSb / Sa / b

Eliminating left recursion from here, we get-

X → SaX’ / bX’

X’ → SbX’ / ∈

Now, given grammar becomes-

X → SaX’ / bX’

X’ → SbX’ / ∈

S → Sb / Xa / a

### **Step-02:**

Substituting the productions of X in S → Xa, we get the following grammar-

X → SaX’ / bX’

X’ → SbX’ / ∈

S → Sb / SaX’a / bX’a / a

### **Step-03:**

Now, eliminating left recursion from the productions of S, we get the following grammar-

X → SaX’ / bX’

X’ → SbX’ / ∈

S → bX’aS’ / aS’

S’ → bS’ / aX’aS’ / ∈

This is the final grammar after eliminating left recursion.

## **Problem-10:**

Consider the following grammar and eliminate left recursion-

S → Aa / b

A → Ac / Sd / ∈

## **Solution-**

This is a case of indirect left recursion.

### **Step-01:**

First let us eliminate left recursion from S → Aa / b

This is already free from left recursion.

### **Step-02:**

Substituting the productions of S in A → Sd, we get the following grammar-

S → Aa / b

A → Ac / Aad / bd / ∈

### **Step-03:**

Now, eliminating left recursion from the productions of A, we get the following grammar-

S → Aa / b

A → bdA’ / A’

A’ → cA’ / adA’ / ∈

This is the final grammar after eliminating left recursion.

# \*[Left Factoring](https://www.gatevidyalay.com/left-factoring-examples-compiler-design/)

## **Grammar With Common Prefixes-**

|  |
| --- |
| If RHS of more than one production starts with the same symbol, then such a grammar is called as **Grammar with Common Prefixes**. |

### **Example-**

**A**→**αβ1 / αβ2 / αβ3**

* This kind of grammar creates a problematic situation for Top down parsers.
* Top down parsers cannot decide which production must be chosen to parse the string in hand.
* To remove this confusion, we use left factoring.

## **Left Factoring-**

|  |
| --- |
| **Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.** |

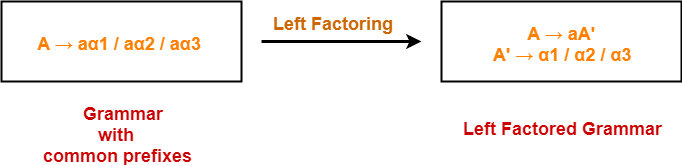
**How?**

In left factoring,

* We make one production for each common prefixes.
* The common prefix may be a terminal or a non-terminal or a combination of both.
* Rest of the derivation is added by new productions.

The grammar obtained after the process of left factoring is called as **Left Factored Grammar**.

**Example-**



## **Problem-01:**

Do left factoring in the following grammar-

**S → iEtS / iEtSeS / a**

**E → b**

## **Solution-**

The left factored grammar is-

**S → iEtSS’ / a**

**S’ → eS / ∈**

**E → b**

## **Problem-02:**

Do left factoring in the following grammar-

**A → aAB / aBc / aAc**

## **Solution-**

### **Step-01:**

**A → aA’**

**A’ → AB / Bc / Ac**

Again, this is a grammar with common prefixes.

### **Step-02:**  **A → aA’**

**A’ → AD / Bc**

**D → B / c**

This is a left factored grammar.

## **Problem-03:**

Do left factoring in the following grammar-

**S → bSSaaS / bSSaSb / bSb / a**

## **Solution-**

### **Step-01:**

**S → bSS’ / a**

**S’ → SaaS / SaSb / b**

Again, this is a grammar with common prefixes.

### **Step-02: S → bSS’ / a**

**S’ → SaA / b**

**A → aS / Sb**

This is a left factored grammar.

## **Problem-04:**

Do left factoring in the following grammar-

**S → aSSbS / aSaSb / abb / b**

## **Solution-**

### **Step-01:**

**S → aS’ / b**

**S’ → SSbS / SaSb / bb**

Again, this is a grammar with common prefixes.

### **Step-02:**

**S → aS’ / b**

**S’ → SA / bb**

**A → SbS / aSb**

This is a left factored grammar.

## **Problem-05:**

Do left factoring in the following grammar-

**S → a / ab / abc / abcd**

## **Solution-**

**Step-01:**

**S → aS’**

**S’ → b / bc / bcd / ∈**

Again, this is a grammar with common prefixes.

**Step-02:**

**S → aS’**

**S’ → bA / ∈**

**A → c / cd / ∈**

Again, this is a grammar with common prefixes.

### **Step-03:**  **S → aS’**

**S’ → bA / ∈**

**A → cB / ∈**

**B → d / ∈**

This is a left factored grammar.

