(KTU - 2019 Scheme)

COMPILER DESIGN

MODULE - II

Introduction to Syntax Analysis.

SYLLABUS :

Role of syntax analyzer - Syntax error handling. Review of context free Grammars - Derivation and parse trees, Eliminating Ambiguity. Basic parsing approaches - Eliminating Left Recursion, Left factoring. Top-Down parsing - Recursive Descent Parsing, Predictive parsing, LL(1) Grammars

Role of Syntax Analyzer :

- Every programming languages has precise rules to describe its syntactic structure.
- The syntax of pgming lang. constructs can be specified by context-free grammars (CFG) or BNF (Backus-Naur Form) notation.
- The parser obtains a string of tokens from the lexical analyzer and verifies that this string of tokens can be generated by the grammar for the soure language.
- The parser also reports any syntax errors to the user and it also recover from the commonly occuring errors to continue processing the remainder of the program.
- For well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing.
- Note : CFG are recognized by push-down automata

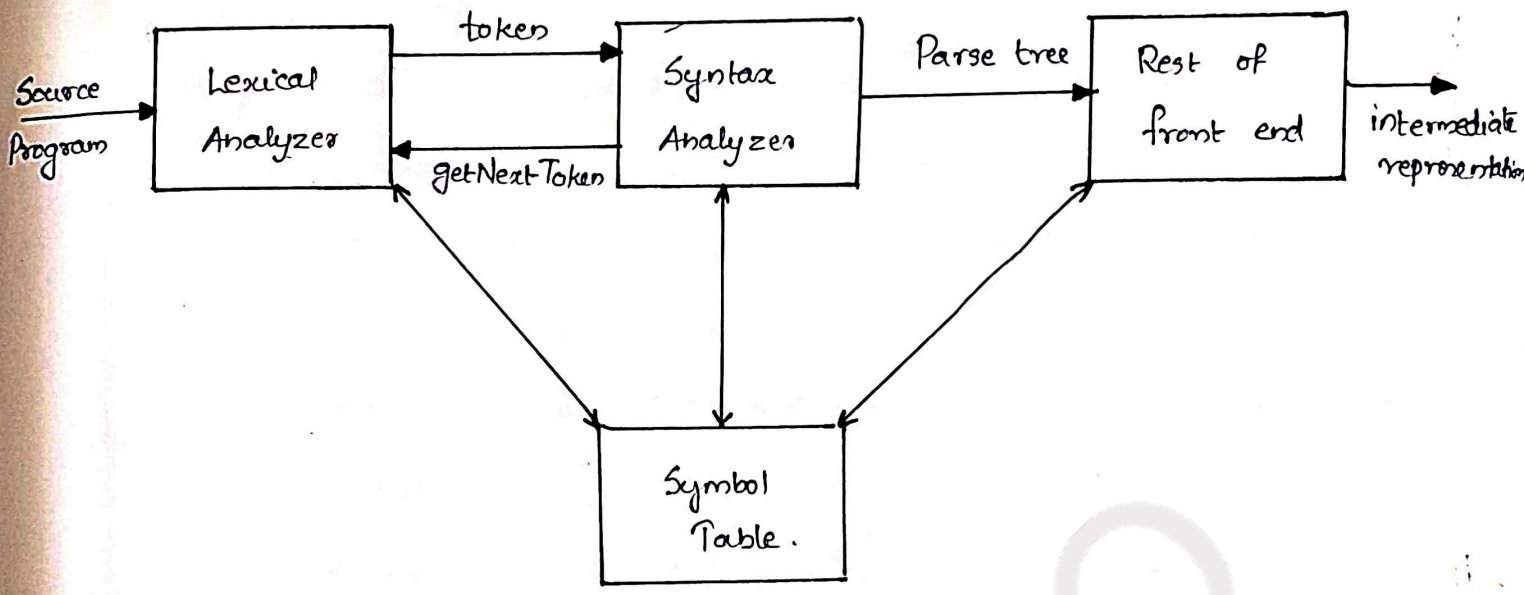


fig: position of parser in compiler model
(Syntax analyzer)

There are 3 types of parsers:

① Universal parser

② Top-down parser

③ Bottom-up parser.

- universal parsing methods such as CYK algorithm and Earley's algorithm can parse any grammars. But these methods are inefficient to use.
- Thus the commonly used parsing methods are: top-down and bottom-up approaches.
- Top-down methods build parse trees from the top (ie root) to the bottom (ie leaves), while bottom-up methods start from the leaves and work their way up to the root. In both the cases, the i/p to the parser is scanned from left-to-right, one symbol at a time.

Syntax Error Handling :

The goals of an error handler in a parser are :

- To report the presence of errors clearly & accurately.
- Recover from each error quickly to detect subsequent errors.
- Add minimal overhead to the processing of correct programs.

- Here, the aim is to detect the syntax errors.

Error - Recovery strategies :-

1. Panic - mode recovery
2. Phrase - level recovery
3. error - productions
4. Global corrections.

1) Panic - mode Recovery :-

In this method, ~~on~~ when the parser discovers an error, it discards the input symbols one at a time, until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as semicolon or }, whose role in the source pgm is clear & unambiguous.

Disadvantage : Panic-mode correction often skips a considerable amount of input without checking it for additional errors.

Advantages: - Simplicity.
- It is guaranteed not to go into an infinite loop

2) Phrase-level Recovery :

- On discovering an error, the parser may perform local correction on the remaining input; that is it may replace a prefix of the remaining input by some string that allows the parser to continue.
- Typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.
- The choice of the local correction is left to the compiler designer.
- we must be careful to choose replacements that do not lead to infinite loops.
- This method has been used in several error-repairing compilers, as it can correct any input string.
- Drawback :- Difficulty in coping with situations in which the actual error has occurred before the point of detection.

3) Error-productions :

- We can augment the grammar for the language with productions that generate the common erroneous constructs.
- Such a parser can detect the anticipated errors when an error production is used during parsing.
- The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input.

4) ~~Error~~ Global Correction :

- Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y, such that the no: of insertions, deletions and changes of tokens required to transform x into y is as small as possible.
- That means, these algorithms will choose a minimal sequence of changes to obtain a globally least-cost correction.
- Unfortunately, these methods are too costly to implement in terms of time and space, so these techniques are currently of theoretical interest only (not practical).

Context-Free Grammars : (CFG)

A context-free grammar, CFG, consists of terminals (T_o), variables or the non-terminals (V), productions (P) and a start symbol (S). ie,

$$G = (V, T, P, S).$$

- Terminals : Basic symbols from which strings are formed.
- Non-terminals : variables that denote set of strings. They impose a hierarchical structure on the language that is key to syntax analysis and translation.
- In a grammar, one non-terminal is distinguished as the start symbol, and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.
- The productions of a grammar specifies the manner in which the terminals and non-terminals can be combined to form strings.

Each production consists of:

(a) A non-terminal called the head or left side of the production.

(b) The symbol '$\longrightarrow$'. Sometimes ::= has been used in place of the arrow.

(c) The body or right side consisting of zero or more terminals and non-terminals. The components of the body describe one way in which strings of the non-terminal at the head can be constructed.

Derivations and Parse-trees:

Derivations: Beginning with the start symbol, each rewriting step replaces a non-terminal by the body of one of its productions. This corresponds to the top-down construction of a parse tree.

