# 18CSC304J – COMPILER DESIGN
# UNIT - II

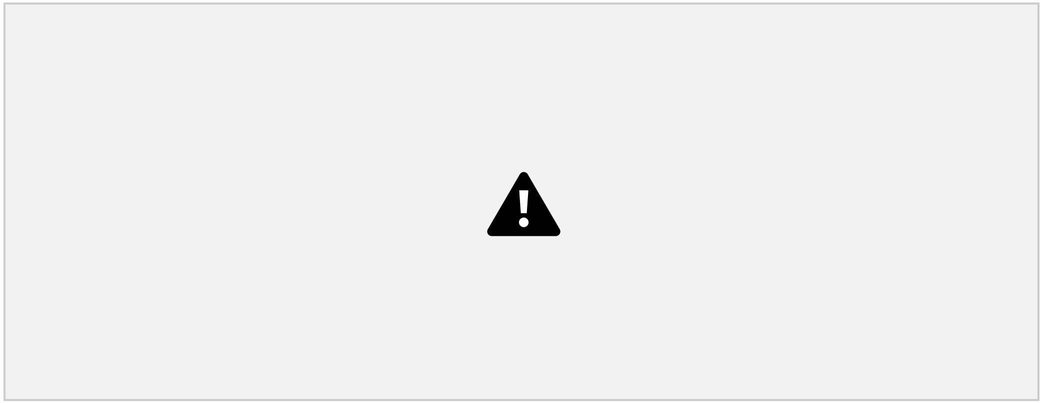
**Topics:** Syntax Analysis Definition - Role of parser, Lexical versus Syntactic Analysis, Representative Grammars, Syntax Error Handling, Elimination of Ambiguity, Left Recursion, Left Factoring, Top-down parsing, Recursive Descent Parsing, Back tracking, Computation of FIRST, Problems related to FIRST, Computation of FOLLOW, Problems related to FOLLOW, Construction of a predictive parsing table, Predictive Parsers LL(1) Grammars, Transition Diagrams for Predictive Parsers, Error Recovery in Predictive Parsing, Predictive Parsing Algorithm, Non Recursive Predictive Parser.

**Textbook:** Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "*Compilers: Principles, Techniques, and Tools*" Addison-Wesley, 1986.

## INTRODUCTION:

When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scans the input and divides it into tokens) to target code generation. A lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis.

Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a Parse tree or Syntax tree. The parse tree is constructed by using the pre-defined grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax, if not, error is reported by syntax analyzer.



## NEED AND ROLE OF SYNTAX ANALYSER:

In the process of compilation, the parser and scanner work together. That means, when parser requires string of tokens it invokes scanner. In turn the scanner supplies tokens to parser. Checks if the code is valid grammatically. The syntactical analyser helps you to apply rules to the code. Helps you to detect all types of syntactic errors. The parser must reject invalid texts by reporting syntax errors. It is also necessary that the parser should recover from commonly occurring errors to continue and find further errors in the code. Because error should not affect compilation of "correct" programs.

## WHY LEXICAL AND SYNTAX ANALYZER ARE SEPARATED OUT?

The lexical analyzer scans the input program and collects the tokens from it. On the other hand, parser builds a parser tree using these tokens. These are two important activities and these activities are independently carried out by these two phases. Separating out these two phases has two advantages:

- It accelerates the process of compilation.
- The errors in the source input can be identified precisely.

## ERROR HANDLING IN COMPILER DESIGN:

The tasks of the **Error Handling** process are to detect each error, report it to the user, and then make some recovery strategy and implement them to handle the error. During this whole process processing time of the program should not be slow.

Functions of Error Handler:

- Error Detection
- Error Report
- Error Recovery

Error handler=Error Detection + Error Report + Error Recovery.

An **Error** is the blank entries in the symbol table. Errors in the program should be detected and reported by the parser. Whenever an error occurs, the parser can handle it and continue to parse the rest of the input. Although the parser is mostly responsible for checking for errors, errors may occur at various stages of the compilation process.

