# UNIT – II

## Syntax Analysis

This is the 2 nd phase of the compiler, checks the syntax and construct the syntax/parse tree.

Input of parser is tokens and output is a parse/ syntax tree.

**Constructing Parse Tree:** Construction of derivation tree for a given input string by using the production of grammar is called parse tree.

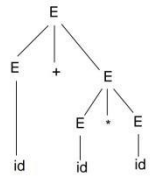
Consider the grammar

$E \rightarrow E + E|E * E$

$E \rightarrow id$

The parse tree for the string

$\omega = id + id * id$ is



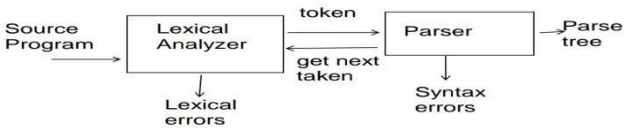

- **Role of the Parser**

1. Constructs parse tree.

2. Error reporting and correcting (or) recovery

A parser can be modeled by using CFG (Context Free Grammar) recognized by using pushdown automata/table driven parser.

3. CFG will only check the correctness of sentence with respect to grammar / syntax, but it doesn't check the meaning of the sentence.




## Construction of a Parse Tree

Parse tree can be constructed in two ways.

(i)**Top-down parser:** It derives the string (parse tree) from the root (top) to the children.

(ii)**Bottom-up parser:** It derives the string from the children and works up to the root. In both cases, the input is scanned from left to right, one symbol at a time.

**Parser Generator:** Parser generator is a tool which creates a parser.

**Example:** compiler – compiler, YACC

The input of these parser generator is grammar we use, and the output will be the parser code.

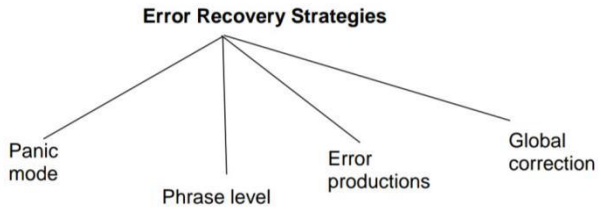
The parser generator is used for construction of the compilers front end.

**Syntax Error Handling**

(a)Reports the presence of errors clearly and accurately.

(b)recovers from each error quickly.

(c)It should not slow down the processing of correct programs.




**Panic Mode:** on discovering an error, the parser discards input symbols one at a time until one of the synchronizing tokens is found.

**Phrase Level:** A parser may perform local correction on the remaining input. It may replace the prefix of the remaining input.

**Error Productions:** Parser can generate appropriate error messages to indicate the erroneous construct that has been recognized in the input.

**Global Corrections:** There are algorithms for choosing a minimal sequence of changes to obtain a globally least cost correction.

- **Context free Grammars and Ambiguity**

A grammar is a set of rules or productions which generates a collection of finite/infinite strings.

It is a 4-tuple defined as G = (V, T, P, S)

Where V = set of variables / Non-Terminals

T = set of terminals

P = set of productions / rules

S = start symbol

**Ex:** S → (S)/e

S → (S) --------- (i)

S → e ---------- (ii)

Here S is start symbol and the only variable.

( , ), e are terminals.

(i)and (ii) are production rules.

**Sentential Forms:** s =>α, Where α may contain non-terminals, then we say that α is a sentential form of G.

**Sentence:** A sentence is a sentential form with no non terminal.

**Ex:** – (id + id) is a sentence of the grammar

E → E + E | – E | (E) |id.

**Left Most derivation (LMD):** In a derivation of a string if only left most non terminals are replaced, then it is a left most derivation.

**Right Most Derivation (RMD):** In a derivation of a string, if only rightmost non terminals are replaced, then it is a right most derivation.

**Ex:** Consider a grammar

E → E + E |E * E | – E |(E) |id.

The LMD and RMD for the string – (id + id) is ?

**Left Most Derivation**

E => – E => – €

=> – (E + E)

=> – (id + E)

=> – (id + id)

**Right Most Derivation**

E => – E => – (E)

=> – (E + E)

=> – (E + id)

=> – (id + id)

****Right most derivations are also known as canonical derivations.**

**Ambigous Grammar:** A grammar 'G' is said to be ambigous if there exists more than one derivation tree for the given input string. (OR)

A grammar that produces more than one left most or more than one right most derivations is ambiguous.