For example, consider the below grammar:

$$E \longrightarrow E+E \mid E*E \mid -E \mid (E) \mid id$$

The string $-(id+id)$ is a sentence of the above grammar because we can derive this string as given below.

$$\begin{aligned} E &\longrightarrow -E \\ &\longrightarrow -(E) \\ &\longrightarrow -(E+E) \\ &\longrightarrow -(id+E) \\ &\longrightarrow -(id+id) \end{aligned}$$

- This is the left-most derivation.

The same sentence $-(id+id)$ can also be derived using the given below right-most derivation :

$$E \longrightarrow -E$$
$$\rightarrow -(E)$$
$$\rightarrow -(E+E)$$
$$\rightarrow -(E+id)$$
$$\rightarrow -(id+id)$$

Thus, the derivations are of two types :

① Left-most derivation (LMD) : the left-most non-terminal in each sentential is always chosen. This can be denoted using 'lm'. ie, $\alpha \xrightarrow[lm]{} \beta$.

② Right-most derivation (RMD) : The right most non-terminal is always chosen. In this case, we write it as, $\alpha \xrightarrow[rm]{} \beta$.

Eg: $S \rightarrow AB$; $A \rightarrow aaA \mid \lambda$; $B \rightarrow Bb \mid \lambda$ ('λ' means 'ε')

Derive the string 'aab' using both LMD & RMD.

Soln: Left-most derivation :-

$$S \rightarrow AB$$
$$\rightarrow aaAB$$
$$\rightarrow aaB$$
$$\rightarrow aaBb$$
$$\rightarrow aab$$

This requires 5 steps.

Right-most derivation :

$S \rightarrow AB$

$\rightarrow ABb$

$\rightarrow Ab$

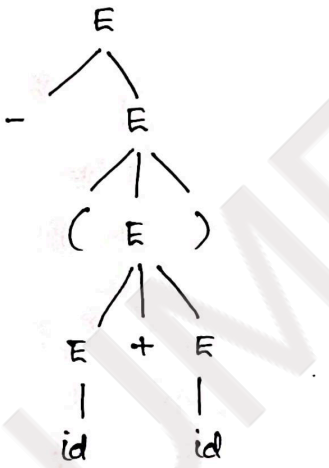
$\rightarrow aaAb$

$\rightarrow aab$.

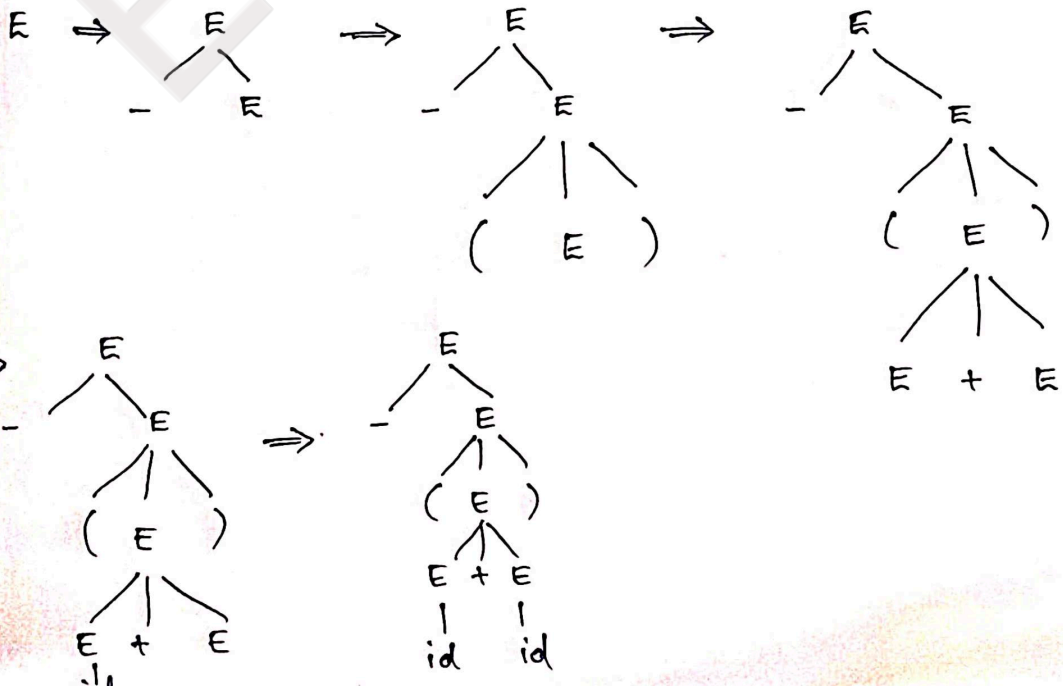
Parse Trees :

Parse tree is a graphical representation of a derivation.

eg: For the previous example, $-(id+id)$, the parse tree is,



The sequence for generating this parse tree is, (in case of LMD),



Ambiguity :

A grammar that produces more than one parse tree for some sentence is called ambiguous grammar. That means, an ambiguous grammar is one that produces more than one left-most derivation or more than one right-most derivation for the same sentence.

eg: The below grammar,

$$E \rightarrow E+E \mid E*E \mid (E) \mid id$$

permits two distinct LMDs for the sentence id + id * id.

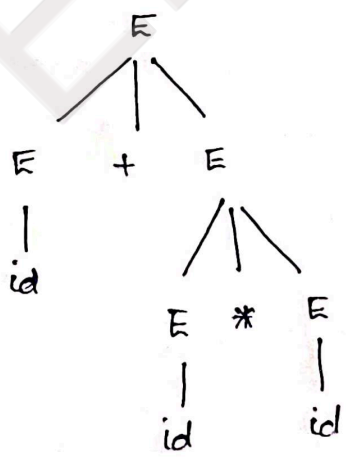
ie,

$E \rightarrow E+E$
$\rightarrow id + E$
$\rightarrow id + E*E$
$\rightarrow id + id * E$
$\rightarrow id + id * id$

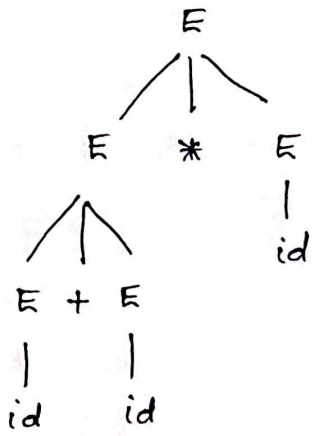
$E \rightarrow E * E$
$\rightarrow E+E * E$
$\rightarrow id + E * E$
$\rightarrow id + id * E$
$\rightarrow id + id * id$

Thus, the above grammar is considered to be ambiguous.

The corresponding two parse trees are:



and



Elimination of Left-Recursion:

- A grammar is left-recursive if it has a non-terminal 'A' such that there is a derivation $A \xrightarrow{+} A\alpha$ for some string 'α'.
- ie, A grammar is left-recursive if it is of the form,

$$A \longrightarrow A\alpha \mid \beta$$

- Then, this grammar can be ~~made~~ converted into,

$$A \longrightarrow \beta A'$$
$$A' \longrightarrow \alpha A' \mid \varepsilon$$

[Note: A' can be any non-terminal other than the ones present in the given grammar]

Now the productions are non-left-recursive productions.

- This also works for any number of A-productions.

ie, Suppose the grammar is,

$$A \longrightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n ,$$

Then we can replace the A production as,

$$A \longrightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$
$$A' \longrightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon .$$

Note:- A grammar of the form $S \rightarrow \alpha S \mid \beta$ is said to be Right recursive. But right recursion does not create any problem for Top-down parsers. Hence, there is no need of eliminating right recursion.

