# UNIT - II

# UNIT - II

# TOPDOWN PARSING

## 1. Context-free Grammars: Definition:

Formally, a context-free grammar G is a 4-tuple G = (V, T, P, S), where:

1. V is a finite set of variables (or nonterminals). These describe sets of "related" strings.
2. T is a finite set of terminals (i.e., tokens).
3. P is a finite set of productions, each of the form

   $A \rightarrow \alpha$

   where $A \in V$ is a variable, and $\alpha \in (V \cup T)^*$ is a sequence of terminals and nonterminals. $S \in V$ is the start symbol.

**Example of CFG:**

E ==>EAE | (E) | -E | id A==> + | - | * | / |

**Where E, A are the non-terminals while id, +, *, -, /,(, ) are the terminals. 2. Syntax analysis:**

In syntax analysis phase the source program is analyzed to check whether if conforms to the source language's syntax, and to determine its phase structure. This phase is often separated into two phases:

- Lexical analysis: which produces a stream of tokens?
- Parser: which determines the phrase structure of the program based on the context-free grammar for the language?

## 2.1 PARSING:

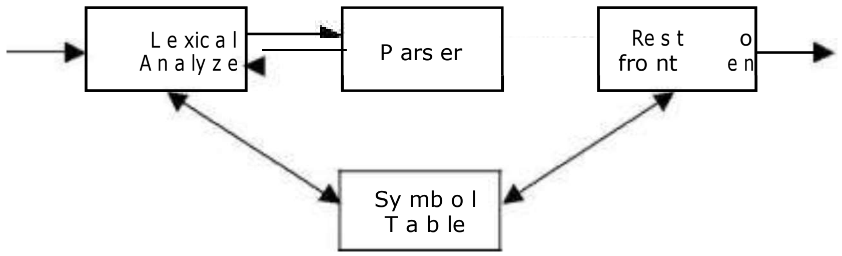
Parsing is the activity of checking whether a string of symbols is in the language of some grammar, where this string is usually the stream of tokens produced by the lexical analyzer. If the string is in the grammar, we want a parse tree, and if it is not, we hope for some kind of error message explaining why not.

There are two main kinds of parsers in use, named for the way they build the parse trees:

- Top-down: A top-down parser attempts to construct a tree from the root, applying productions forward to expand non-terminals into strings of symbols.
- Bottom-up: A Bottom-up parser builds the tree starting with the leaves, using productions in reverse to identify strings of symbols that can be grouped together.

In both cases the construction of derivation is directed by scanning the input sequence from left to right, one symbol at a time.

**Parse Tree:**



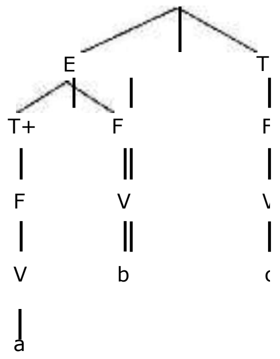
A parse tree is the graphical representation of the structure of a sentence according to its grammar.

**Example:**
Let the production P is:

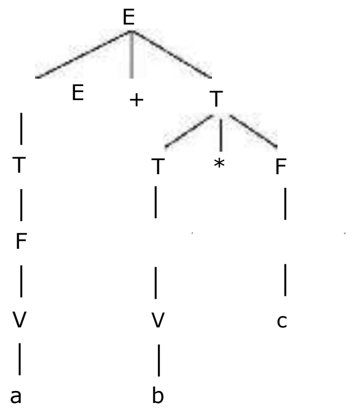
$E \rightarrow T \mid E+T$
$T \rightarrow F \mid T*F$
$F \rightarrow V \mid (E)$
$V \rightarrow a \mid b \mid c \mid d$

The parse tree may be viewed as a representation for a derivation that filters out the choice regarding the order of replacement.

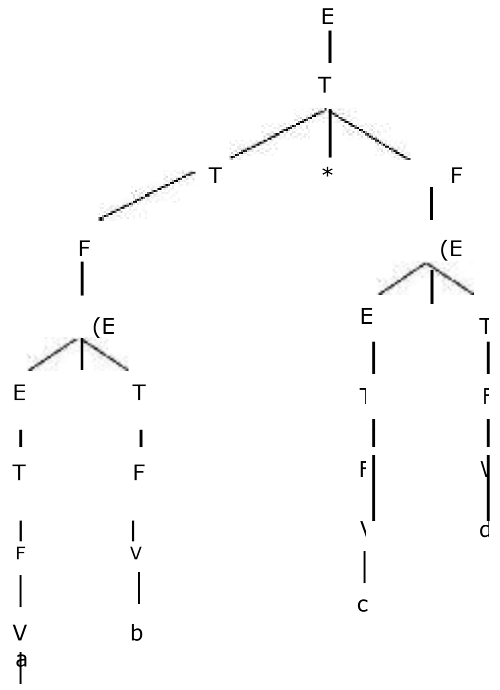
Parse tree for a * b + c



Parse tree for a + b * c is:



Parse tree for (a * b) * (c + d)



