UNIT 2: *SYNTAX ANALYSIS-1*

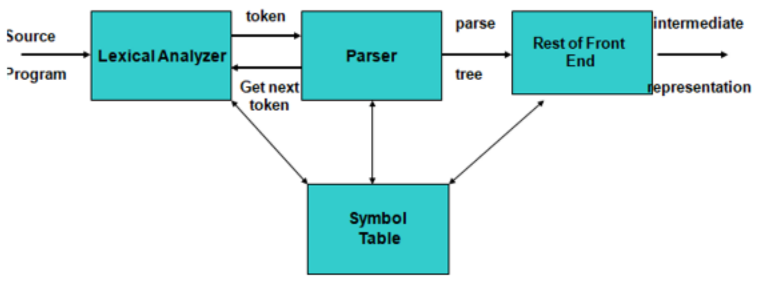
❖ Introduction:

- Syntax Analyser determines the structure of the program.
- The tokens generated from Lexical Analyser are grouped together and checked for valid sequence defined by programming language.
- Syntax Analyser uses context free grammar to define and validate rules for language construct.
- Output of Syntax Analyser is parse tree or syntax tree which is hierarchical / tree structure of the input.
- There is a need of mechanism to describe the structure of syntactic units or syntactic constructs of programming language. So, we use Context free grammars.

❖ Role of a parser:

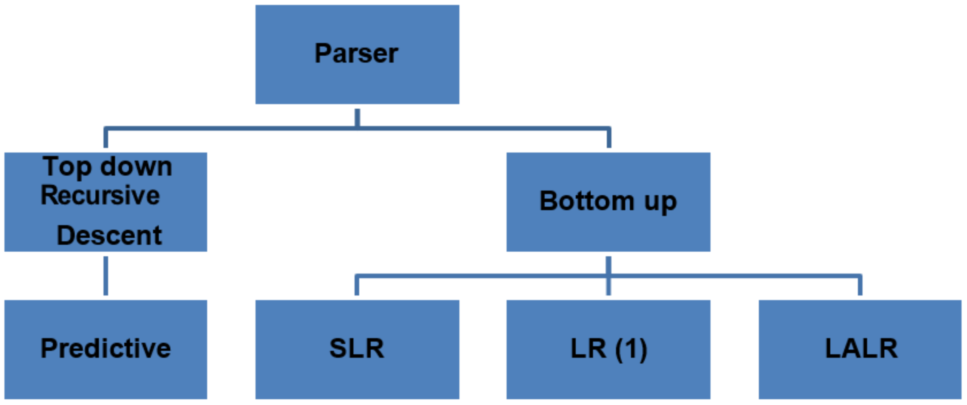
- The stream of tokens is input to the syntax analyser.
- The job of the parser is:
  - ✓ To identify the valid statement represented by the stream of tokens as per the syntax of the language. If it is a valid statement, it will be represented by a parse tree.
  - ✓ If it is not a valid statement, then a suitable error message is displayed, so that the programmer is able to correct the syntax error.

Position of Parser and role in Compiler model



- Usually, the semantic analysis and intermediate code generation can be interspersed with parsing. Hence, in addition to the validation of the programming statements parser also performs the following tasks:
  - ✓ Type-checking and providing the semantic consistency to the source programs.
  - ✓ Execution of semantic actions that are attached with grammar and responsible for generating the required intermediate form of the source program that facilitates some kind of code optimization.

❖ Classification of parser:



❖ *We categorize parser into 2 categories:*

1. **Top-Down Parser**

   The parse tree is created top to bottom, starting from the root.

2. **Bottom-Up Parser**

   The parse is created bottom to top, starting from the leaves

- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing
  - LR for bottom-up parsing

❖ Representative grammars:

The following grammar treats + and * alike, so it is useful for illustrating techniques for handling ambiguities during parsing:

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

Associativity and precedence are captured in the following grammar. E represents expressions consisting of terms separated by + signs, T represents terms consisting of factors separated by * signs, and F represents factors that can be either parenthesized expressions or identifiers:

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

This expression grammar belongs to the class of LR grammars that are suitable for bottom-up parsing. This grammar can be adapted to handle additional operators and additional levels of precedence. However, it cannot be used for top-down parsing because it is left recursive. The following non-left-recursive variant of the expression grammar will be used for top-down parsing:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid id$$

❖ Syntax Error Handling:

Common programming errors can occur at many different levels.

Lexical errors include misspellings of identifiers, keywords, or operators - e.g., the use of an identifier elipsesize instead of ellipsesize - and missing quotes around text intended as a string.

Syntactic errors include misplaced semicolons or extra or missing braces; that is, '((" or ")." As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).

Semantic errors include type mismatches between operators and operands. An example is a return statement in a Java method with result type void.

Logical errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==.
The program containing = may be well formed; however, it may not reflect the programmer's intent.

The error handler in a parser has goals that are simple to state but challenging to realize:

- Report the presence of errors clearly and accurately.
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correct programs.

❖ Error-Recovery Strategies:

1. *Panic-Mode Recovery:*

With this method, on discovering an error, the parser discards 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 program is clear and unambiguous. The compiler designer must select the synchronizing tokens appropriate for the source language.

➢ ***Advantage:***
simple, and is guaranteed not to go into an infinite loop.

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

2. *Phrase-Level Recovery:*

On discovering an error, a 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. A 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.

➢ ***Advantage:***
Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string.

➢ ***Disadvantage:***
Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

3. *Error Productions:*

By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. A parser constructed from a grammar augmented by these error productions detects 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. *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 number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible. Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest. Do note that a closest correct program may not be what the programmer had in mind. Nevertheless, the notion of least-cost correction provides a yardstick for evaluating error-recovery techniques and has been used for finding optimal replacement strings for phrase-level recovery.