Examples :-

$B \longrightarrow Bc \mid b$

Soln:

$\alpha = c$ and $\beta = b$.

$\Rightarrow$ $B \longrightarrow bB'$

$B' \longrightarrow cB' \mid \varepsilon$.

Q) Eliminate left-recursion.

1. $A \longrightarrow ABd \mid Aa \mid a$

$B \longrightarrow b \mid \varepsilon$.

Soln: $A \longrightarrow aA'$

$A' \longrightarrow BdA' \mid aA' \mid \varepsilon$

$B \longrightarrow b \mid \varepsilon$.

2. $E \longrightarrow E+E \mid E*E \mid a$

Soln: $E \longrightarrow aE'$

$E' \longrightarrow +EE' \mid *EE' \mid \varepsilon$

3. $E \longrightarrow E+T \mid T$

$T \longrightarrow T*F \mid F$

$F \longrightarrow id$

Soln: $E \longrightarrow TE'$

$E' \longrightarrow +TE' \mid \varepsilon$

$T \longrightarrow FT'$

$T' \longrightarrow *FT' \mid \varepsilon$

$F \longrightarrow id$

4. $A \rightarrow Ac \mid Aad \mid bd \mid cd$

Soln: $A \rightarrow bdA' \mid cdA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

H:W 5. $S \rightarrow A$

$A \rightarrow Ad \mid Ae \mid aB \mid ac$

$B \rightarrow bBc \mid f$

Soln: $S \rightarrow A$.

$A \rightarrow aBA' \mid acA'$

$A' \rightarrow dA' \mid eA' \mid \varepsilon$

$B \rightarrow bBc \mid f$.

Examples of Indirect-left recursion:-

1. $A \rightarrow Ba \mid Aa \mid c$

$B \rightarrow Bb \mid Ab \mid d$.

Soln: In this example, let's consider the production,

$B \rightarrow Ab$.

This seems to be non-left recursive.

But if we replace 'A' by its production $A \rightarrow Ba$, then the above B-production becomes,

$B \rightarrow Bab$.

Now this production becomes left-recursive. Hence in such grammars, we have to do one extra step to eliminate left-recursion.

Thus, the soln is:

Step 1: Eliminate left-recursions from A-productions.

ie, $A \rightarrow Ba \mid Aa \mid c$

becomes,

$A \rightarrow BaA' \mid cA'$

$A' \rightarrow aA' \mid \varepsilon$.

Now, the given grammar becomes,

$A \rightarrow BaA' \mid cA'$

$A' \rightarrow aA' \mid \varepsilon$

$B \rightarrow Bb \mid Ab \mid d$

Step 2:

Substituting the productions of A, in $B \rightarrow Ab$, we get,

$A \rightarrow BaA' \mid cA'$

$A' \rightarrow aA' \mid \varepsilon$

$B \rightarrow Bb \mid BaA'b \mid cA'b \mid d$

Step 3:

Now eliminate the left-recursions from the B-productions.

ie, $A \rightarrow BaA' \mid cA'$

$A' \rightarrow aA' \mid \varepsilon$

$B \rightarrow cA'bB' \mid dB'$

$B' \rightarrow bB' \mid aA'bB' \mid \varepsilon$

This is the final grammar after eliminating left-recursion.

2) $X \longrightarrow XSb \mid Sa \mid b$

$S \longrightarrow Sb \mid Xa \mid a$.

Soln: The above grammar includes indirect left-recursion.

Step 1: Eliminate left-recursion from X.

$X \longrightarrow SaX' \mid bX'$

$X' \longrightarrow SbX' \mid \varepsilon$

$S \longrightarrow Sb \mid Xa \mid a$

Step 2: Substitute X in $S \longrightarrow Xa$.

$X \longrightarrow SaX' \mid bX'$

$X' \longrightarrow SbX' \mid \varepsilon$

$S \longrightarrow Sb \mid SaX'a \mid bX'a \mid a$.

Step 3: Eliminate left-recursion from S.

$X \longrightarrow SaX' \mid bX'$

$X' \longrightarrow SbX' \mid \varepsilon$

$S \longrightarrow bX'aS' \mid aS'$

$S' \longrightarrow bS' \mid aX'aS' \mid \varepsilon$

H.W

3) $S \longrightarrow Aa \mid b$

$A \longrightarrow Ac \mid Sd \mid \varepsilon$

Soln: Step 1: Remove left recursion from S.

~~$S \longrightarrow Aa \mid b$~~ ~~$A \longrightarrow Sd A' \mid$~~ But S is already free from left-recursion.

Step 2: Now substitute S in $A \longrightarrow Sd$.

$S \longrightarrow Aa \mid b$.

$A \longrightarrow Ac \mid Aad \mid bd \mid \varepsilon$

Step 3:

$S \longrightarrow Aa \mid b$.

$A \longrightarrow bdA' \mid A'$

$A' \longrightarrow cA' \mid adA' \mid \varepsilon$

Elimination of Left-factoring :

- A grammar contains left-factor if it is of the form :

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$$

- To eliminate left-factoring, we have to convert the above grammar into the following form :

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

eg: $S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b.$

Here, the S-production contains left-factor. ie, the 'iEtS' (ie, common prefix) is common. Hence we convert it to,

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \varepsilon$
$E \rightarrow b.$

Examples :

Q1) $A \rightarrow aB \mid aC$

Soln: $A \rightarrow aA'$
$A' \rightarrow B \mid C$

Q2) $S \rightarrow SaB \mid Sac \mid ab$

Soln: $S \rightarrow SaS' \mid ab$
$S' \rightarrow B \mid c$

Q3) $S \rightarrow aAd \mid aB$

$A \rightarrow a \mid ab$

$B \rightarrow ccd \mid ddc$

Soln: $S \rightarrow aS'$

$S' \rightarrow Ad \mid B$

$A \rightarrow aA'$

$A' \rightarrow b \mid \varepsilon$

$B \rightarrow ccd \mid ddc$

Q4) $A \rightarrow aAB \mid aA \mid a$

$B \rightarrow bB \mid b$.

Soln. $A \rightarrow aA'$

$A' \rightarrow AB \mid A \mid \varepsilon$ → Here again left factoring, $A' \rightarrow AS \mid \varepsilon$, $S \rightarrow B \mid \varepsilon$.

$B \rightarrow bB'$

$B' \rightarrow B \mid \varepsilon$.

Q5) $A \rightarrow aAB \mid aBc \mid aAc$

Soln: $A \rightarrow aA'$

$A' \rightarrow AB \mid Bc \mid Ac$

Again, this is a grammar with common prefixes.

$A \rightarrow aA'$

$A' \rightarrow AS \mid Bc$

$S \rightarrow B \mid c$

Q6) $S \rightarrow assbs \mid aSasb \mid abb \mid b$

Soln: $S \rightarrow as' \mid b$.

$S' \rightarrow SSbS \mid SaSb \mid bb$

Again left factoring,

$S \rightarrow aS' | b$

$S' \rightarrow SA | bb.$

$A \rightarrow SbS | aSb.$

Q.7) $S \rightarrow a | ab | abc | abcd.$

Soln: $S \rightarrow aS'$

$S' \rightarrow b | bc | bcd | \varepsilon.$

Again taking the common prefix,

$S \rightarrow aS'$

$S' \rightarrow bA | \varepsilon$

$A \rightarrow c | cd | \varepsilon$

Again repeating,

$S \rightarrow aS'$

$S' \rightarrow bA | \varepsilon$

$A \rightarrow cB | \varepsilon$

$B \rightarrow d | \varepsilon.$

H.W.