## **Problem-06:**

Do left factoring in the following grammar-

S → aAd / aB

A → a / ab

B → ccd / ddc

## **Solution-**

The left factored grammar is-

**S → aS’**

**S’ → Ad / B**

**A → aA’**

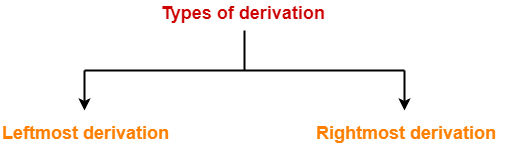
**A’ → b / ∈**

**B → ccd / ddc**

# \*[Parse Tree](https://www.gatevidyalay.com/parse-tree-derivations-automata/)

## **Parse Tree-**

* The process of deriving a string is called as **derivation**.
* The geometrical representation of a derivation is called as a **parse tree** or **derivation tree.**



## **1. Leftmost Derivation-**

* The process of deriving a string by expanding the leftmost non-terminal at each step is called as **leftmost derivation**.
* The geometrical representation of leftmost derivation is called as a **leftmost derivation tree**.

### **Example-**

Consider the following grammar-

B → bS / aBB / b

(**Unambiguous Grammar**)

Let us consider a string w = aaabbabbba

Now, let us derive the string w using leftmost derivation.

### **Leftmost Derivation-**

S   → a**B**

→  aa**B**B                   (Using B → aBB)

→ aaa**B**BB                (Using B → aBB)

→ aaab**B**B                (Using B → b)

→ aaabb**B**                (Using B → b)

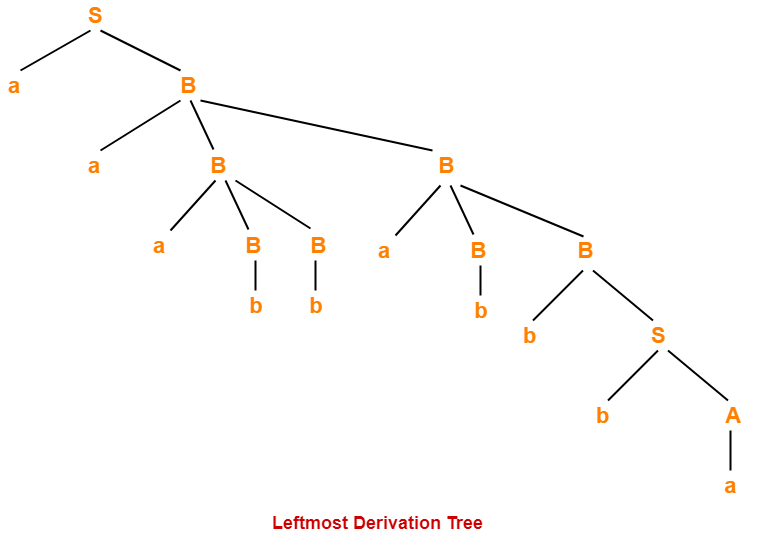
→ aaabba**B**B            (Using B → aBB)

→ aaabbab**B**            (Using B → b)

→ aaabbabb**S**          (Using B → bS)

→ aaabbabbb**A**        (Using S → bA)

→ aaabbabbba         (Using A → a)



## **2. Rightmost Derivation-**

* The process of deriving a string by expanding the rightmost non-terminal at each step is called as **rightmost derivation**.
* The geometrical representation of rightmost derivation is called as a **rightmost derivation tree**.

### **Example-**

Consider the following grammar-

S → aB / bA

S → aS / bAA / a

B → bS / aBB / b

(**Unambiguous Grammar**)

Let us consider a string w = aaabbabbba

Now, let us derive the string w using rightmost derivation.

### **Rightmost Derivation-**

S   → a**B**

→  aaB**B**                    (Using B → aBB)

→ aaBaB**B**                 (Using B → aBB)

→ aaBaBb**S**               (Using B → bS)

→ aaBaBbb**A**             (Using S → bA)

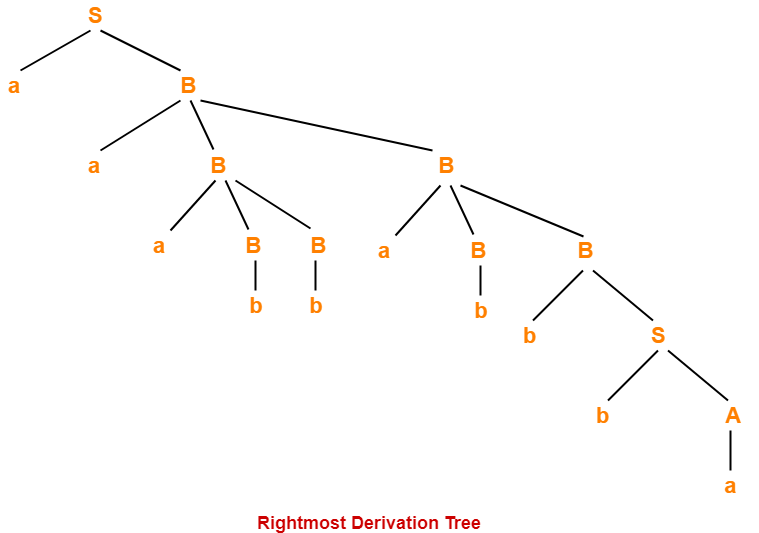
→ aaBa**B**bba              (Using A → a)

→ aa**B**abbba              (Using B → b)

→ aaaB**B**abbba          (Using B → aBB)