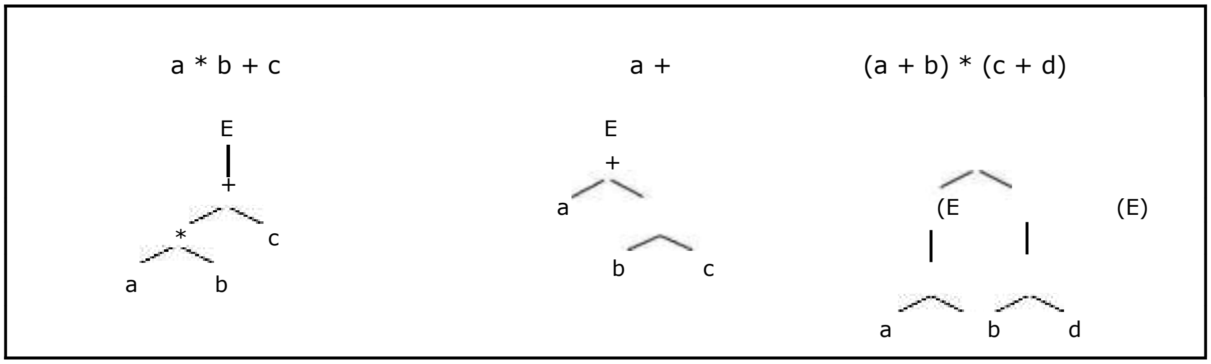
**2.2 SYNTAX TREES:**

Parse tree can be presented in a simplified form with only the relevant structure information by:

- Leaving out chains of derivations (whose sole purpose is to give operators difference precedence).

- Labeling the nodes with the operators in question rather than a non-terminal.

The simplified Parse tree is sometimes called as structural tree or syntax tree.



Synt a x T re e s

**Syntax Error Handling:**

If a compiler had to process only correct programs, its design & implementation would be greatly simplified. But programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying and locating errors.The programs contain errors at many different levels.
For example, errors can be:

1) Lexical – such as misspelling an identifier, keyword or operator
2) Syntactic – such as an arithmetic expression with un-balanced parentheses.
3) Semantic – such as an operator applied to an incompatible operand.
4) Logical – such as an infinitely recursive call.

Much of error detection and recovery in a compiler is centered around the syntax analysis phase. The goals of error handler in a parser are:

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

**2.3Ambiguity:**

Several derivations will generate the same sentence, perhaps by applying the same productions in a different order. This alone is fine, but a problem arises if the same sentence has two distinct parse trees. A grammar is ambiguous if there is any sentence with more than one parse tree.

Any parses for an ambiguous grammar has to choose somehow which tree to return. There are a number of solutions to this; the parser could pick one arbitrarily, or we can provide some hints about which to choose. Best of all is to rewrite the grammar so that it is not ambiguous.

There is no general method for removing ambiguity. Ambiguity is acceptable in spoken languages. Ambiguous programming languages are useless unless the ambiguity can be resolved.

Fixing some simple ambiguities in a grammar:

| | Ambiguous | language | unambiguous |
|---|---|---|---|
| (i) | $A \rightarrow B \mid AA$ | Lists of one or more B's | $A \rightarrow BC$ |
| | | $C \rightarrow A \mid E$ | |
| (ii) | $A \rightarrow B \mid A;A$ | Lists of one or more B's with punctuation | $A \rightarrow BC$ |
| | | $C \rightarrow ;A \mid E$ | |
| (iii) | $A \rightarrow B \mid AA \mid E$ | lists of zero or more B's | $A \rightarrow BA \mid E$ |

Any sentence with more than two variables, such as (arg, arg, arg) will have multiple parse trees.

## 2.4 Left Recursion:

If there is any non terminal A, such that there is a derivation A the$\overset{+}{\Rightarrow}$ A $\alpha$ for some string$\alpha$, then grammar is left recursive.

Algorithm for eliminating left Recursion:

1. Group all the A productions together like this: $A \Rightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid - - - \mid A\,\alpha_m \mid \beta_1 \mid \beta_2 \mid - -$

   $- \mid \beta_n$

   Where,

   A is the left recursive non-terminal,
   $\alpha$ is any string of terminals and
   $\beta$ is any string of terminals and non terminals that does not begin with A.

2. Replace the above A productions by the following: $A \Rightarrow \beta_1\,A^I \mid \beta_2\,A^I \mid - - - \mid \beta_n\,A^I$

   $A^I \Rightarrow \alpha_1\,A^I \mid \alpha_2\,A^I \mid - - - \mid \alpha_m\,A^I \mid \in$ Where, $A^I$ is a new non terminal.

Top down parsers cannot handle left recursive grammars.

If our expression grammar is left recursive:

- This can lead to non termination in a top-down parser.
- for a top-down parser, any recursion must be right recursion.
- we would like to convert the left recursion to right recursion.

**Example 1:**
Remove the left recursion from the production: $A \rightarrow \underbrace{A\ \alpha}_{\textbf{Left Recursive. Eliminate}} \mid \beta$

Applying the transformation yields:

$A \rightarrow \beta\ A^{I}$

$A^{I} \rightarrow \underset{\uparrow}{\alpha}\ A^{I} \mid \in$

Remaining part after A.