Q8) $A \rightarrow abB | aB | cdg | cdeB | cdfB.$

Soln: $A \rightarrow aA' | cdg | cdeB | cdfB.$ [Again common prefix : 'cd']

Imp. $A' \rightarrow bB | B.$

$\Downarrow$

$A \rightarrow cA'' | aA'$

$A'' \rightarrow dg | deB | dfB$

$A' \rightarrow bB | B.$

$\Rightarrow$

$A \rightarrow cdA'' | aA'$

$A'' \rightarrow g | eB | fB$

$A' \rightarrow bB | B.$

TOP-DOWN PARSING :

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in pre-order. Equivalently, top-down parsing can be viewed as finding a left most derivation for an input string.

Eg: Construct top-down parse tree for the sentence id + id * id for the grammar,

$E \longrightarrow TE'$

$E' \longrightarrow +TE' \mid \varepsilon$

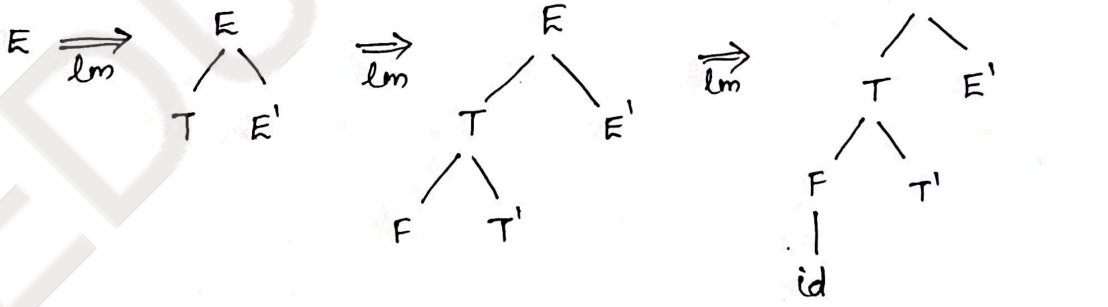
$T \longrightarrow FT'$

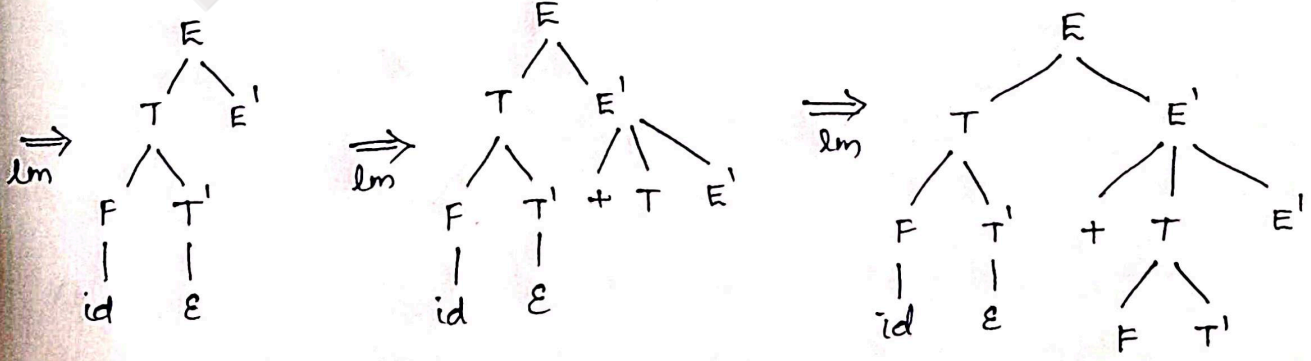
$T' \longrightarrow *FT' \mid \varepsilon$

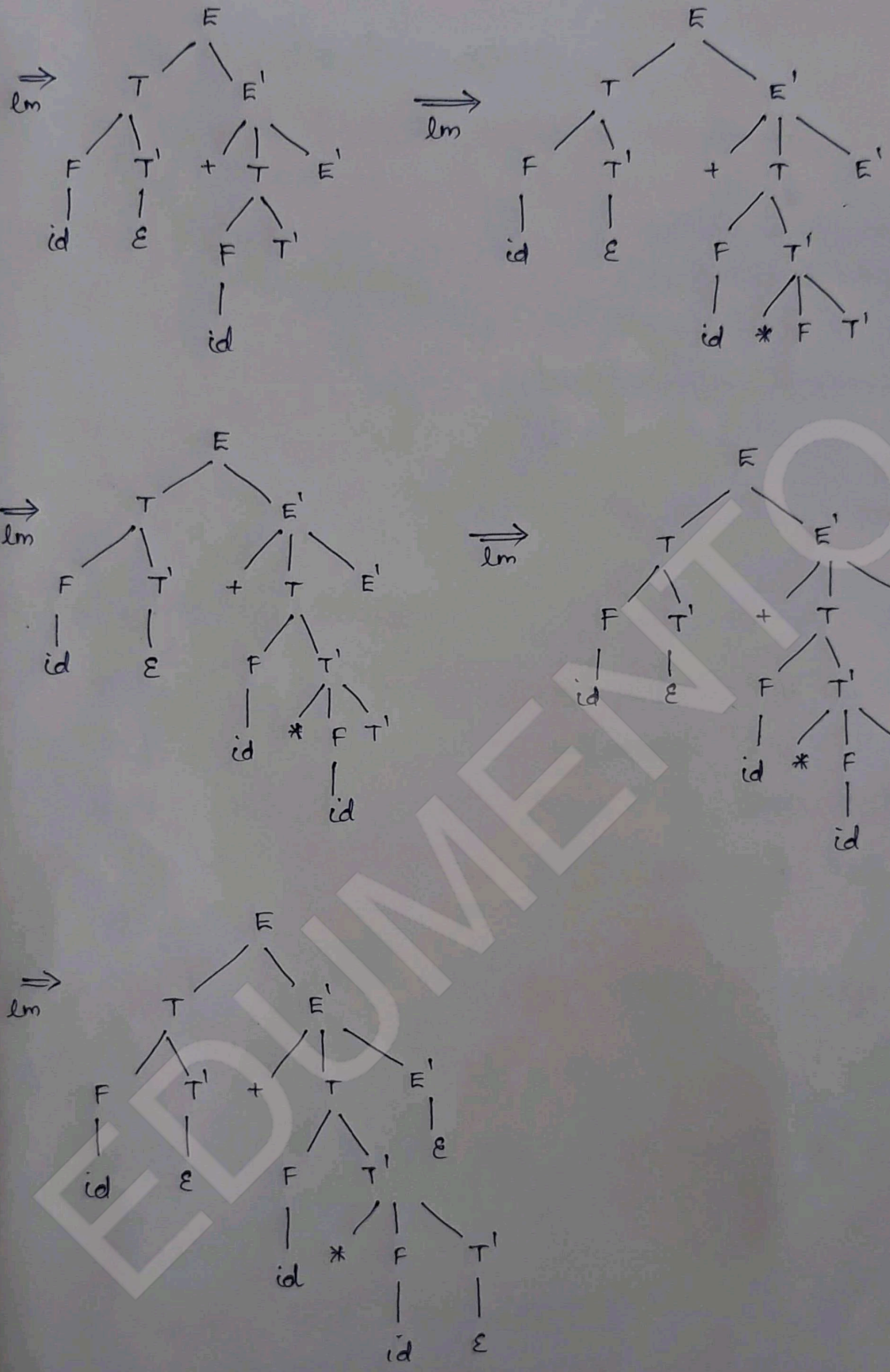
$F \longrightarrow (E) \mid id$

soln :

