# UNIT-II

## SYNTAX ANALYSIS

Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.

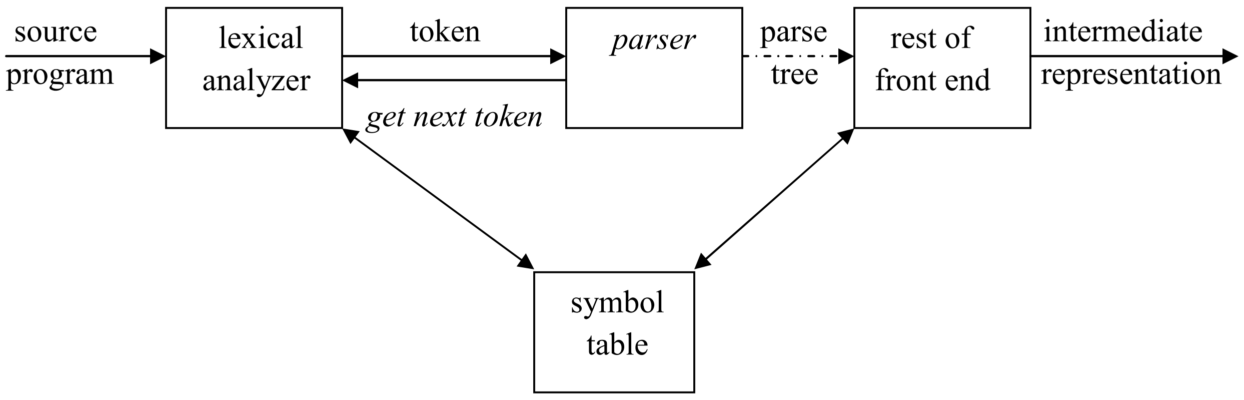
**Advantages of grammar for syntactic specification :**

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.
2. An efficient parser can be constructed automatically from a properly designed grammar.
3. A grammar imparts a structure to a source program that is useful for its translation into object code and for the detection of errors.
4. New constructs can be added to a language more easily when there is a grammatical description of the language.

## THE ROLE OF PARSER

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

***Position of parser in compiler model***



**Functions of the parser :**

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs error recovery.

**Issues :**

Parser cannot detect errors such as:

1. Variable re-declaration
2. Variable initialization before use.
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

**Syntax error handling :**

Programs can contain errors at many different levels. For example :
1. Lexical, such as misspelling a keyword.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.

Functions of error handler :

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

**Error recovery strategies :**

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

**Panic mode recovery:**

On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or **end**. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

**Phrase level recovery:**

On discovering an error, the parser performs local correction on the remaining input that allows it to continue. Example: Insert a missing semicolon or delete an extraneous semicolon etc.

**Error productions:**

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

**Global correction:**

Given an incorrect input string x and grammar G, certain algorithms can be used to find a parse tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in general too costly in terms of time and space.

## CONTEXT-FREE GRAMMARS

A Context-Free Grammar is a quadruple that consists of **terminals**, **non-terminals, start symbol** and **productions.**

**Terminals :** These are the basic symbols from which strings are formed.

**Non-Terminals :** These are the syntactic variables that denote a set of strings. These help to define the language generated by the grammar.

**Start Symbol :** One non-terminal in the grammar is denoted as the "Start-symbol" and the set of strings it denotes is the language defined by the grammar.

**Productions :** It specifies the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

**Example of context-free grammar:** The following grammar defines simple arithmetic expressions:

$$
\begin{aligned}
expr &\rightarrow expr\ op\ expr \\
expr &\rightarrow (expr) \\
expr &\rightarrow -\ expr \\
expr &\rightarrow \mathbf{id} \\
op &\rightarrow + \\
op &\rightarrow - \\
op &\rightarrow * \\
op &\rightarrow / \\
op &\rightarrow \uparrow
\end{aligned}
$$

In this grammar,

- **id** + - * / ↑ ( ) are terminals.
- *expr* , *op* are non-terminals.
- *expr* is the start symbol.
- Each line is a production.