For example consider the following grammar:

String → String + String|String – String

|0|1|2|...|9 9 – 5 + 2 has two parse trees as shown below

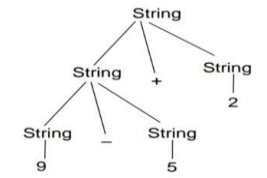



**Fig: Leftmost Derivation Tree**

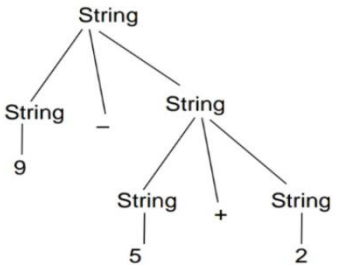



**Fig: Rightmost Derivation Tree**

• Ambiguity is problematic because the meaning of the program can be incorrect.

• Ambiguity of CFG is undecidable.

• Ambiguity can be handled in several ways

**Removal of Ambiguity:** The ambiguity of grammar is undecidable, ambiguity of a grammar can be eliminated by rewriting the grammar.

**Example:**E → E + E |E * E |id

The above grammar is ambiguous. The ambiguous grammar doesn't follow associativity and precedence rule.

Let '+' is Right associative

and '*' is Left associative.

Unambiguous grammar

E → E + T|T * E |id

Let the precedence of '*' is higher than '+'.

The operator whose precedence is higher should appear at lower level.

E → E + T | T

T → T * F | F

F → id

The above grammar is unambiguous grammar.

• **Left Recursion:** In a grammar, if left most variable of Right hand side is same as the variable at left hand side then it is called left recursion. Left recursion can take the parser into infinite loop so we need to remove left recursion.

**Elimination of Left Recursion**

A → Aα|β is a left recursive.

It needs to converted into equivalent right recursive grammar.

$A \rightarrow \beta A^1$

$A\ 1 \rightarrow \alpha A^{1|}\epsilon$

In general

$A \rightarrow A\alpha_1 / A\alpha_2 /... / A\alpha_m /\beta_1 /\beta_2 /.../\beta_n$

We can replace A productions by

$A \rightarrow \beta_1 A^1/\beta_2 A^1/---/\beta_n A^1$

$A^1 \rightarrow \alpha_1 A^1/\alpha_2 A^1/--/\alpha_m A^1/\epsilon$

**Examples:**

**Eg 1.** Eliminate left recursion from

$E \rightarrow E + T/T$

$T \rightarrow T * F/F$

$F \rightarrow (E) /id$

**Sol.** $E \rightarrow E + T/T$ it is in the form

$A \rightarrow A\alpha/\beta$

So, we can write it as $E \rightarrow TE^1$

$E^1 \rightarrow +TE^1 /\epsilon$

Similarly, other productions are written as

$T \rightarrow FT^1$

$T^1 \rightarrow * FT^1 / \epsilon$

$F \rightarrow (E)/id$

**Eg 2.** Eliminate left recursion from the grammar

$S \rightarrow (L) /a$

$L \rightarrow LS/b$

**Sol.** $S \rightarrow (L) /a$

$L \rightarrow bL^I$

$L^I \rightarrow SL^I / \epsilon$

**Left Factoring:** A grammar with common prefixes is called non-deterministic grammar.

To make it deterministic we need to remove common prefixes. This process is called as Left Factoring.

The grammar: $A \rightarrow \alpha\beta^1 /\alpha\beta^2$ has common prefixes ($\alpha$) it is transformed into

$A \rightarrow \alpha A^1$

$A^1 \rightarrow \beta_1 / \beta_2$

**Eg 1.**What is the resultant grammar after left factoring the following grammar?

S → iEtS/iEtSeS/a

E → b

**Sol.** S → iEtSS $^1$ /a

S 1 → eS/ ϵ

E → b

**Non Recursive Descent Parser**

**Example:** (table driven parsing)

• It maintains a stack explicitly, rather thanimplicitly via recursive calls.

• A table driven predictive parser has

1.an input buffer

2.a stack

3.a parsing table

4.output stream

**Recursive Descent Parser**

**Predictive Parsers**

By eliminating left recursion and by left factoring the grammar, we can have parse tree without backtracking. To construct a predictive parser, we must know,

(a) current input symbol

(b) non terminal which is to be expanded

A procedure is associated with each non terminal of the grammar.

**Recursive Descent Parsing:** In recursive descent parsing, we execute a set of recursive procedures to process the input.