**Types** or **Sources of Error:**

1. A **run-time error** is an error that takes place during the execution of a program and usually happens because of adverse system parameters or invalid input data. The lack of sufficient memory to run an application or a memory conflict with another program and logical error is an example of this. Logic errors occur when executed code does not produce the expected result. Logic errors are best handled by meticulous program debugging.
2. **Compile-time errors** rise at compile-time, before the execution of the program. Syntax error or missing file reference that prevents the program from successfully compiling is an example of this.

**Classification of Compile-time error:**

1. **Lexical**: This includes misspellings of identifiers, keywords or operators. Ex: fro instead of for
2. **Syntactical**: a missing semicolon or unbalanced parenthesis. Ex: {, (, ;
3. **Semantical**: incompatible value assignment or type mismatches between operator and operand. Ex: int i=”hello”, “hai” – “hello”
4. **Logical**: code not reachable, infinite loop. Ex: while (1) {  // body of the loop..  }

**Error Recovery:**
The basic requirement for the compiler is to simply stop and issue a message, and cease compilation. There are some common recovery methods that are as follows.
**Panic mode recovery:**

· Insert one character. Ex: vod -> void
· Delete one character. Ex: void1 -> void
· Replace a character by another character. Ex: printt -> printf
· Transpose two adjacent characters. Ex: fi -> if

## CONTEXT FREE GRAMMAR:

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language. Context free grammar G can be defined by four tuples as:

G= (V, T, P, S) where,

**G** describes the grammar

**T** describes a finite set of terminal symbols.

**Terminals symbols:**

· Lower-case letters in the alphabet such as a, b, c,
· Operator symbols such as +, -, *, etc.
· Punctuation symbols such as parentheses, hash, comma
· 0, 1, …, 9 digits
· Boldface strings like id or if, anything which represents a single terminal symbol

**V** describes a finite set of non-terminal symbols.

**Nonterminal symbols:**

· Upper-case letters such as A, B, C
· Lower-case italic names: the expression or some

**S** is the start symbol.

**P** describes a set of production rules.

**Non-terminal -> (V U T)***

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right-hand side of the production, until all non-terminal has been replaced by terminal symbols.

## DERIVATION:

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

### Left-most Derivation:

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

### Right-most Derivation:

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

**Example:**

**Consider the production rules:**

E → E + E
E → E * E
E → id

Input string: id + id * id

The left-most derivation is:

E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id

Notice that the left-most side non-terminal is always processed first.

The right-most derivation is:

E → E + E
E → E + E * E
E → E + E * id
E → E + id * id
E → id + id * id

## PARSE TREE:

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.

We take the left-most derivation of a + b * c

The left-most derivation is:

E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id

Step 1:

| | |
|---|---|
| E → E * E | |

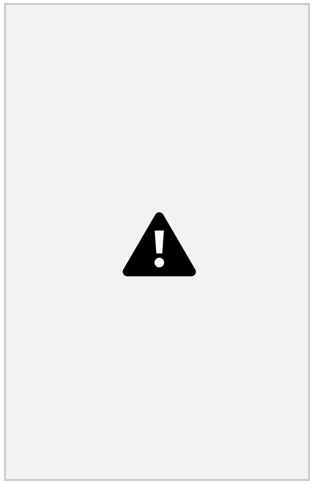
Step 2:

| | |
|---|---|
| E → E + E * E | |

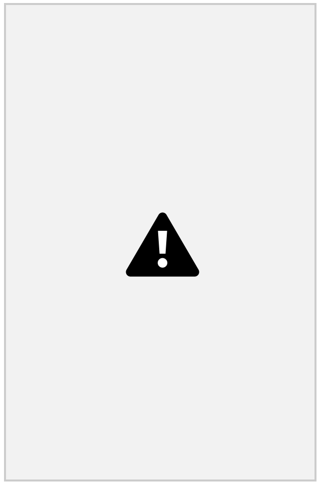
Step 3:

| | |
|---|---|
| E → id + E * E | |

Step 4:

| | |
|---|---|
| E → id + id * E |  |

Step 5:

| | |
|---|---|
| E → id + id * id |  |

In a parse tree:

- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

**CLASSIFICATION OF CONTEXT FREE GRAMMARS:**
**C**ontext **F**ree **G**rammars (CFG) can be classified on the basis of following two properties:
**1) Based on number of strings it generates.**

- If CFG is generating finite number of strings, then CFG is **Non-Recursive** (or the grammar is said to be Non-recursive grammar)
- If CFG can generate infinite number of strings then the grammar is said to be **Recursive** grammar

During Compilation, the parser uses the grammar of the language to make a parse tree (or derivation tree) out of the source code. The grammar used must be unambiguous. An ambiguous grammar must not be used for parsing.

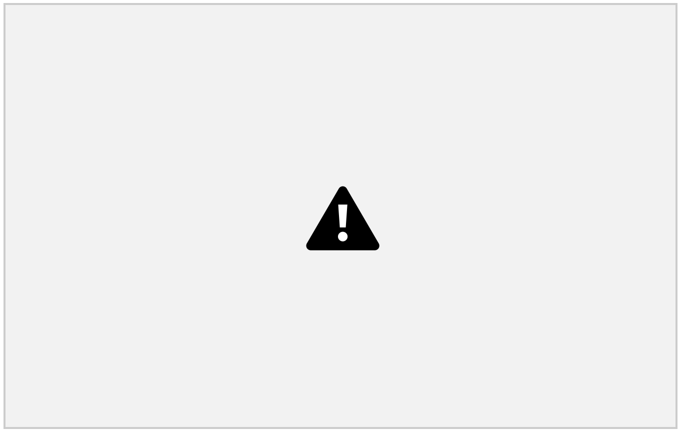
**2) Based on number of derivation trees.**

- If there is only 1 derivation tree then the CFG is **Unambiguous**.
- If there are more than 1 derivation tree, then the CFG is **ambiguous**.

**Ambiguous derivation tree:**
A CFG is said to ambiguous if there exists more than one derivation tree for the given input string i.e., more than one **L**eft **M**ost **D**erivation **T**ree (LMDT) or **R**ight **M**ost **D**erivation **T**ree (RMDT).

**Definition:** G = (V,T,P,S) is a CFG is said to be ambiguous if and only if there exist a string in T* that has more than one parse tree.
where V is a finite set of variables.
T is a finite set of terminals.
P is a finite set of productions of the form, A -> α, where A is a variable and α ∈ (V ∪ T)* S is a designated variable called the start symbol.

**Example based on number of derivation trees: (Ambiguous)**
Let us consider this grammar: **E -> E+E | id**
We can create 2 parse trees from this grammar to obtain a string **id+id+id**
The following are the 2 parse trees generated by left most derivation:



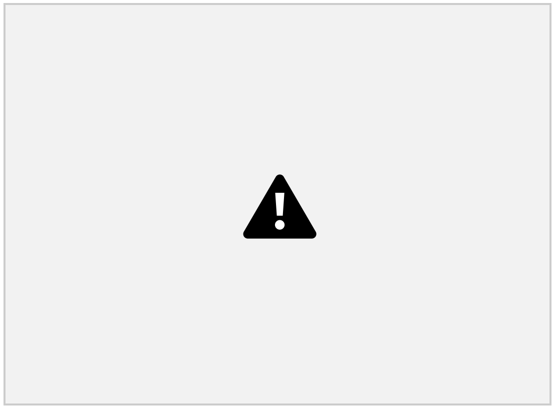
Both the above parse trees are derived from same grammar rules but both parse trees are different. Hence the grammar is ambiguous.

**Example based on number of derivation trees: (Unambiguous)**
Let us consider this grammar:

$S \rightarrow AB$

$A \rightarrow Aa \mid a$

$B \rightarrow b$



**Examples based on number of strings it generates**:
**Recursive Grammars:**
1) S->SaS

   S->b

The language (set of strings) generated by the above grammar is:
{b, bab, babab, …} which is infinite.