**Derivations:**

Two basic requirements for a grammar are :
1. To generate a valid string.
2. To recognize a valid string.

**Derivation** is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

**Example :** Consider the following grammar for arithmetic expressions :

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

To generate a valid string - ( id+id ) from the grammar the steps are

1. E → - E
2. E → - ( E )
3. E → - ( E+E )
4. E → - ( id+E )
5. E → - ( id+id )

In the above derivation,

- E is the start symbol.
- - (id+id) is the required sentence (only terminals).
- Strings such as E, -E, -(E), . . . are called sentinel forms.

**Types of derivations:**

The two types of derivation are:

1. Left most derivation
2. Right most derivation.

- In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.

- In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

**Example:**

Given grammar G : E → E+E | E*E | ( E ) | - E | id

Sentence to be derived : – (id+id)

| LEFTMOST DERIVATION | RIGHTMOST DERIVATION |
|---|---|
| E → - E | E → - E |
| E → - ( E ) | E → - ( E ) |
| E → - ( E+E ) | E → - (E+E ) |
| E → - ( id+E ) | E → - ( E+id ) |
| E → - ( id+id ) | E → - ( id+id ) |

- String that appear in leftmost derivation are called **left sentinel forms.**
- String that appear in rightmost derivation are called **right sentinel forms.**

**Sentinels:**

Given a grammar G with start symbol S, if S → α , where α may contain non-terminals or terminals, then α is called the sentinel form of G.

**Yield or frontier of tree:**

Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called **yield** or **frontier** of the tree.

**Ambiguity:**

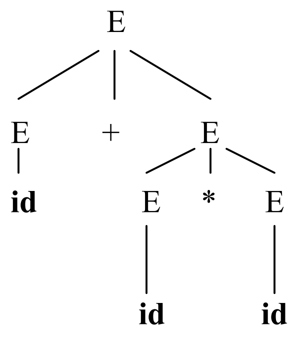
A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**.

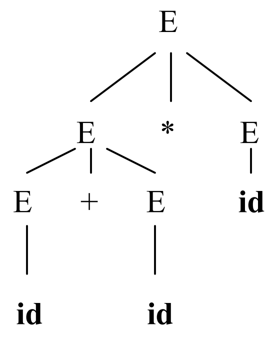
Example : Given grammar G : $E \rightarrow E+E \mid E*E \mid ( E ) \mid - E \mid id$

The sentence id+id*id has the following two distinct leftmost derivations:

| | |
|---|---|
| $E \rightarrow E+ E$ | $E \rightarrow E* E$ |
| $E \rightarrow id + E$ | $E \rightarrow E + E * E$ |
| $E \rightarrow id + E * E$ | $E \rightarrow id + E * E$ |
| $E \rightarrow id + id * E$ | $E \rightarrow id + id * E$ |
| $E \rightarrow id + id * id$ | $E \rightarrow id + id * id$ |

The two corresponding parse trees are :





**WRITING A GRAMMAR**

There are four categories in writing a grammar :

1. Regular Expression Vs Context Free Grammar
2. Eliminating ambiguous grammar.
3. Eliminating left-recursion
4. Left-factoring.

Each parsing method can handle grammars only of a certain form hence, the initial grammar may have to be rewritten to make it parsable.

**Regular Expressions vs. Context-Free Grammars:**

| REGULAR EXPRESSION | CONTEXT-FREE GRAMMAR |
|---|---|
| It is used to describe the tokens of programming languages. | It consists of a quadruple where S → start symbol, P → production, T → terminal, V → variable or non- terminal. |
| It is used to check whether the given input is valid or not using **transition diagram.** | It is used to check whether the given input is valid or not using **derivation**. |
| The transition diagram has set of states and edges. | The context-free grammar has set of productions. |
| It has no start symbol. | It has start symbol. |
| It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth. | It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on. |

- The lexical rules of a language are simple and RE is used to describe them.
- Regular expressions provide a more concise and easier to understand notation for tokens than grammars.
- Efficient lexical analyzers can be constructed automatically from RE than from grammars.
- Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end into two manageable-sized components.