❖ *Context-Free Grammars:*

- Inherently recursive structures of a programming language are defined by a context-free grammar.
- In a context-free grammar, G={ V, T, S, P }, we have:

V: A finite set of non-terminals (syntactic-variables)
T: A finite set of terminals (in our case, this will be the set of tokens or lexical units)
S: A start symbol (one of the non-terminal symbols)
P: A finite set of productions rules in the following form
$A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string)

Example:

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

❖ *Derivations:*

- $E \Rightarrow E+E$ i.e., E+E derives from E, which means that we can replace E by E+E
- To able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar.
- $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$
- A sequence of replacements of non-terminal symbols is called a **derivation**.
- In general, a derivation step is $\alpha A \beta \Rightarrow \alpha\gamma\beta$ if there is a production rule $A \rightarrow \gamma$ in our grammar where α and β are arbitrary strings of terminal and non-terminal symbols

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

$\Rightarrow$: derives in one step

$\overset{*}{\Rightarrow}$: derives in zero or more steps

$\overset{+}{\Rightarrow}$: derives in one or more steps

Example:

E ⇒ -E ⇒ -(E) -(E+E) ⇒ -(id+E) ⇒ -(id+id)

OR

E ⇒ -E ⇒ -(E) ⇒ -(E+E) ⇒ -(E+id) ⇒ -(id+id)

- At each derivation step, we can choose any of the non-terminals in the sentential form of G for the replacement.
- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.

Example: E ⇒ -E ⇒ -(E) ⇒ -(E+E) ⇒ -(id+E) ⇒ -(id+id)

- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

Example: E ⇒ -E ⇒ -(E) ⇒ -(E+E) ⇒ -(E+id) ⇒ -(id+id)

- We will see that the top-down parsers try to find the left-most derivation of the given source program.

We will see that the bottom-up parsers try to find the right-most derivation of the given source program in the reverse order.

❖ *Parse Tree:*

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.

Example: Consider the grammar

S → b A │a B

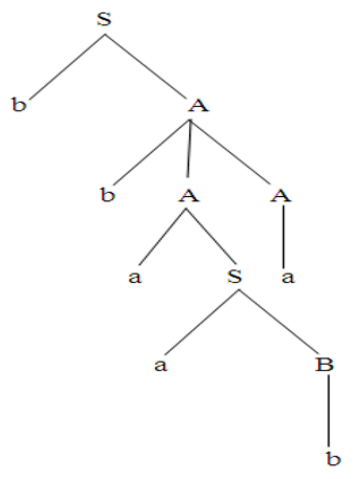
A → b A A │ a S │ a

B → a B B │ b S │ b

Writing leftmost and rightmost derivation for the sentence **bbaaba** along with Parse tree.

i) bbaaba

ii) bbbaaaba (Home Work)



Leftmost Derivation

S ⇒ b A

⇒ b b $\underline{A}$ A

⇒ b b a $\underline{S}$ A

⇒ b b a a $\underline{B}$ A

⇒ b b a a b A

⇒ b b a a b a

Rightmost derivation

S ⇒ b $\underline{A}$

⇒ b b A $\underline{A}$

⇒ b b $\underline{A}$ a

⇒ b b a $\underline{S}$ a

⇒ b b a a $\underline{B}$ a

⇒ b b a a b a

Fig. 3.5 Parse Tree for the string bbaaba

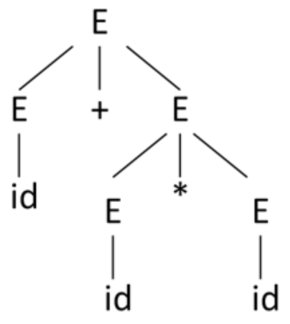
❖ *Ambiguity:*

A grammar produces more than one parse tree for a sentence is called as an ambiguous grammar.

Example:

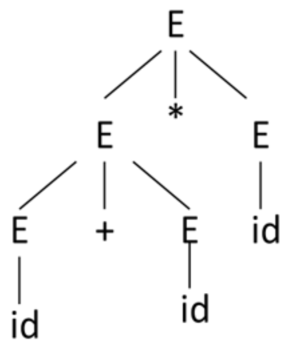
1. $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E$

$\Rightarrow id+id*E \Rightarrow id+id*id$



$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E$

$\Rightarrow id+id*E \Rightarrow id+id*id$



2. $A \rightarrow B\ C \mid a\ a\ C$
   $B \rightarrow a \mid B\ a$
   $C \rightarrow b$

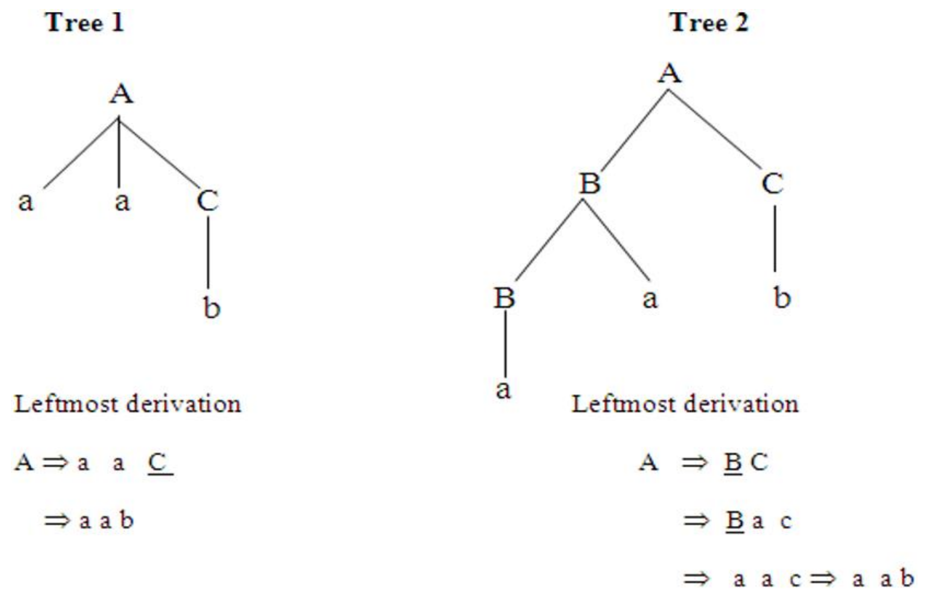


Fig. 3.20 Two leftmost derivation for string a a b

- For the most parsers, the grammar must be unambiguous.
- Unambiguous grammar implies unique selection of the parse tree for a sentence.
- We must prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.