The sequence of procedures called implicitly, defines a parse tree for the input

Construction of LL(1) Praising Table

- **Construction of LL(1) Parsing Table**

**Constructing a Parsing Table:** To construct a parsing table, we have to learn about two functions:

(a) FIRST( )

(b) FOLLOW( )

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

1. If X is a terminal, then FIRST(X) is {X}.

2. If X is non terminal and X → AB

where A →ϵ then

FIRST(X) = FIRST(A)

where A → ϵ then

FIRST(X) = {FIRST(A) – ϵ } U FIRST (B)

$X \rightarrow Y_1 Y_2 Y_3 \ldots\ldots Y_k$

ϵ is present in all $Y_1 Y_2 \ldots\ldots Y_k$ then Add ' ϵ ' to FIRST (X).

3. If X →ϵ is a production, then add ϵ to FIRST(X).

**Eg 1.** Consider the grammar

S → X Y|ϵ

X → a X b|c

Y → Y X |ϵ

Find the first (S), first (X), first (Y)

**Sol.** First (S) = First (X) U {ϵ }

= {a, c, ϵ }

First (X)= first (a X b) U first (c)

= {a} U{c}

= {a, c}

First (Y) = first (YX) U{ϵ }

= first (X) U{ϵ }

= {a, c} U{ϵ }

= {a, c, ϵ }

**Eg 2.** Consider the grammar

S → ABC

A → a|ϵ

B → b|ϵ

C → c

Find the first of S, A, B, C.

**Sol.**

First (A) = {a, ϵ}

First (B) = {b, ϵ}

First (C) = {c}

First (S) = First (ABC)

= {First (A) – {ϵ}} U First (BC)

= {a} U First (B) – {ϵ} U First ©

= {a} U {b} U {c}

= {a, b c}

**Eg.3.** Consider the grammar

S → ABcD

A → a|ϵ

B → b|ϵ

D → d|ϵ

Find the first of S, A, B, D

**Sol.** First (S) = First (ABCD)

= {First (A) – {ϵ}} U First (BcD)

= {a} U{First (B) – {ϵ} U First (cD)

= {a} U {b} U{c}

= {a, b, c}

First (A) = {a, ϵ}

First (B) = {b, ϵ}

First (D) = {d, ϵ}

**Eg.4 .** Consider the grammar

S→ X Y Z W

X → x|ϵ

Y → y|ϵ

Z → z|ϵ

W →w|ϵ

Find the first of S, X, Y, Z, W?

**Sol.** First (S) = First (XYZW)

= {First (X) – {ϵ}} U First (YZW)

= {x} U {First (Y) – ϵ } U First (ZW)

= {x} U {Y} U {First (Z) – {ϵ}} U First (W)

= {x} U {Y} U {Z} U {W, ϵ}

= {x,y,z,w,ϵ}

First (X) = {x, ϵ} First (Z) = {z, ϵ}

First (Y) = {y, ϵ} First (W) = {w, ϵ}

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

1. Place $ in FOLLOW(S), where S is the start symbol and $ is input right end marker.

2. If there is a production A →αBβ, then everything in FIRST(β) except ϵ is placed in FOLLOW(B).

3. If there is a production A → αB or a production A →αBβ, where FIRST (β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW (B).

**Eg.5**. Consider the grammar

$E \rightarrow TE^1$

$E\,1 \rightarrow +TE^1 \mid \epsilon$

$T \rightarrow FT^1$

$T\,1 \rightarrow *FT^1 \mid \epsilon$

F → (E)/id. Then find the follow of E, T, F,$T^1$ , $E^1$ .

**Sol.** FIRST (E) = FIRST (T) = FIRST (F)

= {(, id}

FIRST ($E^1$ ) = {+,ϵ}

FIRST ($T^1$ ) = {*, ϵ}

FOLLOW (E) = FOLLOW ($E^1$ ) = {), $}

FOLLOW(T) = FOLLOW($T^I$) = {FIRST ($E^1$ ) – ϵ} U FOLLOW (E) = { +, ), $}

FOLLOW (F) = {FIRST ($T^1$ ) –ϵ} U FOLLOW (T) = {*, +, ), $}

**Eg.6.** Consider the grammar

S → Xx Xy|Yx Yy

X →ϵ

Y → ϵ

Find the follow of all Non – terminals

**Sol.** Follow (X) = First (x) U First (y)

= {x} U {y}

= {x, y}

Follow (Y) = First (x) U First (y)

= {x, y}

Follow (S) = {$}

**Eg.7**. Consider the grammar

$S \rightarrow XYZ$

$X \rightarrow x|\epsilon$

$Y \rightarrow y|\epsilon$

$Z \rightarrow z$

Find the follow of all Non – Terminals.