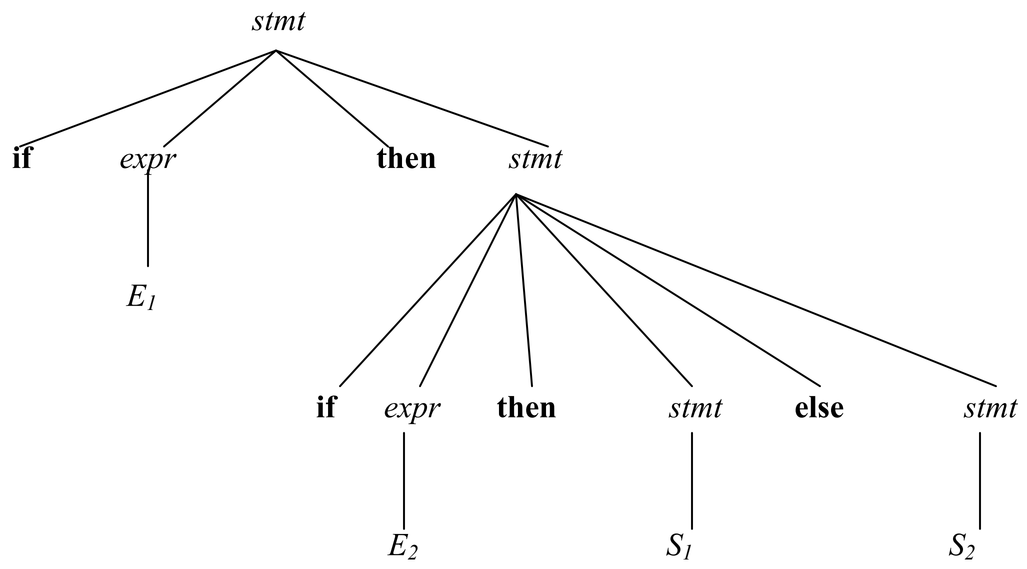
**Eliminating ambiguity:**

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

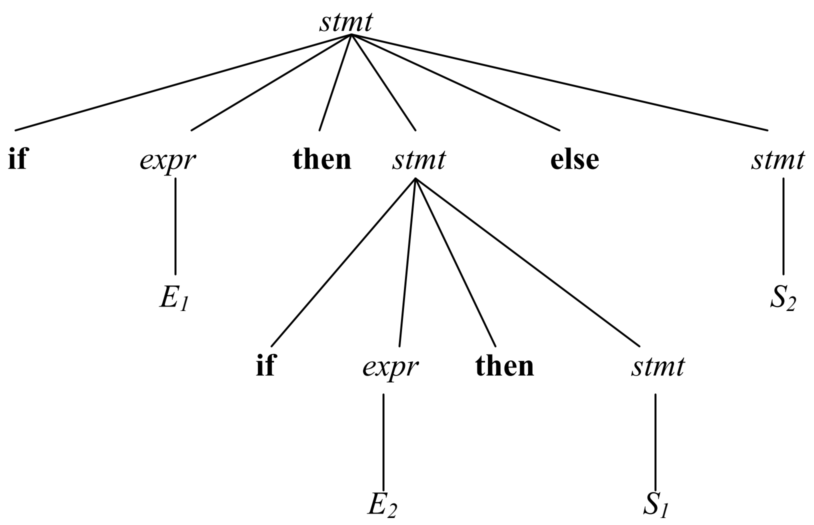
Consider this example, G: *stmt* → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**

This grammar is ambiguous since the string **if $E_1$ then if $E_2$ then $S_1$ else $S_2$** has the following two parse trees for leftmost derivation :

1.



2.