❖ *Ambiguity – Operator Precedence:*

Let us consider the grammar

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\hat{}E \mid id \mid (E)$$

At each step we begin by introducing One non-terminal - NT for each precedence level.

Priority levels (High to low)

Exponentiation (^)	-	F is NT	(right associative rule)
Multiplicative operator (*, /)	-	T is NT	(right associative rule)
Additive operator (+, -)	-	E is NT	(right associative rule)

A subexp 'E' that is indivisible is either an identifier or parenthesized expression which is written as

$$G \rightarrow id \mid (E) \qquad \text{where G is New NT}$$

- To write next rule we take New NT for the next highest priority level, and this is connected with o Zero or more instance of next highest priority operator with previous level NT

$$F \rightarrow G\hat{}F \mid G$$

- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the precedence and associativity rules.

Example:

1. **Disambiguate the grammar** $E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\hat{}E \mid id \mid (E)$

   precedence: ^ (right to left)

   * (left to right)

   + (left to right)

Ans.: $E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T*F \mid TF \mid F$

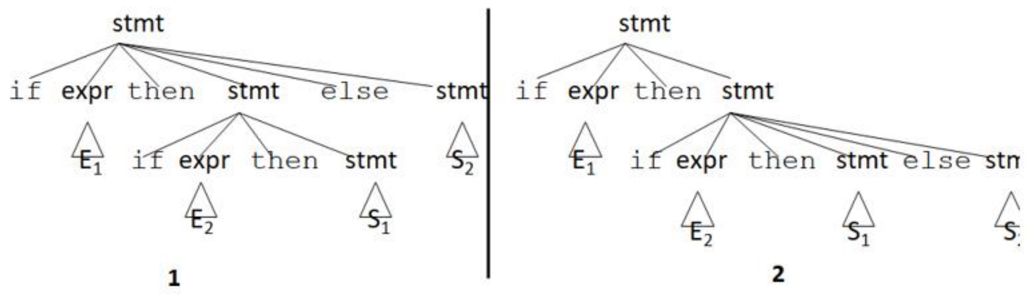
$F \rightarrow G\hat{}F \mid G$

$G \rightarrow id \mid (E)$

2. **Disambiguating the problem of dangling else**

```
stmt → if expr then stmt  |
       if  expr then stmt else stmt  |  otherstmts
```

if E_1 then if E_2 then S_1 else S_2



We prefer the second parse tree (else matches with closest then). So, we have to disambiguate our grammar to reflect this choice. The unambiguous grammar will be:

stmt $\rightarrow$ matchedstmt
| unmatchedstmt

matchedstmt $\rightarrow$ if expr then matchedstmt else matchedstmt
| otherstmts

unmatchedstmt $\rightarrow$ if expr then stmt
| if expr then matchedstmt else unmatchedstmt

❖ *Left Recursion:*

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation,
  $A \Rightarrow A\alpha$ for some string α
- Top-down parsing techniques **cannot** handle left-recursive grammars. So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.
- The left-recursion may appear in a single step of the derivation (*immediate left-recursion*) or may appear in more than one step of the derivation.

❖ *Immediate left Recursion:*

$A \rightarrow A\ \alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\ A'$

$A' \rightarrow \alpha\ A' \mid \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\ \alpha_1 \mid \ldots \mid A\ \alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\ A' \mid \ldots \mid \beta_n\ A'$

$A' \rightarrow \alpha_1\ A' \mid \ldots \mid \alpha_m\ A' \mid \varepsilon$ an equivalent grammar

Example:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

$\Downarrow$ eliminate immediate left recursion

$E \rightarrow T\ E'$

$E' \rightarrow +T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

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

$F \rightarrow id \mid (E)$

❖ *Left-Recursion – Problem:*

A grammar cannot be immediately left-recursive, but it still can be left-recursive. By just eliminating the immediate left-recursion, we may not get a grammar which is not left- recursive.

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive, but it is still *left-recursive*.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ OR

$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

So, we have to eliminate all left-recursions from our grammar.

❖ *Eliminate Left-Recursion – Algorithm:*

- Arrange non-terminals in some order: $A_1 \ldots A_n$

```
- for i from 1 to n do {
        - for j from 1 to i-1 do {
              replace each production
                  $A_i \rightarrow A_j \gamma$
                    by
                  $A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_1 \gamma$
                  where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$
        }
        - eliminate immediate left-recursions among Ai productions
}
```

Example 1:

$$\mathbf{S \rightarrow Aa \mid b}$$

$$\mathbf{A \rightarrow Ac \mid Sd \mid f}$$

- Order of non-terminals: S, A

for S:

- we do not enter the inner loop.
- there is no immediate left recursion in S.

for A:

- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$

So, we will have $A \rightarrow Ac \mid Aad \mid bd \mid f$

- Eliminate the immediate left-recursion in A

$A \rightarrow bdA' \mid fA'$

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

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid fA'$

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

Example 2:

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: A, S

for A:

- we do not enter the inner loop.
- Eliminate the immediate left-recursion in A

$A \rightarrow SdA' \mid fA'$

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

for S:

- Replace $S \rightarrow Aa$ with $S \rightarrow SdA'a \mid fA'a$

So, we will have $S \rightarrow SdA'a \mid fA'a \mid b$

- Eliminate the immediate left-recursion in S

$S \rightarrow fA'aS' \mid bS'$

$S' \rightarrow dA'aS' \mid \varepsilon$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow fA'aS' \mid bS'$