**Example 2:**
Remove the left recursion from the productions:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

Applying the transformation yields:

$E \rightarrow T\ E^{I}$ $\qquad$ $T \rightarrow F\ T^{I}$

$E^{I} \rightarrow T\ E^{I} \mid \in$ $\qquad$ $T^{I} \rightarrow * F\ T^{I} \mid \in$

**Example 3:**
Remove the left recursion from the productions:

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T/F \mid F$

Applying the transformation yields:

$E \rightarrow T\ E^{I}$ $\qquad$ $T \rightarrow F\ T^{I}$

$E \rightarrow + T\ E^{I} \mid - T\ E^{I} \mid \in$ $\qquad$ $T^{I} \rightarrow * F\ T^{I} \mid /F\ T^{I} \mid \in$

**Example 4:**
Remove the left recursion from the productions:

$S \rightarrow A\ a \mid b$

$A \rightarrow A\ c \mid S\ d \mid \in$

1. The non terminal S is left recursive because $S \rightarrow A\ a \rightarrow S\ d\ a$ But it is not immediate left recursive.
2. Substitute S-productions in $A \rightarrow S\ d$ to obtain:

   $A \rightarrow A\ c \mid A\ a\ d \mid b\ d \mid \in$
3. Eliminating the immediate left recursion:

$S \rightarrow A\ a \mid b$
$A \rightarrow b\ d\ A^I \mid A^I$
$A^I \rightarrow c\ A^I \mid a\ d\ A^I \mid \in$

**Example 5:**
Consider the following grammar and eliminate left recursion. $S \rightarrow A\ a \mid b$
$A \rightarrow S\ c \mid d$

The nonterminal S is left recursive in two steps: $S \rightarrow A\ a \rightarrow S\ c\ a \rightarrow A\ a\ c\ a \rightarrow S\ c\ a\ c\ a$ - - -
Left recursion causes the parser to loop like this, so remove: Replace $A \rightarrow S\ c \mid d$ by $A \rightarrow A\ a\ c \mid b\ c \mid d$
and then by using Transformation rules: $A \rightarrow b\ c\ A^I \mid d\ A^I$
$A^I \rightarrow a\ c\ A^I \mid \in$

**2.5 Left Factoring:**
Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
When it is not clear which of two alternative productions to use to expand a non-terminal A, we may be able to rewrite the productions to defer the decision until we have some enough of the input to make the right choice.

**Algorithm:**
For all $A \in$ non-terminal, find the longest prefix $\alpha$ that occurs in two or more right-hand sides of A.
If $\alpha \neq \in$ then replace all of the A productions, $A \rightarrow \alpha\ \beta_I \mid \alpha\ \beta_2 \mid - - - \mid \alpha\ \beta_n \mid r$
With
$A \rightarrow \alpha\ A^I \mid r$
$A^I \rightarrow \beta_I \mid \beta_2 \mid - - - \mid \beta_n \mid \in$
Where, $A^I$ is a new element of non-terminal. Repeat until no common prefixes remain.
It is easy to remove common prefixes by left factoring, creating new non-terminal.
**For example consider:**
$V \rightarrow \alpha\ \beta \mid \alpha\ r$ Change to:
$V \rightarrow \alpha\ V^I\ V^I \rightarrow \beta \mid r$

**Example 1:**
Eliminate Left factoring in the grammar: $S \rightarrow V$ := int
$V \rightarrow$ alpha '[' int ']' | alpha

Becomes:

$S \rightarrow V := int$
$V \rightarrow alpha\ V^I$
$V^I \rightarrow$ ’[‘ int ’] | $\in$

## 2.6 TOP DOWN PARSING:

Top down parsing is the construction of a Parse tree by starting at start symbol and “guessing” each derivation until we reach a string that matches input. That is, construct tree from root to leaves.

The advantage of top down parsing in that a parser can directly be written as a program. Table-driven top-down parsers are of minor practical relevance. Since bottom-up parsers are more powerful than top-down parsers, bottom-up parsing is practically relevant.

For example, let us consider the grammar to see how top-down parser works:

**S $\rightarrow$ if E then S else S | while E do S | print**
**E $\rightarrow$ true | False | id**

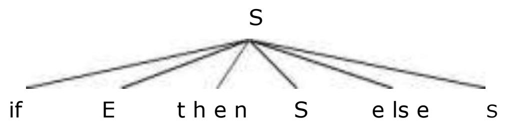
The input token string is: If id then while true do print else print.

1. Tree:

S

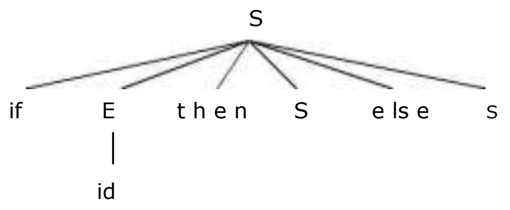
Input: if id then while true do print else print.
Action: Guess for S.

2. Tree:



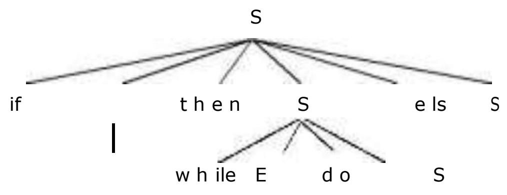
Input: if id then while true do print else print.
Action: if matches; guess for E.

3. Tree:



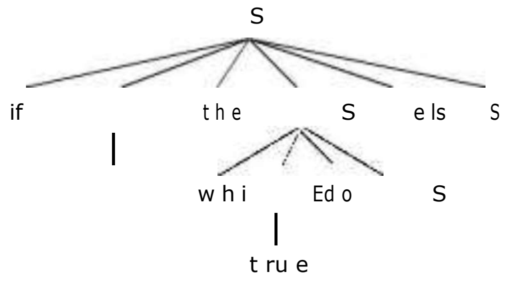
Input: id then while true do print else print.
Action: id matches; then matches; guess for S.