2) S-> Aa

   A->Ab | c

The language generated by the above grammar is:
{ca, cba, cbba …} which is infinite.

**Non-Recursive Grammars:**
   S->Aa

   A->b | c

The language generated by the above grammar is:
{ba, ca} which is finite.

## REMOVAL OF AMBIGUITY:

**Ambiguous vs Unambiguous Grammar:**
The grammars which have more than one derivation tree or parse tree are ambiguous grammars. This grammar is not parsed by any parsers. The grammars which have only one derivation tree or parse tree are called unambiguous grammars.

**Removal of Ambiguity:**
We can remove ambiguity solely on the basis of the following two properties:
**1. Precedence:**
If different operators are used, we will consider the precedence of the operators. The three important characteristics are:

- The level at which the production is present denotes the priority of the operator used.
- **The production at higher levels will have operators with less priority. In the parse tree, the nodes which are at top levels or close to the root node will contain the lower priority operators.**
- The production at lower levels will have operators with higher priority. In the parse tree, the nodes which are at lower levels or close to the leaf nodes will contain the higher priority operators.

**2. Associativity:**
If the same precedence operators are in production, then we will have to consider the associativity. The two important characteristics are:

- **If the associativity is left to right, then we have to prompt a left recursion in the production. The parse tree will also be left recursive and grow on the left side.**
  **+, -, *, / are left associative operators.**
- If the associativity is right to left, then we have to prompt the right recursion in the productions. The parse tree will also be right recursive and grow on the right side.
  ^ is a right associative operator.

**Example: 1 (Based on Associativity)**
Consider the ambiguous grammar
**E -> E - E | id**
Input string: **id – id - id**
The language in the grammar will contain {id, id-id, id-id-id, …}

Let's consider a single value of id = 3 to get more insights.
The result should be:
**3 – 3 - 3 = -3**
Since the same priority operators, we need to consider associativity which is left to right.

**Parse Tree:**
The parse tree which grows on the left side of the root will be the correct parse tree in order to make the grammar unambiguous.

**Example: 2 (Based on Precedence)**
Consider the grammar shown below, which has two different operators.
**E -> E + E | E * E | id**
Clearly, the above grammar is ambiguous as we can draw two parse trees for the input string **“id + id * id”** as shown below.

For the given input string:
**id + id * id**
**= 3 + 2 * 5          //  “*” has more priority than “+”**
**The correct answer is : (3+(2*5))=13**

The “+” having the least priority has to be at the upper level and has to wait for the result produced by the “*” operator which is at the lower level. So, the first parse tree is the correct one and gives the same result as expected.

## PARSING TECHNIQUES:

There are two parsing techniques, these parsing techniques work on the following principles:

- The parser scans the input string from left to right and identifies that the derivation is leftmost or rightmost.
- The parser makes use of production rules for choosing the appropriate derivation. The different parsing techniques use different approaches in selecting the appropriate rule for derivation. And finally, a parse tree is constructed.

Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase. A parser takes input in the form of sequence of tokens and produces output in the form of parse tree. Parsing is of two types: top-down parsing and bottom-up parsing.

**Top-down parsing:**
In the top-down parsing, the parsing starts from the start symbol and transform it into the input symbol. Parse Tree representation of input string "acdb" is as follows:
**Example:**

S -> aABe
A -> Abc | b
B -> d

Now, let’s consider the input to read and to construct a parse tree with top-down approach.
**Input: abbcde$**

- First, you can start with S -> a A B e and then you will see input string a in the beginning and e in the end.
- Now, you need to generate ~~a~~bbcd~~e~~ .
- Expand A-> Abc and Expand B-> d.
- Now, we have string like aAbcde and your input string is abbcde.
- Expand A->b.
- Final string, you will get **abbcde**.