$S' \rightarrow dA'aS' \mid \varepsilon$

$A \rightarrow SdA' \mid fA'$

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

❖ *Left-Factoring:*

A predictive parser (a top-down parser without backtracking) insists that the grammar must be left-factored.

grammar $\rightarrow$ a new equivalent grammar suitable for predictive parsing

stmt $\rightarrow$ if expr then stmt else stmt

| if expr then stmt

when we see if, we cannot immediately decide which production rule to choose to expand stmt in the derivation. In general,

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

Here α is non-empty and the first symbols of β_1 and β_2 (if they have one) are different. while processing if the input begins string derived from α we do not know or decide whether to expand

A to $\alpha\beta_1$ or A to $\alpha\beta_2$

However, we can defer the decision by first expanding A to $\alpha A'$ and then after seeing the i/p derived from α and look ahead symbol, we expand A' to β_1 or β_2. This is left factored, and the production are re-written for the grammar and is as follows:

$$A \rightarrow \alpha A'$$

$A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

❖ *Left-Factoring – Algorithm:*

Input: Grammar G

Output: An equivalent left factored grammar

Method:

1. For each Non-terminal A find the longest prefix α common to two or more alternatives (production rules).
2. If $\alpha <> \varepsilon$ then replace all of A-productions

   $A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$ where γ_i represents all alternatives that do not begin with α

   by

   $A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$ Here A' is a new non-terminal

   $A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$
3. Steps 1 and 2 are repeated until no two alternatives for a non-terminal have a common prefix.

Example 1:

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

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

Example 2:

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

❖ *Parsing:*

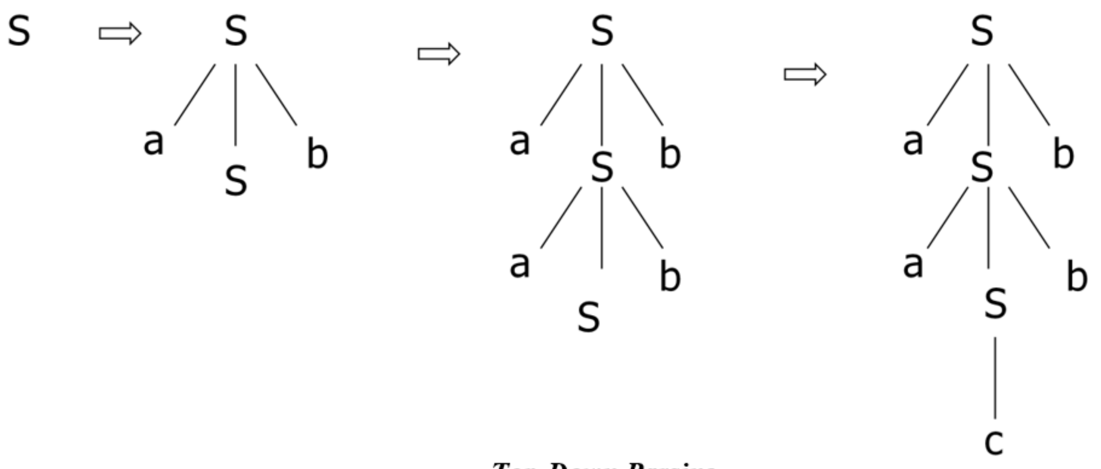
➢ *Top-Down Parsing*

In Top-Down Parsing we start from the start symbol of the grammar and by choosing the production judiciously we try to derive the given sentence.

➢ *Bottom-Up Parsing*

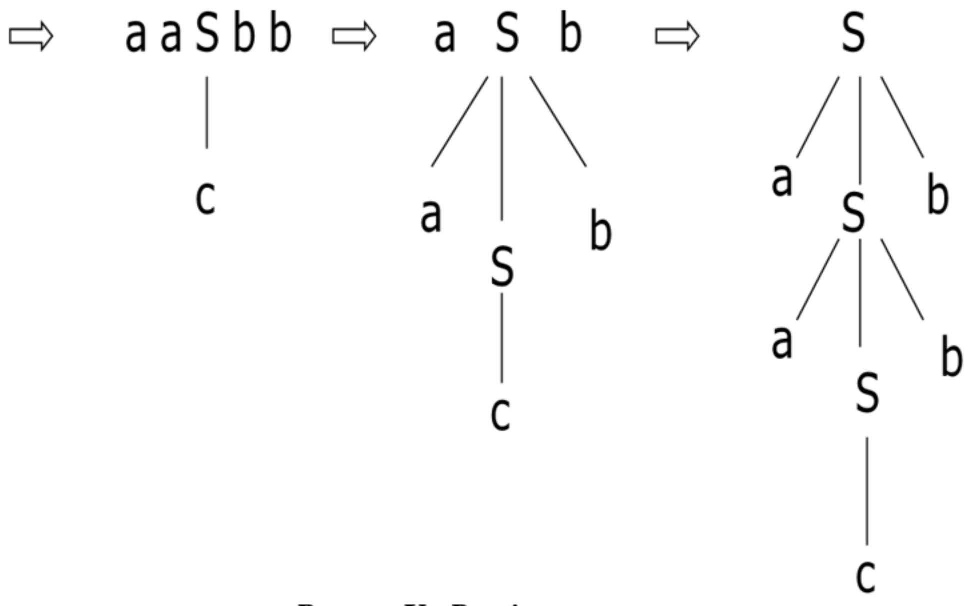
In bottom-up parsing we start from the given sentence and using various production, we try to reach the start symbol.

Consider the grammar S -> aSb | c for the string aacbb



Top-Down Parsing

aacbb=>



Bottom-Up Parsing

❖ *Design strategy for Top-Down Parser:*

Basically, Top-Down Parsing can be viewed as an attempt to find leftmost derivation for an input string and constructs a parse tree from root to leaves. The following steps may be followed.