4. Tree:



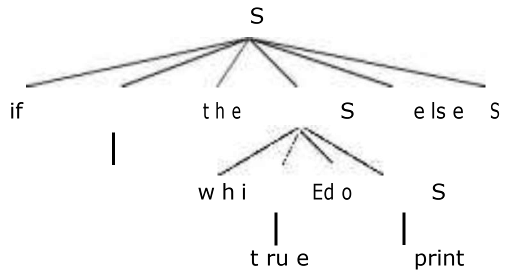
Input: while true do print else print.
Action: while matches; guess for E.

5. Tree:



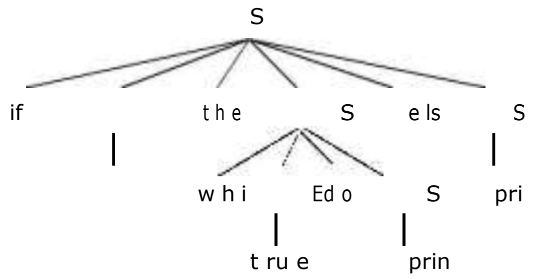
Input:  true do print else print
Action:true matches; do matches; guess S.

6. Tree:



Input:  print else print.
Action: print matches; else matches; guess for S.

7. Tree:



Input: print.
Action: print matches; input exhausted; done.

**2.6.1. Recursive Descent Parsing:**

Top-down parsing can be viewed as an attempt to find a left most derivation for an input string. Equivalently, it can be viewd as a attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

The special case of recursive –decent parsing, called predictive parsing, where no backtracking is required. The general form of top-down parsing, called recursive descent, that may involve backtracking, that is, making repeated scans of the input.

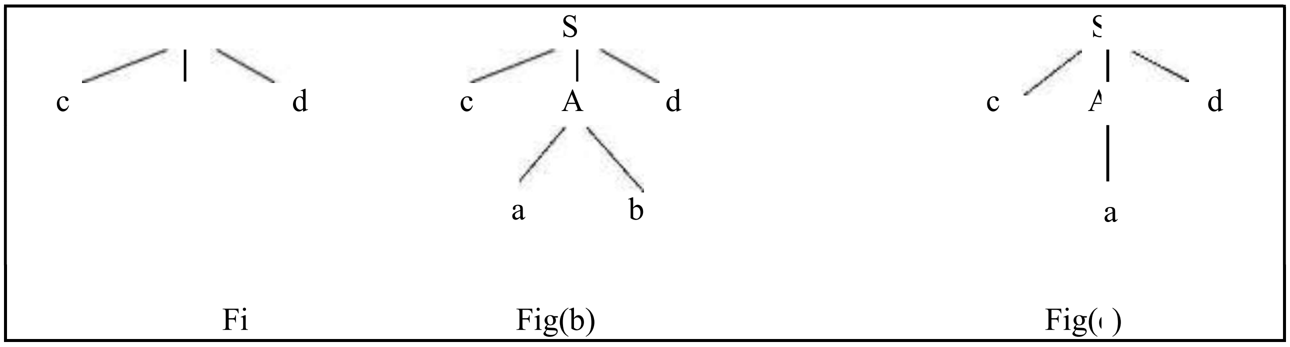
Recursive descent or predictive parsing works only on grammars where the first terminal symbol of each sub expression provides enough information to choose which production to use.

Recursive descent parser is a top down parser involving backtracking. It makes a repeated scans of the input. Backtracking parsers are not seen frequently, as backtracking is very needed to parse programming language constructs.

**Example:** consider the grammar

S→cAd
A→ab|a

And the input string w=cad. To construct a parse tree for this string top-down, we initially create a tree consisting of a single node labeled scan input pointer points to c, the first symbol of w. we then use the first production for S to expand tree and obtain the tree of Fig(a).



Fi Fig(b) Fig( )

The left most leaf, labeled c, matches the first symbol of w, so we now advance the input pointer to a ,the second symbol of w, and consider the next leaf, labeled A. We can then expand A using the first alternative for A to obtain the tree in Fig (b). we now have a match for the second input symbol so we advance the input pointer to d, the third, input symbol, and compare d against the next leaf, labeled b. since b does not match the d ,we report failure and go back to A to see where there is any alternative for Ac that we have not tried but that might produce a match.

In going back to A, we must reset the input pointer to position2,we now try second alternative for A to obtain the tree of Fig(c).The leaf matches second symbol of w and the leaf d matches the third symbol .

The left recursive grammar can cause a recursive- descent parser, even one with backtracking, to go into an infinite loop.That is ,when we try to expand A, we may eventually find ourselves again trying to ecpand A without Having consumed any input.

**2.6.2. Predictive Parsing:**

Predictive parsing is top-down parsing without backtracking or look a head. For many languages, make perfect guesses (avoid backtracking) by using 1-symbol look-a-head. i.e., if:

**$A \rightarrow \alpha_1 \mid \alpha_2 \mid - - - \mid \alpha_n$.**

**Choose correct $\alpha_i$ by looking at first symbol it derive. If $\in$ is an alternative, choose it last.**

This approach is also called as predictive parsing. There must be at most one production in order to avoid backtracking. If there is no such production then no parse tree exists and an error is returned.

The crucial property is that, the grammar must not be left-recursive.