To eliminate ambiguity, the following grammar may be used:

*stmt* → *matched_stmt* | *unmatched_stmt*

*matched_stmt* → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other**

*unmatched_stmt* → **if** *expr* **then** *stmt* | **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

**Eliminating Left Recursion:**

A grammar is said to be *left recursive* if it has a non-terminal *A* such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

**If there is a production $A \rightarrow A\alpha \mid \beta$ it can be replaced with a sequence of two productions**

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

without changing the set of strings derivable from A.

**Example** : Consider the following grammar for arithmetic expressions:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

First eliminate the left recursion for E as

$E \rightarrow TE'$

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

Then eliminate for T as

$T \rightarrow FT'$

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

Thus the obtained grammar after eliminating left recursion is

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

**Algorithm to eliminate left recursion:**

```
1. Arrange the non-terminals in some order $A_1, A_2 \ldots A_n$.
2. for $i$ := 1 to $n$ do begin
       for $j$ := 1 to $i$-1 do begin
           replace each production of the form $A_i \rightarrow A_j \gamma$
               by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$
               where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current $A_j$-productions;
       end
       eliminate the immediate left recursion among the $A_i$-productions
   end
```

**Left factoring:**

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

**If there is any production $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$, it can be rewritten as**

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

Consider the grammar , G : $S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b$

Left factored, this grammar becomes

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

**PARSING**

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

**Parse tree:**

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

**Types of parsing:**

1. Top down parsing
2. Bottom up parsing

- Top–down parsing : A parser can start with the start symbol and try to transform it to the input string.
  Example : LL Parsers.
- Bottom–up parsing : A parser can start with input and attempt to rewrite it into the start symbol.
  Example : LR Parsers.

**TOP-DOWN PARSING**

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

**Types of top-down parsing :**

1. Recursive descent parsing
2. Predictive parsing

**1. RECURSIVE DESCENT PARSING**

- Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input**.**
- This parsing method may involve **backtracking**, that is, making repeated scans of the input.

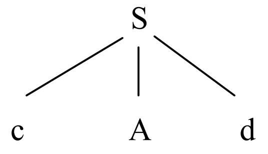
**Example for backtracking :**

Consider the grammar G : S → cAd
A → ab | a
and the input string w=cad.

The parse tree can be constructed using the following top-down approach :

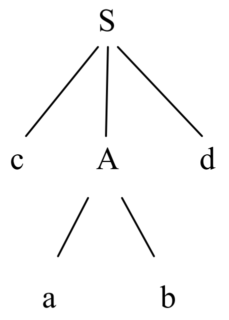
**Step1:**

Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



**Step2:**

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.



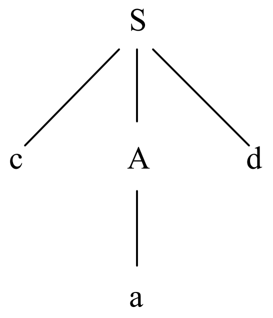
**Step3:**

The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'**.** But the third leaf of tree is b which does not match with the input symbol **d.**

Hence discard the chosen production and reset the pointer to second position. This is called **backtracking.**

**Step4:**

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

**<u>Example for recursive decent parsing</u>:**

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop. Hence, **elimination of left-recursion** must be done before parsing.

Consider the grammar for arithmetic expressions

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

After eliminating the left-recursion the grammar becomes,

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Now we can write the procedure for grammar as follows:

**<u>Recursive procedure</u>**:

```
Procedure E()
begin
        T( );
        EPRIME( );
end
```

```
Procedure EPRIME( )
begin
        If input_symbol=’+’ then
        ADVANCE( );
        T( );
        EPRIME( );
end

Procedure T( )
begin
        F( );
        TPRIME( );
end

Procedure TPRIME( )
begin
        If input_symbol=’*’ then
        ADVANCE( );
        F( );
        TPRIME( );
end

Procedure  F( )
begin
        If input-symbol=’id’ then
        ADVANCE( );
        else if input-symbol=’(‘ then
        ADVANCE( );
        E( );
        else if input-symbol=’)’ then
        ADVANCE( );
end

else  ERROR( );
```