1. Given a Non-terminal (Initially Start symbol) which is to be expanded, the first alternative (production rule) is used for expansion.

2. Within the newly expanded string, the substring of terminals from left are compared with input string. If found to be match, then next left most non-terminal is selected for expansion and the step 2 is repeated.

3. Otherwise, the current alternative structure (production rule) selected is incorrect, hence undo the previous expansion and use the next alternative structure of the non-terminal for expansion and step 2 is repeated.

4. In the process of step 2 and 3 if no alternative structure for a non-terminal to be tried then process is backed up by undoing all the previous expansion. In the process of backtracking, if we reach start symbol and no alternative structure to be tried then input is invalidated. Otherwise, if no non-terminal are left for expansion then input is validated.

❖ *Recursive-Descent Parsing (uses Backtracking):*

- In order to find the correct production, the general form of Top-Down parser uses **backtracking** (via recursive calls) and is called is **Recursive Descent parser**.

- It consists of **set of procedures**, one for each non-terminal and looks for a substring of input string and if it is found it returns **TRUE**, otherwise, it returns **FALSE**.

- Execution begins with a call to procedure for **start symbol** which **halts and announces successful parsing** if its procedure body scans the entire input string, otherwise

- The pseudo-code for a typical non-terminal may be written as follows:

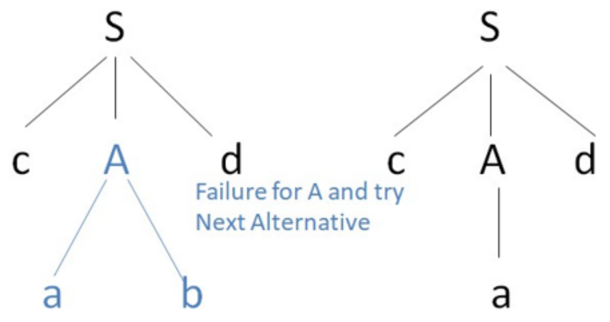
Recursive-Descent Parsing (uses Backtracking)

- Example:

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

input: cad



Pseudo Code for implementation:

```
main()
{
    i=1; /* index pointing to input string */ read input
        if (S() and input[i] == $)
            print(“ String is valid “);
               else
            print(“ String is Invalid “);
}
```

```
int S()
{
  If input[i] ==‘c’ then
  {
     i=i+1;
     If A() then
     {
        if input[i]==‘d’
        {
           i=i+1;
           return 1;
        }
     }
     else
            return 0;
  }
   else
        return 0;
}
```

```
Int A()
{
  isave=i;
  if input[i] == ‘a’ then
  {
     i=i+1;
     if input[i] == ‘b’ then
     {
        i=i+1;
        return 1;
     }
  }
      i=isave;
      if input[i]==‘a’ then
      {
         i=i+1;
         return 1;
      }
   else
   {
      return 0;
   }
}
```

Pseudo-code for Non-terminal in Recursive-descent Parser (RDP):

```
Void A()
{
      Choose an A-production $A \rightarrow X_1X_2X_3 \ldots X_k$
      for ( i=1 to k)
      {
         if ($X_i$ is Non-terminal) call procedure $X_i$()
              else if ($X_i$ equals the current input symbol ‘a’)
         advance the input to the next symbol
              else
         error() /* error and try next alternative for A-production */
      }
}
```

Difficulties of Recursive Descent parser:

1. A left recursive grammar creates Top-Down Parser to go into an infinite loop. i.e if $A \rightarrow A\alpha$ is a A-production then, when we try to expand A, we may find ourselves again trying to expand A without having consumed any input.

2. A second problem concerns backtracking. If we make a sequence of erroneous expansion, we may have to undo the semantic action taken. This slows the process of parsing hence backtracking must be avoided.

3. The order in which the alternative structure (production rules) is selected for non-terminal would affect the language accepted.

Prerequisites for Predictive Top-Down Parsers:

- Elimination of Left-recursion
- Left Factoring
- First Set
- Follow Set

First and follow sets:

- The implementation of both Top-down parser and bottom parser is aided by two function name **FIRST** and **FOLLOW** sets associated with **grammar G**.
- These two sets allow us to choose which production to be selected based on the next input symbol
- FIRST(α): It is defined to be the set terminals that begin string derived from α.
- How is it used?

Consider two A-production

$A \rightarrow \alpha \mid \beta$ where FIRST(α) and FIRST(β) are disjoint sets.

Let us consider the terminal ‘a’ to be first symbol which is either in FIRST(α) or FIRST(β) but not in both. When choosing A-production we see the look-ahead symbol ‘a’ from the input. If ‘a’ in FIRST(α) then select A production as $A \rightarrow \alpha$ or If ‘a’ in FIRST(β) then select A production as $A \rightarrow \beta$

FIRST set Computation:

FIRST(**X**) for all Grammar symbols **X** can be computed by applying 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**)= {**X**}

2. **IF X = ε or X→ε then** FIRST(**X**) = {**ε**}

3. **IF X is a Non-Terminal and X→**$Y_1Y_2Y_3 \cdots Y_k$ **then**

FIRST (**X**)=FIRST ($Y_1Y_2Y_3 \cdots Y_k$)

= FIRST(Y_1) → **if FIRST(Y_1) does not derive any empty string ε**

FIRST (**X**)=FIRST ($Y_1Y_2Y_3 \cdots Y_k$)

= FIRST(Y_1) – {ε} U FIRST (Y_2Y_3 $Y_k \cdots$ Y)

→ **if Y_1 derive an empty string ε.**

FIRST ($Y_2Y_3 \ldots Y_k$) = FIRST(**Y2**)

→ **if Y_2 does not derive an empty string ε.**

FIRST(Y_2Y_3 Y_k) =FIRST (**Y2**) – {ε}U FIRST ($Y_3 \ldots Y_k$)