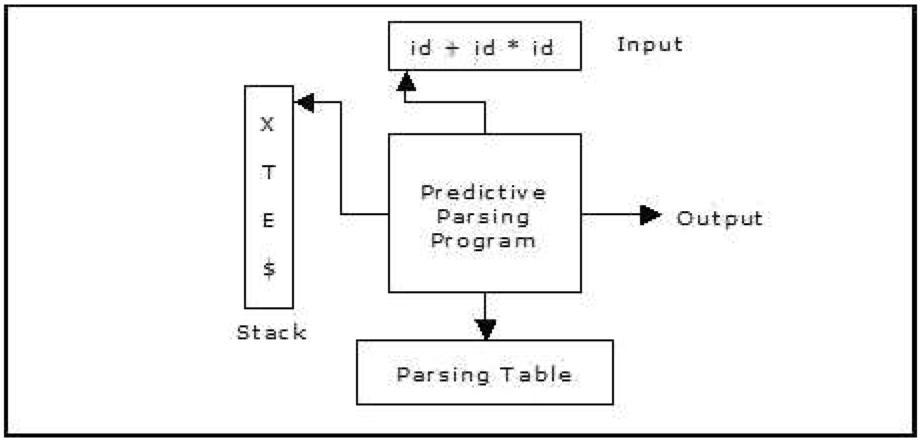
Predictive parsing works well on those fragments of programming languages in which keywords occurs frequently.

For example:

stmt → **if** exp **then** stmt **else** stmt | **while** expr **do** stmt
| **begin** stmt-list **end**.

then the keywords if, while and begin tell, which alternative is the only one that could possibly succeed if we are to find a statement.

The model of predictive parser is as follows:



A predictive parser has:

- Stack
- Input
- Parsing Table
- Output

The input buffer consists the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string.

The stack consists of a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially the stack consists of the start symbol of the grammar on the top of $.

Recursive descent and LL parsers are often called predictive parsers, because they operate by predicting the next step in a derivation.

**The algorithm for the Predictive Parser Program is as follows: Input:** A string w and a parsing table M for grammar G

**Output:** if w is in L(g),a leftmost derivation of w; otherwise, an error indication.

**Method:** Initially, the parser has $S on the stack with S, the start symbol of G on top, and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is:

```
Set ip to point to the first symbol of w$; repeat
    let x be the top stack symbol and a the symbol pointed to by ip; if X is a terminal or $
    then
        if X = a then
            pop X from the stack and advance ip else error()
    else                /* X is a non-terminal */
        if M[X, a] = X → Y1 Y2.................... Yk then begin
```

```
                pop X from the stack;
                push $Y_k$, $Y_{k-1}$, .................... $Y_1$ onto the stack, with $Y_1$ on top; output the
                production X $\rightarrow$ $Y_1$ $Y_2$.............. $Y_k$
            end
            else error()
        until X = $ /*stack is empty*/
```

**2.6.3 FIRST and FOLLOW:**

The construction of a predictive parser is aided by two functions with a grammar G. these functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible. Sets of tokens yielded by the **FOLLOW** function can also be used as synchronizing tokens during pannic-mode error recovery.

If α is any string of grammar symbols, let FIRST (α) be the set of terminals that begin the strings derived from α. If α=>€,then € is also in FIRST(α).

Define FOLLOW (A), for nonterminals A, to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exist a derivation of the form S=>αAaβ for some α and β. If A can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A).

**Computation of FIRST ():**

To compute **FIRST(X)** for all grammar symbols X, apply the following rules until no more terminals or € can be added to any FIRST set.

- If X is terminal, then FIRST(X) is {X}.
- If X→€ is production, then add € to FIRST(X).
- If X is nonterminal and X→$Y_1$ $Y_2$……$Y_k$ is a production, then place a in FIRST(X) if for some i,a is in FIRST($Y_i$),and € is in all of FIRST($Y_i$),and € is in all of FIRST($Y_1$),….. FIRST($Y_{i-1}$);that is $Y_1$.................... $Y_{i-1}$=>€.if € is in FIRST($Y_j$), for all j=,2,3…….k, then add € to FIRST(X).for example, everything in FIRST($Y_1$) is surely in FIRST(X).if $Y_1$ does not derive €,then we add nothing more to FIRST(X),but if $Y_1$=>€,then we add FIRST($Y_2$) and so on.

**FIRST (A) = FIRST ($\alpha_1$) U FIRST ($\alpha_2$) U - - - U FIRST ($\alpha_n$) Where, A → $\alpha_1$ | $\alpha_2$ |----- |$\alpha_n$, are all the productions for A. FIRST (A$\alpha$) = if $\in$ $\notin$ FIRST (A) then FIRST (A)**
**else (FIRST (A) - {$\in$}) U FIRST ($\alpha$)**

**Computation of FOLLOW ():**

To compute **FOLLOW (A)** for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

- Place \$ in FOLLOW(s), where S is the start symbol and \$ is input right end marker .
- If there is a production $A \to \alpha B \beta$,then everything in FIRST($\beta$) except for € is placed in FOLLOW(B).
- If there is production $A \to \alpha B$, or a production $A \to \alpha B \beta$ where FIRST ($\beta$) contains € (i.e.,$\beta \to$€),then everything in FOLLOW(A)is in FOLLOW(B).