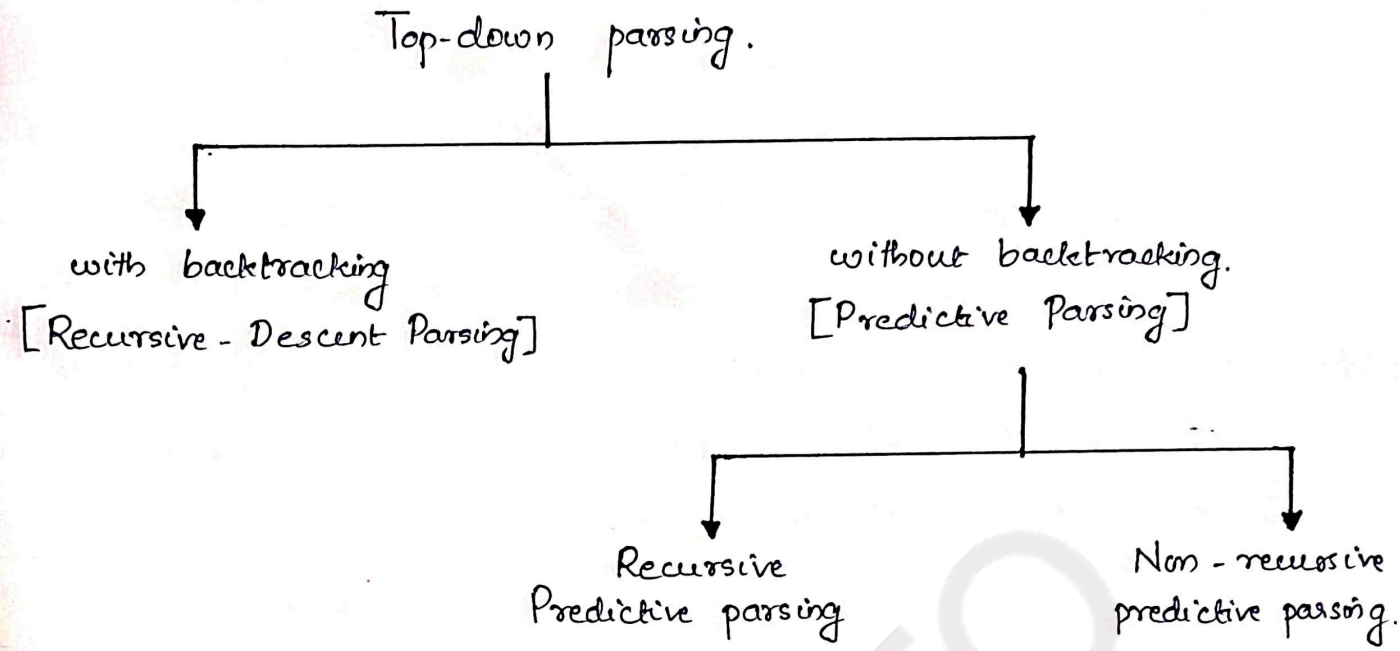
i.e







At each step of a top-down parse, the key problem is that of determining the production to be applied for a non-terminal. Either, it sometimes requires backtracking to find the correct ~~the~~ productions, or it needs to look-ahead the input a fixed number of symbols (typically one symbol).



Recursive - Descent Parsing :

- A recursive - descent parsing program consists of a set of procedures, one for each non-terminal.
- Execution begins with the procedure for start symbol, which halts and announces success if its procedure body scans the entire input string.
- Pseudocode for a typical non-terminal is given below.

```
void A()
  {
  Choose an A-production, A → X1 X2 ..... Xk ;
  for (i=1 to k)
  {
    if ( Xi is a non-terminal)
        call procedure Xi();
    else if (Xi equals the current i/p symbol a)
        advance the i/p to the next symbol;
    else /* an error has occured */ ;
  }
}
```

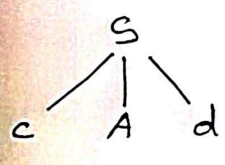
- General recursive-descent may require backtracking; that is, it may require repeated scans over the input. But, backtracking is not very efficient.
- To allow backtracking, the failure at the final 'else' portion of the above pseudocode is not ultimate failure, but suggests only that we need to return to the first line and try another A-production. Only if there are no more A-productions to try, we declare than an error has been found.

eg: Consider the grammar,

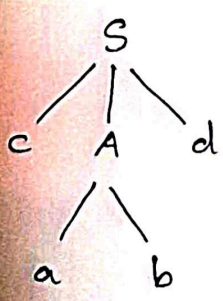
$$S \longrightarrow cAd$$
$$A \longrightarrow ab \mid a$$

construct a parse-tree for the i/p string, $w = cad$.

Sol^n:



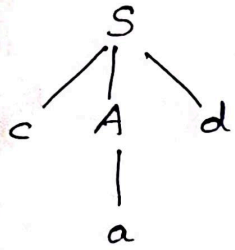
- i/p pointer will be first pointing to the symbol 'c' of w.
- Now the leftmost leaf, labeled 'c', matches the 1st input symbol of 'w'. So we advance the i/p pointer to 'a'.
- Now consider the next leaf, A. (Its a non-terminal). Hence expand A using the first production $A \longrightarrow ab$.



Now the leaf labeled 'a' matches the 2nd i/p symbol of w. Hence we advance the i/p pointer to the next symbol 'd', and combare it against the next leaf node 'b'. Since,

'b' does not match 'd', we report failure and backtrack to A to see whether there is another alternative for A-production that has not been tried, and produce a match.

- Thus we consider the A production $A \longrightarrow a$.



- Now the leaf 'a' matches the 2nd i/p symbol of w, and the leaf 'd' matches the 3rd i/p symbol. Since we have produced a parse tree for w, we halt and announce successful completion of parsing.

Note :- A left-recursive grammar can cause a recursive-descent parser (even the one with backtracking) to go into an infinite loop.

FIRST & FOLLOW :

- The construction of both top-down and bottom-up parsers is aided by two functions :
  1. FIRST
  2. FOLLOW
- During top-down parsing, FIRST & FOLLOW allow us to choose which production to apply, based on the next i/p symbol.
- During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

FIRST :-

Define FIRST(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α.

- If $\alpha \overset{*}{\Rightarrow} \varepsilon$, then ε is also in FIRST(α).
- To compute FIRST(X),
  1) If X is a terminal, then FIRST(X) = {X}
     ie FIRST(terminal) = {that terminal itself}.
  2) If $X \rightarrow \varepsilon$ is a production, then add ε to the FIRST(X).
  3) If X is a non-terminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a product" for some $k \geq 1$,
     - then place 'a' in FIRST(X) if for some i, 'a' is in FIRST(Y_i), and ε is in all of FIRST(Y_1),..., FIRST(Y_{i-1}); that is, $Y_1 \dots Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$.
     - If ε is in FIRST(Y_j) for all $j = 1, 2, \dots k$, then add ε to FIRST(X).

eg: Consider the grammar,

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$f \rightarrow (E) \mid id.$

Find the FIRST() of all the non-terminals.

