# **Context free grammar**

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

1. G= (V, T, P, S)

Where,

**G** describes the grammar

**T** describes a finite set of terminal symbols.

**V** describes a finite set of non-terminal symbols

**P** describes a set of production rules

**S** is the start symbol.

In CFG, the start symbol is used to derive the string. You 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the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.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:**

L= {wcwR | w € (a, b)\*}

**Production rules:**

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

1. S ⇒ aSa
2. S ⇒ abSba
3. S ⇒ abbSbba
4. S ⇒ abbcbba

By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

# **Capabilities of CFG**

There are the various capabilities of CFG:

* Context free grammar is useful to describe most of the programming languages.
* If the grammar is properly designed then an efficientparser can be constructed automatically.
* Using the features of associatively & precedence information, suitable grammars for expressions can be constructed.
* Context free grammar is capable of describing nested structures like: balanced parentheses, matching begin-end, corresponding if-then-else's & so on.

# **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 replaced with production rule.

### **Left-most Derivation**

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

### **Example:**

**Production rules:**

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

a - b + c

**The left-most derivation is:**

1. S = S + S
2. S = S - S + S
3. S = a - S + S
4. S = a - b + S
5. S = a - b + c

## **Right-most Derivation**

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

### **Example:**

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

a - b + c

**The right-most derivation is:**

1. S = S - S
2. S = S - S + S
3. S = S - S + c
4. S = S - b + c
5. S = a - b + c

# **Parse tree**

* Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
* In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
* It is the graphical representation of symbol that can be terminals or non-terminals.
* 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.

## **The parse tree follows these points:**

* All leaf nodes have to be terminals.
* All interior nodes have to be non-terminals.
* In-order traversal gives original input string.

### **Example:**

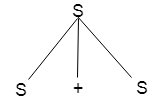
**Production rules:**

1. T= T + T | T \* T
2. T = a|b|c

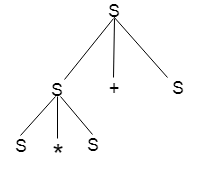
Input:

a \* b + c

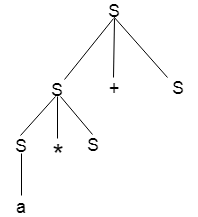
### **Step 1:**



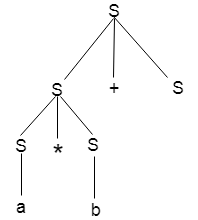
### **Step 2:**



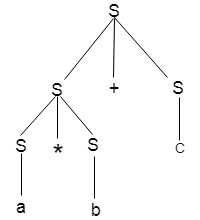
### **Step 3:**



### **Step 4:**



### **Step 5:**

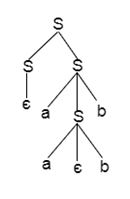
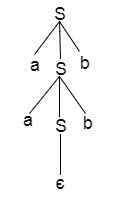
# **Ambiguity**

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

### **Example:**

1. S = aSb | SS
2. S = ∈

For the string aabb, the above grammar generates two parse trees:

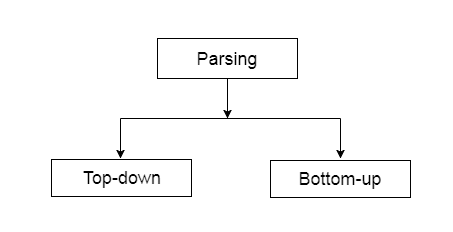
If the grammar has ambiguity then it is not good for a compiler construction. No method can automatically detect and remove the ambiguity but you can remove ambiguity by re-writing the whole grammar without ambiguity.

## **parser**

Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase.

A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.

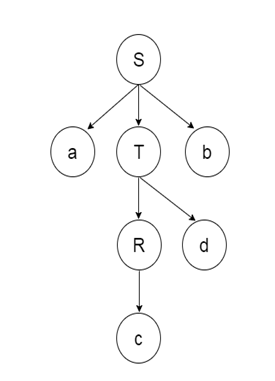
Parsing is of two types: top down parsing and bottom up parsing.



## **Top down parsing**

* The top down parsing is known as recursive parsing or predictive parsing.
* Bottom up parsing is used to construct a parse tree for an input string.
* In the top down parsing, the parsing starts from the start symbol and transform it into the input symbol.

Parse Tree representation of input string "acdb" is as follows:



## **Bottom up parsing**

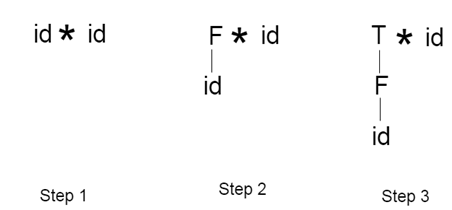
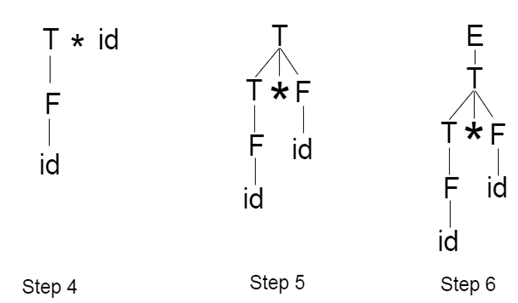
* Bottom up parsing is also known as shift-reduce parsing.
* Bottom up parsing is used to construct a parse tree for an input string.
* In the bottom up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse.

### **Example**

**Production**

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

Parse Tree representation of input string "id \* id" is as follows:

Bottom up parsing is classified in to various parsing. These are as follows:

1. Shift-Reduce Parsing
2. Operator Precedence Parsing
3. Table Driven LR Parsing
4. LR( 1 )
5. SLR( 1 )
6. CLR ( 1 )
7. LALR( 1 )