**Example:**

**Construct the FIRST and FOLLOW for the grammar:**

$A \to BC \mid EFGH \mid H$
$B \to b$
$C \to c \mid \in$
$E \to e \mid \in$
$F \to CE$
$G \to g$
$H \to h \mid \in$

**Solution:**

1. Finding first () set:
    1. first (H) = first (h) $\cup$ first ($\varepsilon$) = {h, $\varepsilon$}
    2. first (G) = first (g) = {g}
    3. first (C) = first (c) $\cup$ first ($\varepsilon$) = c, $\varepsilon$}
    4. first (E) = first (e) $\cup$ first ($\varepsilon$) = {e, $\varepsilon$}
    5. first (F) = first (CE) = (first (c) - {$\varepsilon$}) $\cup$ first (E)

       = (c, $\varepsilon$} {$\varepsilon$}) $\cup$ {e, $\varepsilon$} = {c, e, $\varepsilon$}
    6. first (B) = first (b)={b}
    7. first (A) = first (BC) $\cup$ first (EFGH) $\cup$ first (H)

       = first (B) $\cup$ (first (E) – { $\varepsilon$}) $\cup$ first (FGH) $\cup$ {h, $\varepsilon$}

       = {b, h, $\varepsilon$} $\cup$ {e} $\cup$ (first (F) – {$\varepsilon$}) $\cup$ first (GH)

       = {b, e, h, $\varepsilon$} $\cup$ {C, e} $\cup$ first (G)

       = {b, c, e, h, $\varepsilon$} $\cup$ {g} = {b, c, e, g, h, $\varepsilon$}

2. Finding follow() sets:

1. follow(A) = {\$}
2. follow(B) = first(C) – {ε} ∪ follow(A) = {C, \$}
3. follow(G) = first(H) – {ε} ∪ follow(A)

   ={h, ε} – {ε} ∪ {\$} = {h, \$}
4. follow(H) = follow(A) = {\$}
5. follow(F) = first(GH) – {ε} = {g}
6. follow(E) = first(FGH) m- {ε} ∪ follow(F)

   = ((first(F) – {ε}) ∪ first(GH)) – {ε} ∪ follow(F)

   = {c, e} ∪ {g} ∪ {g} = {c, e, g}
7. follow(C) = follow(A) ∪ first (E) – {ε} ∪ follow (F)

   ={\$} ∪ {e, ε} ∪ {g} = {e, g, \$}

**Example 1:**

Construct a predictive parsing table for the given grammar or Check whether the given grammar is LL(1) or not.

$E \rightarrow E + T \mid T$

**$T \rightarrow T * F \mid F$ $F \rightarrow (E) \mid id$**

**Step 1:**

Suppose if the given grammar is left Recursive then convert the given grammar (and ∈) into non-left Recursive grammar (as it goes to infinite loop).

**$E \rightarrow T E^I$**

**$E^I \rightarrow + T E^I \mid \in$ $T^I \rightarrow F T^I$**

**$T^I \rightarrow * F T^I \mid \in$ $F \rightarrow (E) \mid id$**

**Step 2:**

Find the FIRST(X) and FOLLOW(X) for all the variables.

The variables are: $\{E, E^I, T, T^I, F\}$

Terminals are: {+, *, (, ), id} and \$

**Computation of FIRST() sets:**

**FIRST (F) = FIRST ((E)) U FIRST (id) = {(, id}**
**FIRST ($T^I$) = FIRST ($*FT^I$) U FIRST ($\in$) = {*, $\in$}**
**FIRST (T) = FIRST ($FT^I$) = FIRST (F) = {(, id}**
**FIRST ($E^I$) = FIRST ($+TE^I$) U FIRST ($\in$) = {+, $\in$}**
**FIRST (E) = FIRST ($TE^I$) = FIRST (T) = {(, id}**

Computation of FOLLOW () sets:

| | Relevant production |
|---|---|
| FOLLOW (E) = {\$} U FIRST ( ) ) = {\$, )} | $F \rightarrow (E)$ |
| FOLLOW ($E^I$) = FOLLOW (E) = {\$, )} | $E \rightarrow TE^I$ |
| FOLLOW (T) = (FIRST ($E^I$) - {$\in$}) U FOLLOW (E) U FOLLOW ($E^I$) <br> = {+, | $E \rightarrow TE^I$ <br> $E^I \rightarrow +TE^I$ |
| FOLLOW ($T^I$) = FOLLOW (T) = {+, ), \$} | $T \rightarrow FT^I$ |
| FOLLOW (F) = (FIRST ($T^I$) - {$\in$}) U FOLLOW (T) U FOLLOW ($T^I$) <br> = {*, +, | $T \rightarrow T^I$ |

**Step 3:**
Construction of parsing table:

| Termina / Variables | + | | ( | ) | id | \$ |
|---|---|---|---|---|---|---|
| E | | | $E \rightarrow TE$ | | $E \rightarrow TE^I$ | |
| $E^I$ | $E^I \rightarrow +TE^I$ | | | $E^I \rightarrow \in$ | | $E^I \rightarrow \in$ |
| T | | | $T \rightarrow FT$ | | $T \rightarrow FT^I$ | |
| $T^I$ | $T^I \rightarrow \in$ | $T^I \rightarrow *F$ | | $T^I \rightarrow \in$ | | $T^I \rightarrow \in$ |
| F | | | $F \rightarrow (E)$ | | $F \rightarrow id$ | |

Table 3.1. Parsing Table

**Fill the table with the production on the basis of the FIRST($\alpha$). If the input symbol is an $\in$ in FIRST($\alpha$), then goto FOLLOW($\alpha$) and fill $\alpha \rightarrow \in$, in all those input symbols.**

**3.1. Let us start with the non-terminal E, FIRST(E) = {(, id}. So, place the production $E \rightarrow TE^I$ at ( and id.**

**3.2. For the non-terminal $E^I$, FIRST ($E^I$) = {+, $\in$}.**

So, place the production $E^I \rightarrow +TE^I$ at + and also as there is a $\in$ in FIRST($E^I$), see FOLLOW($E^I$) = {\$, )}. So write the production $E^I \rightarrow \in$ at the place \$ and ).