→ **if Y_2 derive an empty string ε.**

This is repeated for each Yi until no more terminals or ε can be added

Examples to solve:

1. E → E+T | T

   T → T*F | F

   F → id | (E)

2. E → T E'

   E' → +T E' | ε

   T → F T'

   T' → *F T' | ε

   F → id | (E)

3. S → ACB | CbB | Ba

   A → da | BC

   B → g | ε

   C → h | ε

4. S → AaAb | BbBa

   A → ε

   B → ε

FOLLOW computation:

- If the grammar is **ε-free** then **FIRST** symbols are used in selecting the appropriate production for some non-terminal and these gets added to Parsing table
- But when the grammar is **not ε-free**, the FIRST symbols cannot be used to decide the appropriate productions, as these are not added to parsing table. i.e If there is production **A→ε** in the grammar then when **A** is replaced by ε cannot be decided by the FIRST symbols and hence additional information is required to decide when **A→ε** is to be used so that it can be added in the table. Here we need **FOLLOW** symbols to take the decision.
- FOLLOW(A) : It is defined to be the set of terminals 'a' that can appear immediately to the right of A in some sentential form
- $S \Rightarrow \alpha A a \beta$
- To Compute FOLLOW(**A**) for all Non-terminals, apply the following rules until nothing can be added to any follow Set

1. Place **$** in FOLLOW(**S**), where **S** is the start symbol **$** is the input right end-marker.
2. If there is a production $A \rightarrow \alpha \mathbf{B} \beta$ then everything in FIRST(**β**) except ε is in FOLLOW(**B**).
3. If there is production $A \rightarrow \alpha \mathbf{B}$ or a production $A \rightarrow \alpha \mathbf{B} \beta$ where FIRST(β) contains **ε** then everything in FOLLOW(**A**) is FOLLOW(**B**). i.e FOLLOW(**B**) = FOLLOW(**A**)

LL(1) Parsers:

A grammar such that it is possible to choose the correct production with which to expand a given nonterminal, looking only at the next input symbol, is called LL(1). These grammars allow us to construct a predictive parsing table that gives, for each nonterminal and each lookahead symbol, the correct choice of production. Error correction can be facilitated by placing error routines in some or all of the table entries that have no legitimate production.

The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of lookahead at each step to make parsing action decisions.

A grammar G is LL(1) if and only if whenever A → **a** |**B** are two distinct productions of G, the following conditions hold:

1. For no terminal a do both **a** and **B** derive strings beginning with a.
2. At most one of a and **B** can derive the empty string.
3. If **B** =>**E**, then a does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if **a** => **E** then **B** does not derive any string beginning with a terminal in FOLLOW(A).

ALGORITHM: construction of predictive parsing table
Consider the Grammar- G:

For each production **A** → **α** do the following:

a) Find **FIRST (α) – call set as { S1 }**
and **FOLLOW (A) - call set as { S2 }**

b) For all symbols in **{S1}** make entries in the table as
TABLE**[A, a] = A → α ,** where **a is S1**

c) if **ε** is in **{S1}** then make the entries in the table as
TABLE**[A, b] = A → α.** where **b is S2**

Example:

for this grammar,

$E \rightarrow T\,E'$

$E' \rightarrow +\,T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *\,F\,T' \mid \varepsilon$

$F \rightarrow (\,E\,) \mid id$

We have first and follow symbols as →

	FIRST	**FOLLOW**
E	**(id**	**) $**
E'	**+ , ε**	**) $**
T	**(id**	**+) $**
T'	*** , ε**	**+) $**
F	**(id**	**+ *) $**

	id	+	*	(	)	$
E	**E→TE'**			**E→TE'**		
E'		**E' → + T E'**			**E' →ε**	**E' → ε**
T	**T→FT'**			**T→ FT'**		
T'		**T'→ ε**	**T'→*FT'**		**T'→ ε**	**T'→ ε**
F	**F→id**			**F→ (E)**		

And this is the parsing table that can be obtained.

Example:

for this grammar,

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

We have first and follow symbols as →

	FIRST
S	**a, b**
A	**ε**
B	**ε**

M	a	b	$
S	S ->AaAb	S->BbBa	
A	**A → ε**	**A → ε**	
B	**B → ε**	**B → ε**	

Tracing for input string ba:

matched	**stack**	**input**	**action**
	S$	ba$	
	BbBa $	ba$	Consult table M[S,b] i.e., S->BbBa pushed onto stack
	bBa $	ba$	Consult table M[B,b] i.e., **B → ε** pushed onto stack
b	Ba $	a$	Match b
	a $	a$	Consult table M[B,a] i.e., **B → ε** pushed onto stack
a	$	$	Match a

As we have $ on top of stack and input pointer, string is successfully parsed. And parse tree can be generated for this string.

Non-recursive Predictive Parsing:

A non-recursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls.

If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols a such that

*

S=> w**a**

lm

The table-driven parser has an input buffer, a stack containing a sequence of grammar symbols, a parsing table constructed by Algorithm, and an output stream.