**Stack implementation:**

To recognize input **id+id*id :**

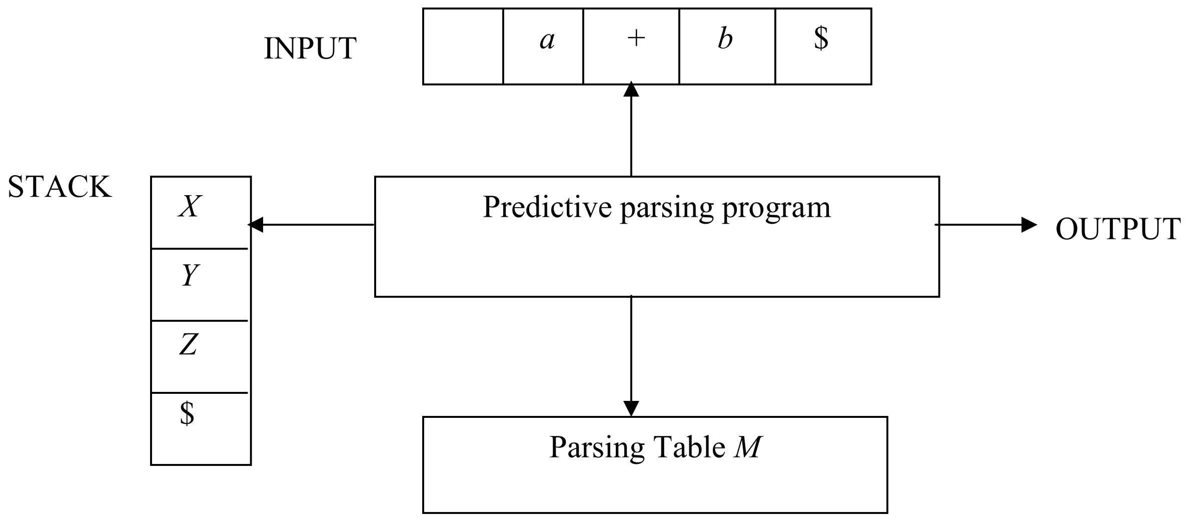
| PROCEDURE | INPUT STRING |
|---|---|
| E( ) | **id**+id*id |
| T( ) | **id**+id*id |
| F( ) | **id**+id*id |
| ADVANCE( ) | id+id*id |

| | |
|---|---|
| TPRIME( ) | id+id*id |
| EPRIME( ) | id+id*id |
| ADVANCE( ) | id+**id***id |
| T( ) | id+**id***id |
| F( ) | id+**id***id |
| ADVANCE( ) | id+id*id |
| TPRIME( ) | id+id*id |
| ADVANCE( ) | id+id* **id** |
| F( ) | id+id* **id** |
| ADVANCE( ) | id+id* **id** |
| TPRIME( ) | id+id* **id** |

**2. PREDICTIVE PARSING**

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

**Non-recursive predictive parser**



The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

**Input buffer:**

It consists of strings to be parsed, followed by $ to indicate the end of the input string.

**Stack:**

It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

**Parsing table:**

It is a two-dimensional array $M[A, a]$, where ***'A'*** is a non-terminal and ***'a'*** is a terminal.

**Predictive parsing program:**

The parser is controlled by a program that considers $X$, the symbol on top of stack, and $a$, the current input symbol. These two symbols determine the parser action. There are three possibilities:

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, 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 either be an $X$-production of the grammar or an error entry.
   If $M[X, a] = \{X \rightarrow UVW\}$,the parser replaces $X$ on top of the stack by $WVU$.
   If $M[X, a]$ = **error**, the parser calls an error recovery routine.