Similarly:

**3.3. For the non-terminal T, FIRST(T) = {(, id}. So place the production $T \rightarrow FT^I$ at ( and id.**

**3.4. For the non-terminal $T^I$, FIRST ($T^I$) = {*, ∈}**

So place the production $T^I \rightarrow *FT^I$ at * and also as there is a ∈ in FIRST ($T^I$), see FOLLOW ($T^I$) = {+, $, )}, so write the production $T^I \rightarrow \in$ at +, $ and ).

**3.5. For the non-terminal F, FIRST (F) = {(, id}.**
**So place the production F → id at id location and F → (E) at ( as it has two productions.**

**3.6.** Finally, make all undefined entries as error.
As these were no multiple entries in the table, hence the given grammar is LL(1).

**Step 4:**
Moves made by predictive parser on the input id + id * id is:

| STACK | INPUT | REMARKS |
|---|---|---|
| $ **E** | **id** + id * id $ | E and id are not identical; so see E on id in parse table, the production is $E \rightarrow TE^I$; pop E, push $E^I$ and T i.e., move in reverse order. |
| $ $E^I$ **T** | **id** + id * id $ | See T on id the production is $T \rightarrow F\ T^I$ ; Pop T, push $T^I$ and F; Proceed until both are identical. |
| $ $E^I$ $T^I$ **F** | **id** + id * id $ | F → id |
| $ $E^I$ $T^I$ **id** | **id** + id * id $ | Identical; pop id and remove id from input symbol. |
| $ $E^I$ $\mathbf{T^I}$ | + id * | See $T^I$ on +; $T^I \rightarrow \in$ so, pop $T^I$ |
| $ $\mathbf{E^I}$ | + id * | See $E^I$ on +; $E^I \rightarrow +T\ E^I$; push $E^I$ , + and T |
| $ $E^I$ T + | + id * | Identical; pop + and remove + from input symbol. |
| $ $E^I$ **T** | **id** * | |
| $ $E^I$ $T^I$ **F** | **id** * | $T \rightarrow F\ T^I$ |
| $ $E^I$ $T^I$ **id** | **id** * | F → id |
| $ $E^I$ $\mathbf{T^I}$ | * | |
| $ $E^I$ $T^I$ F * | * | $T^I \rightarrow *\ F\ T^I$ |
| $ $E^I$ $T^I$ **F** | | |
| $ $E^I$ $T^I$ **id** | | F → id |

|  |  |  |
|---|---|---|
| $\$ E^{I} \mathbf{T^{I}}$ |  | $T^{I} \rightarrow \in$ |
| $\$ \mathbf{E^{I}}$ |  | $E^{I} \rightarrow \in$ |
| **$** |  | Accept. |

Table 3.2 Moves made by the parser on input id + id * id

Predictive parser accepts the given input string. We can notice that $ in input and stuck, i.e., both are empty, hence accepted.

### 2.6.3 LL (1) Grammar:

**The first L stands for "Left-to-right scan of input". The second L stands for "Left-most derivation". The '1' stands for "1 token of look ahead".**
**No LL (1) grammar can be ambiguous or left recursive.**

If there were no multiple entries in the Recursive decent parser table, the given grammar is LL (1).

If the grammar G is ambiguous, left recursive then the recursive decent table will have atleast one multiply defined entry.

The weakness of LL(1) (Top-down, predictive) parsing is that, must predict which production to use.

### Error Recovery in Predictive Parser:

Error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appear. Its effectiveness depends on the choice of synchronizing set. The Usage of FOLLOW and FIRST symbols as synchronizing tokens works reasonably well when expressions are parsed.

For the constructed table., fill with **synch** for rest of the input symbols of FOLLOW set and then fill the rest of the columns with **error** term.

| Terminal / Variables | + | * | ( | ) | id | $ |
|---|---|---|---|---|---|---|
| E | error | error | $E \rightarrow TE$ | synch | $E \rightarrow TE^{I}$ | synch |
| $E^{I}$ | $E^{I} \rightarrow +TE^{I}$ | error | error | $E^{I} \rightarrow \in$ | error | $E^{I} \rightarrow \in$ |
| T | synch | error | $T \rightarrow FT$ | synch | $T \rightarrow FT^{I}$ | synch |
| $T^{I}$ | $T^{I} \rightarrow \in$ | $T^{I} \rightarrow *F$ | error | $T^{I} \rightarrow \in$ | error | $T^{I} \rightarrow \in$ |
| F | synch | synch | $F \rightarrow (E)$ | synch | $F \rightarrow id$ | synch |

Table3.3 :Synchronizing tokens added to parsing table for table 3.1.