Soln: FIRST(E) = FIRST(T) = FIRST(F) = { (, id }

FIRST(E') = {+, ε}

FIRST(T) = FIRST(F) = {(, id}

FIRST(T') = {*, ε}

FIRST(F) = {(, id}

Examples :-

Q1) $S \rightarrow abc \mid def \mid ghi$. find FIRST(S).

Soln: FIRST(S) = {a, d, g}

Q2) $S \rightarrow ABC \mid ghi \mid jkl$

$A \rightarrow a \mid b \mid c$

$B \rightarrow b$

$D \rightarrow d$

Soln: FIRST(D) = d

FIRST(B) = b.

FIRST(A) = {a, b, c}

FIRST(S) = {FIRST(A), g, j}

= {a, b, c, g, j}

Q3) $S \rightarrow ABC$

$A \rightarrow a \mid b \mid \varepsilon$

$B \rightarrow c \mid d \mid \varepsilon$

$C \rightarrow e \mid f \mid \varepsilon$

Soln: FIRST(C) = {e, f, ε}

FIRST(B) = {c, d, ε}

FIRST(A) = {a, b, ε}

FIRST(S) = FIRST(A)

• { a, b, FIRST(B)}

= { a, b, c, d, FIRST(c)}

= { a, b, c, d, e, f, ε }

Q4) S → a BDh

B → cC

C → bC | ε

D → EF

E → g | ε

F → f | ε

Soln: FIRST(F) = { f, ε }

FIRST(E) = { g, ε }

FIRST(D) = FIRST(E)

= { g, FIRST(F)}

= { g, f, ε }

FIRST(C) = { b, ε }

FIRST(B) = { c }

FIRST(S) = { a }

IMP

Q5) S → A

A → aB | Ad

B → b

C → g.

Soln: The above grammar is left-recursive. Hence first remove left recursion from the grammar and then find the FIRST( ).

Thus, the grammar changes to,

$S \longrightarrow A$

$A \longrightarrow a\,BA'$

$A' \longrightarrow d\,A' \mid \varepsilon$

$B \longrightarrow b$

$C \longrightarrow g$

Now find the FIRST().

FIRST(C) = {g}

FIRST(B) = {b}

FIRST(A') = {d, ε}

FIRST(A) = {a}

FIRST(S) = FIRST(A) = {a}

FOLLOW() :-

Follow(A), for a non-terminal A, is the set of terminals that can appear immediately to the right of A in some sentential form; that is, the set of terminals 'a' such that there exists a derivation of the form $S \overset{*}{\Rightarrow} \alpha A a \beta$, for some α and β.

Note: If 'A' is the rightmost symbol in some sequential form, then '$' is in FOLLOW(A); '$' is a special "endmarker" symbol that is assumed not to be a symbol of any grammar.

To compute FOLLOW(A), apply the following rules:

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

2) If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except 'ε' is in FOLLOW(B).

3) If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

eg:- Consider the grammar,

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

find the FOLLOW() of all non-terminals.

soln:- FOLLOW(E) = {), $ } . [Since E is the start symbol, we have to add $ to the FOLLOW(E)]

FOLLOW(E') = { FOLLOW(E) } // since E' is the rightmost symbol.

= {), $ }.

FOLLOW(T) = { FIRST(E') }

we know that FIRST(E') = { +, ε }. Thus, everything except ε is in FOLLOW(T); that is, the symbol '+' is in FOLLOW(T). However, since FIRST(E') contains ε, the productions that contain 'T' on the right hand side, ie,

$E \rightarrow TE'$ and $E' \rightarrow +TE'$ becomes,

$E \rightarrow T$ and $E' \rightarrow +T$.

Now T is the rightmost symbol, and hence,

FOLLOW(T) includes FOLLOW(E) and FOLLOW(E'). (But note that FOLLOW(E) = FOLLOW(E').

Thus,

$$\text{FOLLOW}(T) = \{+, \text{FOLLOW}(E)\}$$
$$= \{+,), \$\}.$$

Similarly,

$$\text{FOLLOW}(T') = \text{FOLLOW}(T) = \{+,), \$\}$$

and,

$$\text{FOLLOW}(F) = \{\text{FIRST}(T')\}$$
$$= \{*, \text{FOLLOW}(T)\}$$
$$= \{*, +,), \$\}.$$

Imp: Note that follow will never contain 'ε'.

Q1) Find the FOLLOW(A) and FOLLOW(B).

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

Soln:

$\text{FOLLOW}(A) = \{a, b\}$

$\text{FOLLOW}(B) = \{b, a\}$

$\text{FOLLOW}(S) = \{\$\}$

Q2) $S \rightarrow Bb \mid Cd$

$B \rightarrow aB \mid \varepsilon$

$C \rightarrow cC \mid \varepsilon$

Soln:

$\text{FOLLOW}(S) = \{\$\}$

FOLLOW(B) = { b }

FOLLOW(C) = { d }

Q3) $S \rightarrow ACB \mid CbB \mid Ba$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid \varepsilon$

$C \rightarrow h \mid \varepsilon$

Soln: FOLLOW(S) = { $ }

FOLLOW(A) = { FIRST(C) }

= { h, FIRST(B) }

= { h, g, FOLLOW(S) }

= { h, g, $ }

FOLLOW(B) = { a, FOLLOW(S), FIRST(C) }

= { a, $, h, FOLLOW(A) }

= { a, $, h, g }

FOLLOW(C) = { FIRST(B), b, FOLLOW(A) }

= { g, FOLLOW(S), b, h, g, $ }

= { g, $, b, h }

Q4) $S \rightarrow aABb$

$A \rightarrow c \mid \varepsilon$

$B \rightarrow d \mid \varepsilon$

Soln: FOLLOW(S) = { $ }

FOLLOW(A) = { FIRST(B) }
= { d, b }

FOLLOW(B) = { b }.

Q5) S → ABC

A → DEF

B → ε

C → ε

D → ε

E → ε

F → ε

Sol^n:

FOLLOW(S) = { $ }

FOLLOW(A) = { FIRST(B) }
= { FIRST(C) } (since B → ε)
= { FOLLOW(S) } (Since C → ε)
= { $ }

FOLLOW(B) = { FIRST(C) }
= { FOLLOW(S) }
= { $ }

FOLLOW(C) = FOLLOW(S) = { $ }

FOLLOW(D) = FIRST(E) = FIRST(F) = FOLLOW(A) = { $ }

FOLLOW(E) = FIRST(F) = FOLLOW(A) = { $ }

FOLLOW(F) = FOLLOW(A) = { $ }

Q6) $S \rightarrow ABCDE$

$A \rightarrow a|\varepsilon$

$B \rightarrow b|\varepsilon$

$C \rightarrow c$

$D \rightarrow d|\varepsilon$

$E \rightarrow e|\varepsilon$.

Soln:

$\text{FOLLOW}(S) = \{\$\}$

$\text{FOLLOW}(A) = \{\text{FIRST}(B)\}$

$= \{b, \text{FIRST}(C)\}$

$= \{b, c\}$

$\text{FOLLOW}(B) = \{\text{FIRST}(C)\}$

$= \{c\}$

$\text{FOLLOW}(C) = \{\text{FIRST}(D)\}$

$= \{d, \text{FIRST}(E)\}$

$= \{d, e, \text{FOLLOW}(S)\}$

$= \{d, e, \$\}$

$\text{FOLLOW}(D) = \{\text{FIRST}(E)\}$

$= \{e, \text{FOLLOW}(S)\}$

$= \{e, \$\}$

$\text{FOLLOW}(E) = \text{FOLLOW}(S)$

$= \{\$\}$

LL(1) Grammars :-

Predictive parsers, that is, recursive-descent parsers without backtracking, can be constructed for a class of grammars called LL(1). The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a left-most derivation, and the "1" for using one lookahead i/p symbol at each step to make parsing action decisions.