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

```
set ip to point to the first symbol of $w\$$;
repeat
    let $X$ be the top stack symbol and $a$ the symbol pointed to by ip;
    if $X$ is a terminal or $ then
        if $X = a$ then
            pop $X$ from the stack and advance ip
        else error()
    else            /* $X$ is a non-terminal */
        if $M[X, a] = X \rightarrow Y_1 Y_2 \ldots Y_k$ then begin
```

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

**<u>Predictive parsing table construction:</u>**

The construction of a predictive parser is aided by two functions associated with a grammar G :

1. FIRST

2. FOLLOW

**Rules for first( ):**

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

2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST($X$).

3. If $X$ is non-terminal and X → $a\alpha$ is a production then add $a$ to FIRST(X).

4. If X is non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, 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, $Y_1, \ldots . Y_{i-1} \Rightarrow \varepsilon$. If ε is in FIRST($Y_j$) for all j=1,2,..,k, then add ε to FIRST($X$).

**Rules for follow( ):**

1. If $S$ is a start symbol, then FOLLOW($S$) contains $.

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

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

**<u>Algorithm for construction of predictive parsing table:</u>**

**Input** : Grammar $G$

**Output** : Parsing table $M$

**Method** :

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 → α 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**.

**Example:**

Consider the following grammar :

E → E+T | T
T → T*F | F
F → (E) | id

After eliminating left-recursion the grammar is

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

**First( ) :**

FIRST(E) = { ( , id}

FIRST(E') ={+ , ε }

FIRST(T) = { ( , id}

FIRST(T') = {*, ε }

FIRST(F) = { ( , id }

**Follow( ):**

FOLLOW(E) = { $, ) }

FOLLOW(E') = { $, ) }

FOLLOW(T) = { +, $, ) }

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

FOLLOW(F) = {+, * , $ , ) }

**Predictive parsing table :**

| NON-TERMINAL | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E → TE' | | | E → TE' | | |
| E' | | E' → +TE' | | | E' → ε | E'→ ε |
| T | T → FT' | | | T → FT' | | |
| T' | | T'→ ε | T'→ *FT' | | T' → ε | T' → ε |
| F | F → id | | | F → (E) | | |

**Stack implementation:**

| stack | Input | Output |
|---|---|---|
| $E | id+id*id $ | |
| $E'T | id+id*id $ | $E \rightarrow TE'$ |
| $E'T'F | id+id*id $ | $T \rightarrow FT'$ |
| $E'T'id | id+id*id $ | $F \rightarrow id$ |
| $E'T' | +id*id $ | |
| $E' | +id*id $ | $T' \rightarrow \varepsilon$ |
| $E'T+ | +id*id $ | $E' \rightarrow +TE'$ |
| $E'T | id*id $ | |
| $E'T'F | id*id $ | $T \rightarrow FT'$ |
| $E'T'id | id*id $ | $F \rightarrow id$ |
| $E'T' | *id $ | |
| $E'T'F* | *id $ | $T' \rightarrow *FT'$ |
| $E'T'F | id $ | |
| $E'T'id | id $ | $F \rightarrow id$ |
| $E'T' | $ | |
| $E' | $ | $T' \rightarrow \varepsilon$ |
| $ | $ | $E' \rightarrow \varepsilon$ |

**<u>LL(1) grammar</u>:**

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Consider this following grammar:

$S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b$

After eliminating left factoring, we have

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

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

FIRST(S) = { i, a }

FIRST(S') = {e, ε }

FIRST(E) = { b}

FOLLOW(S) = { $ ,e }

FOLLOW(S’) = { $ ,e }

FOLLOW(E) = {t}

**Parsing table:**

| **NON-TERMINAL** | a | b | e | i | t | $ |
|---|---|---|---|---|---|---|
| S | S → a | | | S → iEtSS’ | | |
| S’ | | | S’ → eS<br>S’ → ε | | | S’ → ε |
| E | | E → b | | | | |

Since there are more than one production, the grammar is not LL(1) grammar.

**Implementation of predictive parser:**

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table.