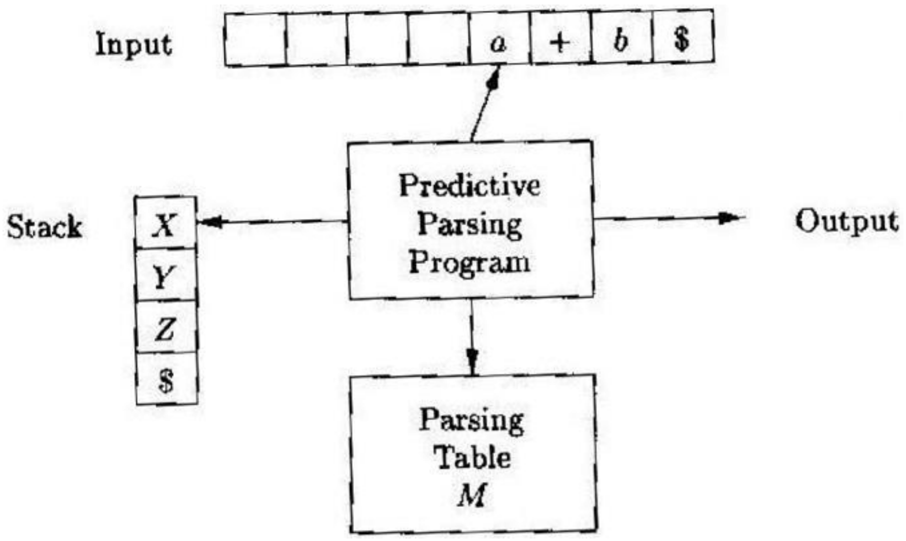
The input buffer contains the string to be parsed, followed by the endmarker $.

We reuse the symbol $ to mark 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 nonterminal, the parser chooses an X-production by consulting entry M[X, a] of the parsing table .

Otherwise, it checks for a match between the terminal X and current input symbol a.



Algorithm:
Table-driven predictive parsing.

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.

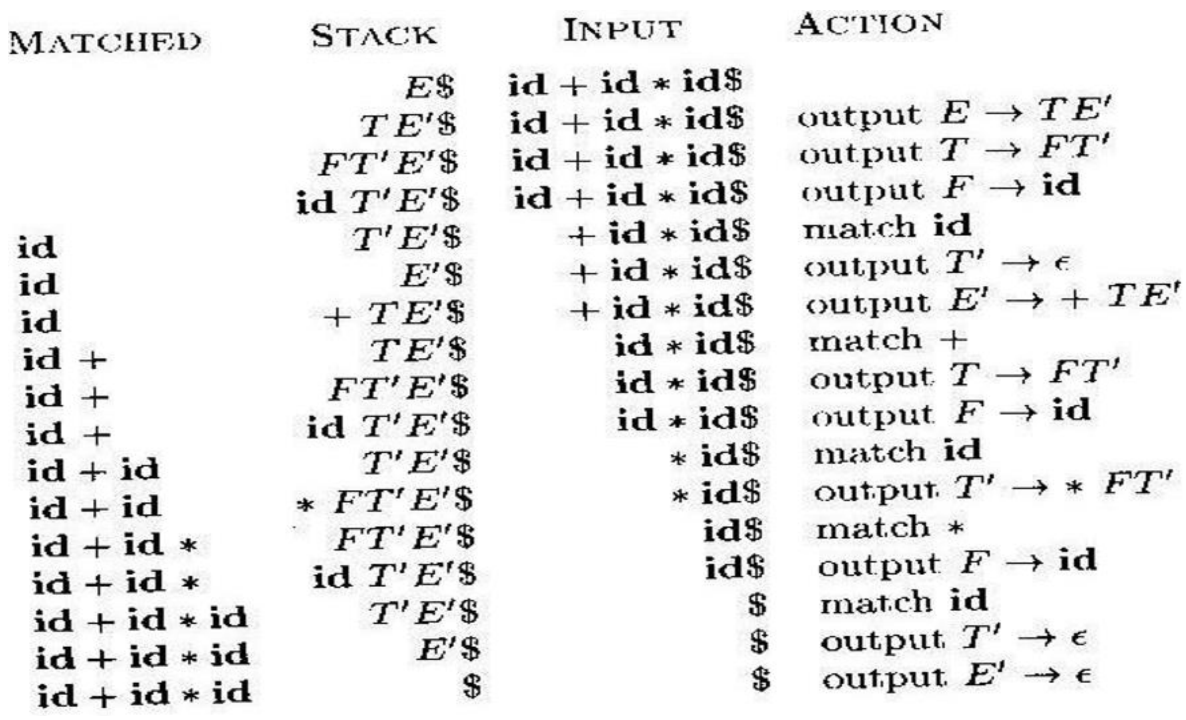
METHOD:
/* Initially, the parser is in a configuration with w$ in the input buffer and the start symbol S of G on top of the stack, above $.*/

```
set ip to point to the first symbol of w;
set X to the top stack symbol; while ( X!= $ )
{

/* stack is not empty */
if ( X is a )
      pop the stack and advance zp;
else if ( X is a terminal )
      error();
else if ( M[X, a] is an error entry )
      error();
else if ( M[X,a] = X -+ Y1Y2 Yk )
{
      output the production X -+ YlY2 -. Yk;
pop the stack; push Yk, Yk-1,. . . , Yl onto the stack, with Yl on top;
}
set X to the top stack symbol;

}
```

Example:
On input id + id * id, the non-recursive predictive parser of Algorithm makes the sequence of moves.



MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to + TE'$
$\mathbf{id}\ +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id}\ +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id}\ +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
$\mathbf{id} + \mathbf{id}\ *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id}\ *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