Given below is the Diagram explanation for constructing top-down parse tree.

**Bottom-up parsing:**

Bottom-up parsing is used to construct a parse tree for an input string. In the bottom-up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse.

**Example:**

**Consider the production rule:**

E → T

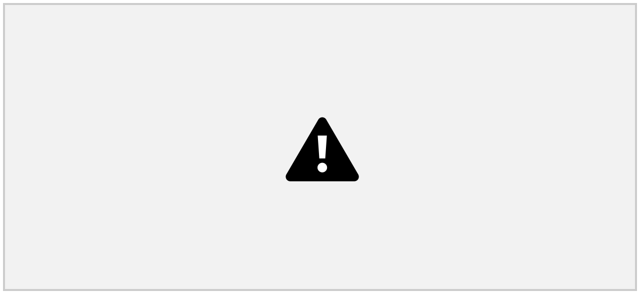
T → T * F

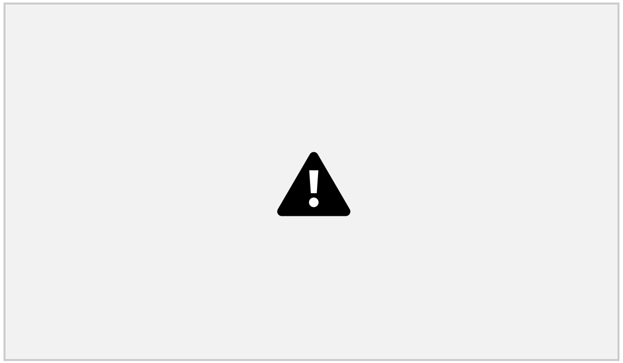
T → id

F → T

F → id

Parse Tree representation of input string "id * id" is as follows:





| **Top-down parser** | **Bottom-up parser** |
|---|---|
| Parse tree can be built from root to leaves. | Parse tree can be built from leaves to root. |
| This is simple to implement. | This is complex to implement. |
| It is applicable to small class of languages. | It is applicable to a broad class of languages. |
| Various parsing techniques are:<br>· Recursive descent parser<br>· Predictive parser | Various parsing techniques are:<br>· Shift reduce parser<br>· Operator precedence parser<br>· LR parser |

## RECURSIVE DESCENT PARSING:

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

**Basic steps for construction of Recursive Descent Parser:**

The R.H.S of the rule is directly converted into program code symbol by symbol.

- If the input symbol is a non-terminal, then a call to the procedure corresponding to the non-terminal is made.
- If the input symbol is terminal, then it is matched with the lookahead from input. The lookahead pointer has to be advanced on matching of the input symbol.
- If the production rule has many alternates, then all these alternates have to be combined into a single body of procedure.
- The parser should be activated by a procedure corresponding to the start symbol.

**Example:**

Consider the grammar:

E → i E'

E'→ + i E' | ε

```
int main ()
{
E();                    // E is a start symbol.
if (l == '$')           // if l == $, it represents the end of the string and “l” is lookahead.
Printf("Parsing Successful");
}

// Definition of E, as per the given production
E()
{
if (l == 'i')
{
match('i');
E'();
}
}

// Definition of E' as per the given production
E'()
{
if (l == '+') {
match('+');
match('i');
E'();
}           //The second condition of E'
else
return ();
}

// Match function
match (char t)
{
if (l == t)
{
l = getchar();          //Lookahead points to next token
}
else
printf("Error");
```

}

## BACKTRACKING:

**Step 1:** Keep a pointer just before the first symbol in the input string to be parsed
**Step 2:** Expand the start symbol
**Step 3:** Compare the terminal after the pointer with the unmatched leaf (from left to right, initially it is the first leaf).
Here there may be three cases,

- **The leaf is a non-terminal:** Then expand it by its production.
- **A matching terminal:** Advance the pointer by one position.
- **A non-matching terminal:** Do not advance pointer, but undo the current expansion/s and move pointer backwards basing on the matched symbols in the undone expansions and expand the non-terminal by its other alternative if it exists.