Note :- No left-recursive or ambiguous grammar can be LL(1).

A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha | \beta$ are two distinct productions of G, the following conditions hold :

1) For no terminal 'a', do both α and β derive strings beginning with 'a'.

2) At most one of α and β can derive the empty string.

3) If $\beta \overset{*}{\Rightarrow} \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \overset{*}{\Rightarrow} \varepsilon$, then, β does not derive any string beginning with a terminal in FOLLOW(A).

The first two conditions are equivalent to the statement that FIRST(α) and FIRST(β) are disjoint sets. The third condition is equivalent to stating that if ε is in FIRST(β), then FIRST(α) and FOLLOW(A) are disjoint sets, and likewise if ε is in FIRST(α).

The following algorithm collects the information from FIRST and FOLLOW sets into a predictive parsing table M[A, a], a two-dimensional array, where A is a non-terminal, and 'a' is a terminal or the symbol '$', the input endmarker. The algorithm is based on the following idea:

The production $A \longrightarrow \alpha$ is chosen if the next input symbol 'a' is in FIRST(α). The only complication occurs when $\alpha = \varepsilon$ or, more generally, $\alpha \overset{*}{\Longrightarrow} \varepsilon$. In this case, we should again choose $A \longrightarrow \alpha$, if the current input symbol is in FOLLOW(A), or if $ on the input has been reached and $ is in FOLLOW(A).

ALGORITHM :- Construction of a predictive parsing table.

Input : Grammar G

Output : Parsing table M.

Method : For each production $A \longrightarrow \alpha$ of the grammar, do the following :

1) For each terminal 'a' in FIRST(α), add $A \longrightarrow \alpha$ to M[A, a].
2) If ε is in FIRST(α), then for each terminal 'b' in FOLLOW(A), add $A \longrightarrow \alpha$ to M[A, b]. If ε is in FIRST(α) and $ is in FOLLOW(A), add $A \longrightarrow \alpha$ to M[A, $] as well.

If, after performing the above, there is no production at all in M[A, a], then set M[A, a] to error, which we normally represent by an empty entry in the table.

Eg:- For the below grammar, produce a parsing table.
(Note: For creating the parsing table, you have to find out the FIRST() and FOLLOW() of all the non-terminals)

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Soln:- Find out the FIRST() and FOLLOW(). [already done]

FIRST(E) = { (, id } ; FOLLOW(E) = {), $ }

FIRST(E') = { +, ε } ; FOLLOW(E') = {), $ }

FIRST(T) = { (, id } ; FOLLOW(T) = { +,), $ }

FIRST(T') = { *, ε } ; FOLLOW(T') = { +,), $ }

FIRST(F) = { (, id } ; FOLLOW(F) = { *, +,), $ }

Now, create the parsing table, [~~rows → Nonterminal~~

[rows ⇒ non-terminals ; columns → all the terminals (except ε) and also add $]

Non-terminals	Input Symbols (terminals & $)					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

This algorithm can be applied to any grammar G to produce a parsing table M.

IMP :- But, note that for every LL(1) grammar, each parsing-table entry uniquely identifies a production or signals an error. For some grammars, M, (ie the parsing table), may have multiple entries in the same cell. Such grammars are not LL(1). For example, consider the below grammar,

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b.$

Construct the parsing table & check if it is LL(1) or not.

Sol^n:

FIRST(S) = {i, a}

FIRST(S') = {e, ε} $\Rightarrow$ FOLLOW(S') = { FOLLOW(S)}

= { $, FIRST(S')}

= { $, e}

FIRST(E) = {b}

Parsing table :

	i	t	a	e	b	$
S	$S \rightarrow iEtSS'$		$S \rightarrow a$			
S'				$S' \rightarrow eS$ $S' \rightarrow \varepsilon$		$S' \rightarrow \varepsilon$
E					$E \rightarrow b$.	

Here, it is clear that this grammar is not LL(1).

[Because, in the table, M[S', e] contains two productions, $S' \rightarrow eS$ and $S' \rightarrow \varepsilon$]

Predictive Parsing :-

The goal of predictive parsing is to construct a top-down parser that never backtracks. To do so, we must transform a grammar in two-ways:

① Eliminate left-recursion

② Perform left-factoring.

- These rules eliminate the most common causes for backtracking although they do not guarantee to produce LL(1) grammars which is completely back-track free parsing.

Predictive Parsers (i.e parser w/o backtracking) are of two types :

① Recursive Predictive Parser

② Non-recursive predictive parser.

Non-Recursive Predictive Parser :-

- It is possible to build a non-recursive predictive parser by maintaining a stack explicitly.
- The key problem during predictive parsing is that of determining the production to be applied for a non-terminal.
- The ~~pa~~ non-recursive parser looks up the production to be applied ~~for~~ in a parsing table.
- Requirements :-

① Stack

② Parsing Table

③ Input Buffer

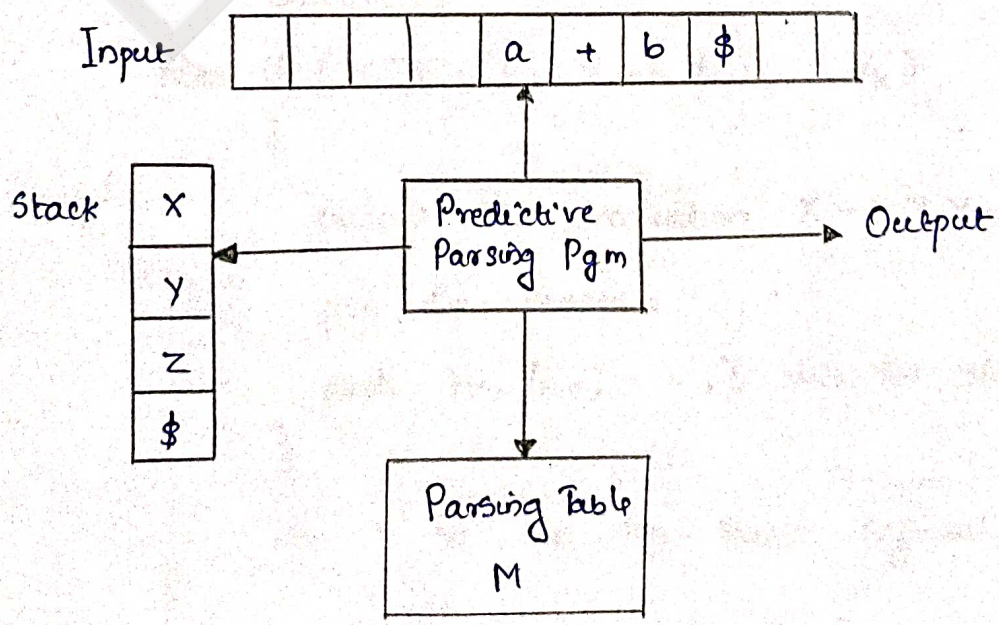
④ Parser.