→ aaa**B**babbba          (Using B → b)

→ aaabbabbba           (Using B → b)



|  |
| --- |
| **NOTES**  * For unambiguous grammars, Leftmost derivation and Rightmost derivation represents the same parse tree. * For ambiguous grammars, Leftmost derivation and Rightmost derivation represents different parse trees. |

Here,

* The given grammar was unambiguous.
* That is why, leftmost derivation and rightmost derivation represents the same parse tree.

|  |
| --- |
| **Leftmost Derivation Tree = Rightmost Derivation Tree** |

## **Properties Of Parse Tree-**

* Root node of a parse tree is the start symbol of the grammar.
* Each leaf node of a parse tree represents a terminal symbol.
* Each interior node of a parse tree represents a non-terminal symbol.
* Parse tree is independent of the order in which the productions are used during derivations.

## **Yield Of Parse Tree-**

* Concatenating the leaves of a parse tree from the left produces a string of terminals.
* This string of terminals is called as **yield of a parse tree**.

## **PRACTICE PROBLEMS BASED ON DERIVATIONS AND PARSE TREE-**

## **Problem-01:**

Consider the grammar-

S → bB / aA

A → b / bS / aAA

B → a / aS / bBB

For the string w = bbaababa, find-

1. Leftmost derivation
2. Rightmost derivation
3. Parse Tree

## **Solution-**

### **1. Leftmost Derivation-**

S   → b**B**

→ bb**B**B              (Using B → bBB)

→ bba**B**              (Using B → a)

→ bbaa**S**            (Using B → aS)

→ bbaab**B**          (Using S → bB)

→ bbaaba**S**        (Using B → aS)

→ bbaabab**B**      (Using S → bB)

→ bbaababa       (Using B → a)

### **2. Rightmost Derivation-**

S   → b**B**

→ bbB**B**                (Using B → bBB)

→ bbBa**S**              (Using B → aS)

→ bbBab**B**            (Using S → bB)

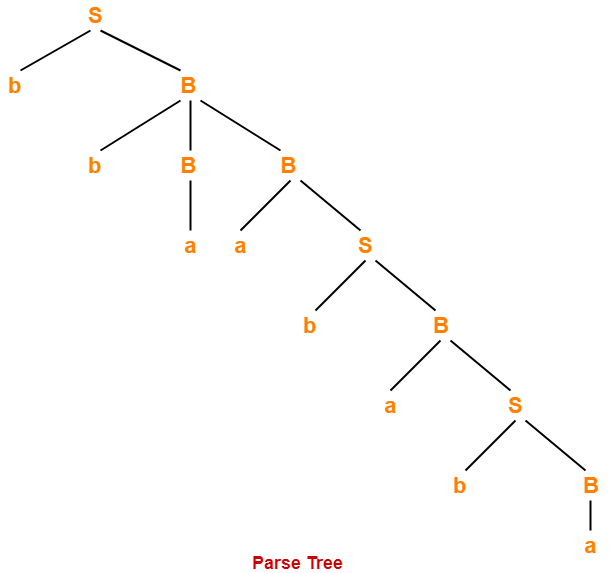
→ bbBaba**S**          (Using B → aS)

→ bbBabab**B**        (Using S → bB)

→ bb**B**ababa        (Using B → a)

→ bbaababa         (Using B → a)

### **3. Parse Tree-**



* Whether we consider the leftmost derivation or rightmost derivation, we get the above parse tree.
* The reason is given grammar is unambiguous.

## **Problem-02:**

Consider the grammar-

S → A1B

A → 0A / ∈

B → 0B / 1B / ∈

For the string w = 00101, find-

1. Leftmost derivation
2. Rightmost derivation
3. Parse Tree

## **Solution-**

### **1. Leftmost Derivation-**

S   → **A**1B

→ 0**A**1B              (Using A → 0A)

→ 00**A**1B            (Using A → 0A)

→ 001**B**              (Using A → ∈)

→ 0010**B**            (Using B → 0B)

→ 00101**B**          (Using B → 1B)

→ 00101             (Using B → ∈)

### **2. Rightmost Derivation-**

S   → A1**B**

→ A10**B**                (Using B → 0B)

→ A101**B**              (Using B → 1B)

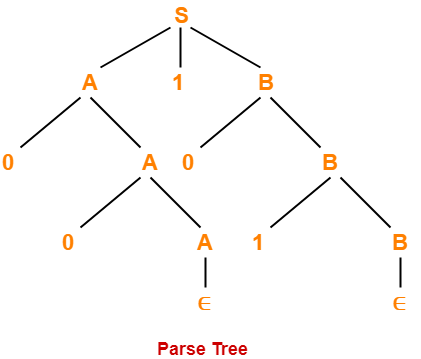
→ **A**101                (Using B → ∈)

→ 0**A**101              (Using A → 0A)

→ 00**A**101            (Using A → 0A)

→ 00101               (Using A → ∈)

### **3. Parse Tree-**



* Whether we consider the leftmost derivation or rightmost derivation, we get the above parse tree.
* The reason is given grammar is unambiguous.