If the parser looks up entry in the table as synch, then the non terminal on top of the stack is popped in an attempt to resume parsing. If the token on top of the stack does not match the input symbol, then pop the token from the stack.

The moves of a parser and error recovery on the erroneous input) id*+id is as follows:

| STACK | IN | REMARKS |
|---|---|---|
| $ **E** | **)** id * + | Error, skip ) |
| $ **E** | **id** * + | |
| $ $E^I$ **T** | **id** * + | |
| $ $E^I$ $T^I$ **F** | **id** * + | |
| $ $E^I$ $T^I$ **id** | **id** * + | |
| $ $E^I$ $\mathbf{T^I}$ | * + | |
| $ $E^I$ $T^I$ F * | * + | |
| $ $E^I$ $T^I$ **F** | + | Error; F on + is synch; F has been popped. |
| $ $E^I$ $\mathbf{T^I}$ | + | |
| $ $\mathbf{E^I}$ | + | |
| $ $E^I$ T + | + | |
| $ $E^I$ **T** | | |
| $ $E^I$ $T^I$ **F** | | |
| $ $E^I$ $T^I$ **id** | | |
| $ $E^I$ $\mathbf{T^I}$ | | |
| $ $\mathbf{E^I}$ | | |
| **$** | | Accept. |

Table 3.4. Parsing and error recovery moves made by predictive parser

**Example 2:**

Construct a predictive parsing table for the given grammar or Check whether the given grammar is LL(1) or not.

$S \rightarrow iEtSS^I \mid a$
$S^I \rightarrow eS \mid \in$
$E \rightarrow b$

**Solution:**

1. Computation of First () set:

   1. First (E) = first (b) = {b}
   2. First ($S^I$) = first (eS) ∪ first (ε) = {e, ε}
   3. first (S) = first (iEtSS$^I$) ∪ first (a) = {i, a}

2. Computation of follow() set:
   1. follow (S) = {\$} ∪ first ($S^I$) – {ε} ∪ follow (S) ∪ follow ($S^I$)
      = {\$} ∪ {e} = {e, \$}
   2. follow ($S^I$) = follow (S) = {e, \$}
   3. follow (E) = first (tSS$^I$) = {t}

3. The parsing table for this grammar is:

| | a | b | e | i | t | |
|---|---|---|---|---|---|---|
| S | $S \to a$ | | | $S \to$ iEtSS$^I$ | | |
| $S^I$ | | | $S^I \to \in$<br>$S^I \to e$ | | | $S^I \to \in$ |
| E | | $E \to b$ | | | | |

As the table multiply defined entry. The given grammar is not LL(1).

**Example 3:**

**Construct the FIRST and FOLLOW and predictive parse table for the grammar:**

$S \to AC\$$
$C \to c \mid \in$
$A \to aBCd \mid BQ \mid \in$
$B \to bB \mid d$
$Q \to q$

**Solution:**

1. Finding the first () sets: First (Q) = {q}

   First (B) = {b, d}

First (C) = {c, ε}

First (A) = First (aBCd) ∪ First (BQ) ∪ First (ε)

= {a} ∪ First (B) ∪ First (d) ∪{ε}

= {a} ∪ First (bB) ∪ First (d) ∪ {ε}

= {a} ∪ {b} ∪ {d} ∪ {ε}

= {a, b, d, ε} First (S) = First (AC$)

= (First (A) – {ε}) ∪ (First (C) – {ε}) ∪ First (ε)

= ({a, b, d, ε} – {ε}) ∪ ({c, ε} – {ε}) ∪ {ε}

= {a, b, d, c, ε}

2. Finding Follow () sets: Follow (S) = {#}

Follow (A) = (First (C) – {ε}) ∪ First ($) = ({c, ε} – {ε}) ∪ {$} Follow (A) = {c, $}

Follow (B) = (First (C) – {ε}) ∪ First (d) ∪ First (Q)

= {c} ∪ {d} ∪ {q} = {c, d, q} Follow (C) = (First ($) ∪ First (d) = {d, $}

Follow (Q) = (First (A) = {c, $}

3. The parsing table for this grammar is:

| | a | b | c | D | q | $ | |
|---|---|---|---|---|---|---|---|
| S | S → AC$ | S → AC $ | S → AC $ | S → AC $ | | S → AC $ | |
| A | A → aBCd | A → BQ | A → ε | A → BQ | | A → ε | |
| B | | B → bB | | B → d | | | |
| C | | | C → c | C → ε | | C → ε | |
| Q | | | | | Q → q | | |

4. Moves made by predictive parser on the input abdcdc$ is:

| Stack symbol | Input | Remarks |
|---|---|---|
| #S | abdcdc$# | $S \rightarrow AC\$$ |
| #$CA | abdcdc$# | $A \rightarrow aBCd$ |
| #$CdCBa | abdcdc$# | Pop a |
| #$CdCB | bdcdc$# | $B \rightarrow bB$ |
| #$CdCBb | bdcdc$# | Pop b |
| #$CdCB | dcdc$# | $B \rightarrow d$ |
| #$CdCd | dcdc$# | Pop d |
| #$CdC | cdc$# | $C \rightarrow c$ |
| #$Cdc | cdc$# | Pop C |
| #$Cd | dc$# | Pop d |
| #$C | c$# | $C \rightarrow c$ |
| #$c | c$# | Pop c |
| #$ | $# | Pop $ |
| # | # | Accepted |