The parser mimics a left-most derivation. If 'w' is the i/p that has been matched so far, then the stack holds a sequence of grammar symbols α such that,

$$S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha .$$

The table-driven parser has an input buffer, a stack containing a sequence of grammar symbols, a parsing table and an output stream. The i/p buffer contains the string to be parsed, followed by the endmarker $. We also use $ symbol to indicate the bottom of the stack, which initially contains the start symbol of the grammar on top of $.

The parser is controlled by a program that considers X, the symbol on top of the stack, and 'a', the current input symbol. If X is a non-terminal, the parser chooses an X-production by consulting the entry M[X, a] of the parsing table M. Otherwise, it checks for a match between the terminal X and current i/p symbol 'a'.



Algorithm :-

Table-driven predictive parsing.

Input :- A string 'w' and a parsing table M for grammar G_1.

Output :- If 'w' is in $L(G_1)$, a LMD of 'w'; otherwise, an error indication.

Method :- Initially, the parser is in a configuration with w$ in the input buffer and the start symbol S of G_1 on top of the stack, above $. The given below program uses the predictive parsing table M to produce a predictive parse for the input.

Pgm :-

```
let 'a' be the first symbol of 'w';
let 'X' be the top stack symbol;
while (X ≠ $)          // i.e. stack is not empty.
{
  if (X = a)
     pop the stack and let 'a' be the next symbol of w
  else if (X is a terminal)
     error();
  else if (M[X,a] is an error entry)
     error();
  else if (M[X,a] = X → Y1 Y2 .... Yk)
     {
       output the production X → Y1 Y2 .... Yk;
       pop the stack;
       push Yk, Yk-1, ...., Y1 into the stack, with Y1 on top;
     }
     let X be the top stack symbol;
}
```

Example :- Consider the grammar with $w = id + id * id$.

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

[Note: we have already created the parsing table M, for this given grammar].

Matched	Stack	Input	Action.
	$E\$$	$id + id * id \$$	
	$TE'\$$	$id + id * id \$$	output $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id \$$	output $T \rightarrow FT'$
	$id\,T'E'\$$	$id + id * id \$$	output $F \rightarrow id$
id	$T'E'\$$	$+ id * id \$$	match id
	$E'\$$	$+ id * id \$$	output $T' \rightarrow \varepsilon$
	$+TE'\$$	$+ id * id \$$	output $E' \rightarrow +TE'$
id+	$TE'\$$	$id * id \$$	match +
	$FT'E'\$$	$id * id \$$	output $T \rightarrow FT'$
	$id\,T'E'\$$	$id * id \$$	output $F \rightarrow id$
id+ id	$T'E'\$$	$* id \$$	match id
	$*FT'E'\$$	$* id \$$	output $T' \rightarrow *FT'$
id+id *	$FT'E'\$$	$id \$$	match *
	$id\,T'E'\$$	$id \$$	output $F \rightarrow id$
id+id* id	$T'E'\$$	$\$$	match id
	$E'\$$	$\$$	output $T' \rightarrow \varepsilon$
	$\$$	$\$$	output $E' \rightarrow \varepsilon$

Hence the string w is accepted by the parser.

Eliminating Ambiguity :-

To eliminate the ambiguity from a grammar, you have to re-write the grammar by considering two factors:

① Precedence of operators.

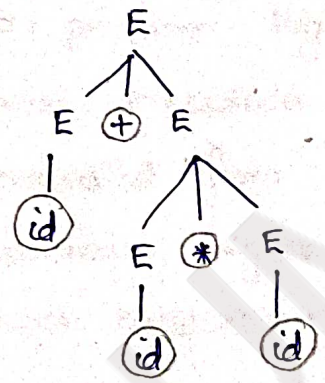
② Associativity of operators.

eg: Consider the below grammar for the string id + id * id.

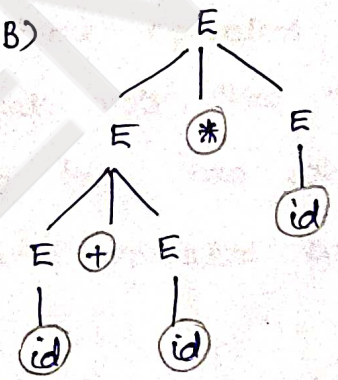
$E \longrightarrow E+E \mid E*E \mid (E) \mid id$

we can construct two parse trees:

(A)



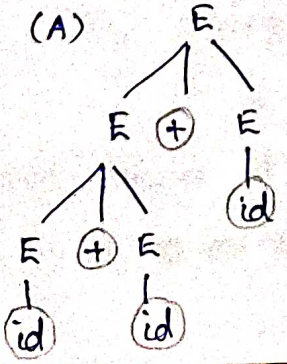
(B)



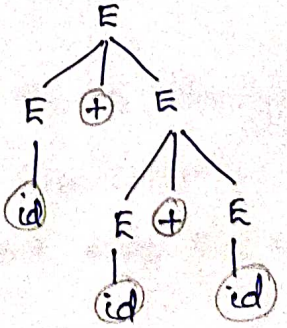
In parse tree (A), id * id is done before '+' operator. But in parse tree (B), + is done before *. But we know that * have higher precedence than +. And hence the parse tree (A) is correct.

Similarly, consider the string id+id+id.

(A)



(B)



we know that + is left associative and hence in id + id + id, the first + should be evaluated first followed by the second one. (ie, (id+id) + id). Thus it is clear that the parse tree (A) is correct.

Thus inorder to ensure the associativity and precedence of operators, we have to convert the grammar to,

$E \longrightarrow E+T \mid T$

$T \longrightarrow T * F \mid F$

$F \longrightarrow (E) \mid id$

ie, Inorder to ensure the precedence of operators, we have to perform higher precedence operators first followed by the lower precedence ones. But note that the parse tree are evaluated from bottom to top. Thus while rewriting the grammar, higher precedence operators will be produced by lower productions.

ie $E \longrightarrow E+E \mid E*E$ is converted like

$E \longrightarrow E+T \mid T$

$T \longrightarrow T * F \mid F$

Now * will be executed before +.

Similarly to ensure the associativity of operators, we make the grammar left recursive (for left-associative operators).

ie $E \longrightarrow E+E$ can be left or right recursive.

By changing it to,

$E \longrightarrow E+T \mid T$, this is changed to left recursion only.

and hence it becomes left associative.

(2020 KTU)

Q) Eliminate the ambiguity from the grammar

$E \rightarrow E * E \mid E - E \mid E \wedge E \mid E / E \mid E + E \mid (E) \mid id$

The associativity of the operators is as given below. The operators are listed in the decreasing order of precedence.

(i) ( )

(ii) / and + are right associative

(iii) $\wedge$ is left associative

(iv) * and - are left associative,

Soln:

$E \rightarrow E * T \mid E - T \mid T$ // lowest precedence for * & - and its left associative

$T \rightarrow T \wedge F \mid F$

$F \rightarrow G / F \mid G + F \mid G$

$G \rightarrow (E) \mid id$ // highest precedence for ( )

To check the correctness, consider any string.

~~(2+3) * 5~~ ie (id + id) * id

$E \rightarrow E * T$

$\rightarrow T * T$

$\rightarrow F * T$

$\rightarrow G * T$

$\rightarrow (E) * T$

$\rightarrow (T) * T$

$\rightarrow (F) * T \rightarrow (G+F) * T \rightarrow (id + F) * T$

$\rightarrow (id + G) * T \rightarrow (id + id) * T$

$\rightarrow (id + id) * F \rightarrow (id + id) * G$

$\rightarrow (id + id) * id$