**Sol.** Follow (S) = {$}

Follow (X) = first (YZ)

= {first (Y) – {ϵ}} U first (Z)

= {y} U {z}

= {y, z}

Follow (Y) = first (Z)

= {z}

Follow (Z) = Follow (S) = {$}

**Steps for the Construction of Predictive Parsing Table**

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.

2. For each terminal a in FIRST (α), add $A \rightarrow \alpha$ to M[A, a]

3. If ϵ is in FIRST (α), add $A \rightarrow \alpha$ to M[A, b] for each terminal b in FOLLOW (A). If ϵ is in FIRST(α) and $ is in FOLLOW (A), add $A \rightarrow \alpha$ to M[A, $]

4. Make each undefined entry of M be error.

By applying these rules to the above grammar, we will get the following parsing table.

| **Non Terminal** | **Input Symbol** | | | | | |
|---|---|---|---|---|---|---|
| | **id** | **+** | ***** | **(** | **)** | **$** |
| E | $E \rightarrow TE^1$ | | | $E \rightarrow TE^1$ | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| $E^1$ | | E1→+ $TE^1$ | | | $E^1 \rightarrow \epsilon$ | $E^1 \rightarrow \epsilon$ |
| T | T→F $T^1$ | | | T→FT 1 | | |
| $T^1$ | | $T^1 \rightarrow \epsilon$ | T1→* $FT^1$ | | $T^1 \rightarrow \epsilon$ | $T^1 \rightarrow \epsilon$ |
| F | F→id | | | F→(E ) | | |

The parser is controlled by a program. The program consider x, the symbol on top of the stack and 'a' the current input symbol.

1. If x = a = $, the parser halts and announces successful completion of parsing.

2. If x = a ≠ $, the parser pops x off the stack and advances the input pointer to the next input symbol.

3. If x is a non terminal, the program consults entry M[x, a] of the parsing table M. This entry will be either an x-production of the grammar or an error entry. If M[x, a] = {x→ X Y Z}, the parser replaces x on top of the stack by Z Y X with X on the top.

If M[x, a] = error, the parser calls an error recovery routine.

For example, consider the moves made by predictive parser on input **id + id *id**, which are Shown below

| **Matched** | **Stack** | **Input** | **Action** |
|---|---|---|---|
| | E$ | id + id * id$ | |
| | $TE^1$$ | id + id * id$ | Output $E \rightarrow TE^1$ |
| | $FT^1E^1$$ | id + id * id$ | Output $T \rightarrow FT^1$ |
| | $idT^1E^1$ $ | id + id * id$ | Output F→id |
| id | $T^1E^1$$ | + id * id$ | Match id |
| id | $E^1$$ | + id * id$ | Output $T^1 \rightarrow \epsilon$ |
| id | $+TE^1$$ | + id * id$ | Output $E^1 \rightarrow +TE^1$ |
| id + | $TE^1$$ | id * id$ | Match + |
| id + | $FT^1E^1$$ | id * id$ | Output $T \rightarrow FT^1$ |

| | | | |
|---|---|---|---|
| id + | $idT^1E^1$ $ | id * id$ | Output F→id |
| id + id | $T^1E^1$ | * id$ | Match id |
| id + id | $*FT^1E^1$ $ | * id$ | Output $T^1 \rightarrow *FT^1$ |
| id + id * | $FT^1E^1$ | id$ | Match * |
| id + id * | $idT^1E^1$ $ | id$ | Output F→id |
| id + id * id | $T^1E^1$ | $ | Match id |
| id + id * id | $E^1$ | $ | Output $T^1 \rightarrow \epsilon$ |
| id + id * id | $ | $ | Output $E^1 \rightarrow \epsilon$ |

**Eg.8.** What will be the entries in predictive parsing table for given grammar?

$S \rightarrow iBfSS^I / d$

$S\ I \rightarrow eS / \epsilon$

$B \rightarrow b$

**Sol.** First (S) = {i, d}

First ($S^I$) = {e, ϵ}

First (B) = {b}

Follow (S) = {$} U {First ($S^I$) – ϵ}

= {$, e}

Follow ($S^I$) = Follow(S) = {$, e}

Follow (B) = {f}

**Predictive Parsing table**

| | d | b | e | i | f | $ |
|---|---|---|---|---|---|---|
| S | S→d | | | S→iBf $SS^1$ | | |
| $S^1$ | | | $S^1 \rightarrow \epsilon$ <br> S→eS | | | S1→ ϵ |
| B | | B→b | | | | |

The given grammar has multiple entries in [$S^I$ , e] therefore given grammar is not LL (1).