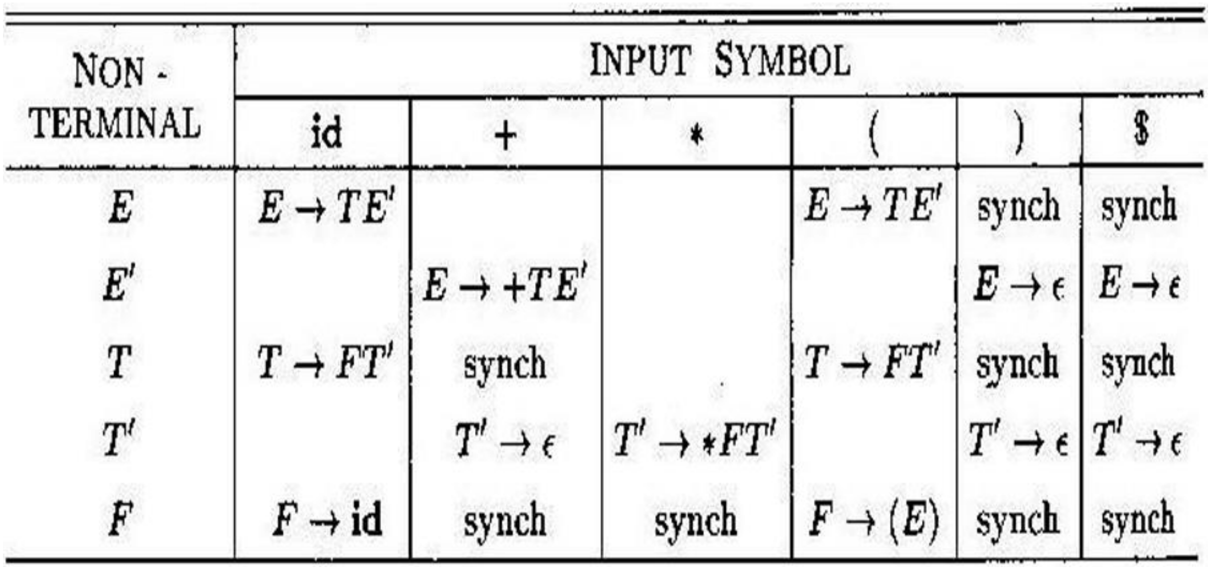
Error Recovery in Predictive Parsing:

An Error is detected when:

- TRM on top of stack does not match with next i/p symbol.
- TABLE [A, a] is error i.e. table entry is empty.

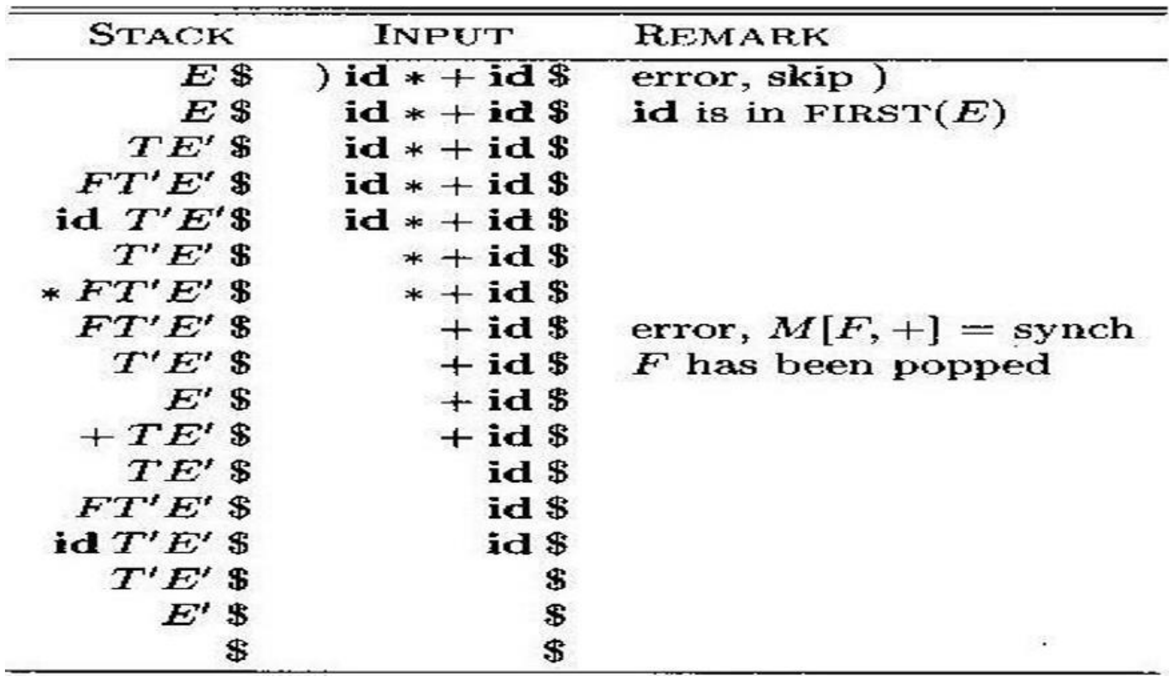
1. *PANIC MODE OF ERROR RECOVERY:*

- Skipping the symbol on the i/p until a token in selected set of synchronizing tokens appears and popping the current non-terminal from the stack
- SYNC- TOKEN (A) = FOLLOW (A)
- If we add symbols in FIRST (A) to Synchronizing set of non TRM A, then it may be possible to resume parsing according to A if a symbol in FIRST (A) appears in the i/p.
- If A→ ε ; this can be used so that some error detection may be postponed, but cannot cause error to be missed.
- If TRM cannot be matched, pop the terminal and issue an message saying that TRM was inserted and continue parsing.
- ***Example:*** parsing table above can be modified as →



NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ id	synch	synch	$F \to (E)$	synch	synch

And now the nonrecursive predictive parser of Algorithm makes the sequence of moves



STACK	INPUT	REMARK
E $	) id * + id $	error, skip)
E $	id * + id $	id is in FIRST(E)
TE' $	id * + id $	
$FT'E'$ $	id * + id $	
id $T'E'$$	id * + id $	
$T'E'$ $	* + id $	
* $FT'E'$ $	* + id $	
$FT'E'$ $	+ id $	error, $M[F, +]$ = synch
$T'E'$ $	+ id $	F has been popped
E' $	+ id $	
+ TE' $	+ id $	
TE' $	id $	
$FT'E'$ $	id $	
id $T'E'$ $	id $	
$T'E'$ $	$	
E' $	$	
$	$	

2. *PHRASE-LEVEL ERROR RECOVERY:*

- This is implemented by filling the blank entries in the parsing table with pointers to error routines. These routines may change, insert, or delete symbols in the input or STACK and issue appropriate error messages.