**Step 4:** Repeat step 2 until the entire parse tree is constructed or failure occurs.

**Example:**
Consider the grammar
$S \rightarrow cAd$
$A \rightarrow ab \mid a$
and parsing the string w = cad
Initially we maintain an input pointer to c in w and expand S by cAd as mentioned below

Now, the left most leaf c is matched with first symbol in w. So, we advance the input pointer to a, the second symbol in w and consider the next leaf
As the next leaf is a non-terminal A, expand it as displayed below

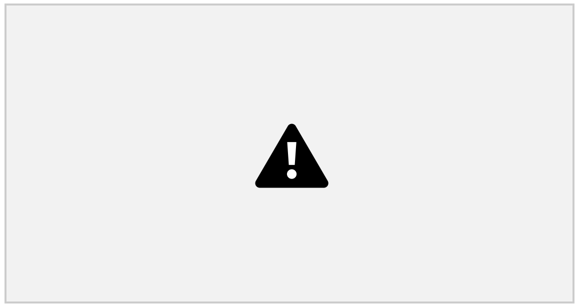
Now, we have a match for the second symbol that is a in w. So, we advance the input pointer to d. Now, there is a mismatch between leaf b and d. So, we go back to A to expand it by another alternative. In going back to A, we must reset the pointer in w to point again to a

Now, we have a match for the second symbol that is a in w and third symbol d in w. So, we halt and announce successful completion of parsing of string w.

## NON-BACKTRACKING:

## PREDICTIVE PARSER / LL(1) PARSER:

- Predictive parsers are top-down parsers.
- It is a type of recursive descent parser but with no backtracking.
- It can be implemented non recursively by using stack data structure.
- They can also be termed as LL(1) parser as it is constructed for a class of grammars called LL(1).
- The production to be applied for a non-terminal is decided based on the current input symbol.



- Left recursion is eliminated.
- Common prefixes are also eliminated (Left factoring).

## ELIMINATING LEFT RECURSION:

A grammar becomes left-recursive if it has any non-terminal ‘A’ whose derivation contains ‘A’ itself as the left-most symbol. Left-recursive grammar is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the Start symbol, which in itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.

A production of grammar is said to have **left recursion** if the leftmost variable of its RHS is same as variable of its LHS. A grammar containing a production having left recursion is called as Left Recursive Grammar.

## Example:

$A \Rightarrow A\alpha \mid \beta$
$S \Rightarrow A\alpha \mid \beta$
$A \Rightarrow Sd$

A top-down parser will first parse the A, which in-turn will yield a string consisting of A itself and the parser may go into a loop forever.

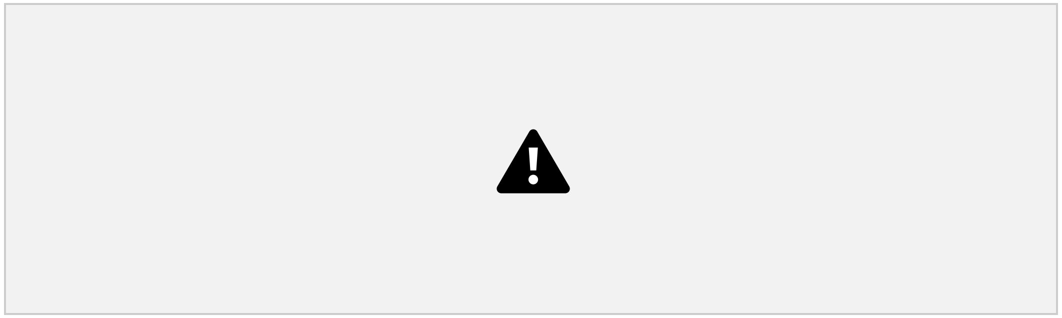
**Removal of Left Recursion:**
One way to remove left recursion is to use the following technique:
The production

$$A \rightarrow A\alpha \mid \beta$$

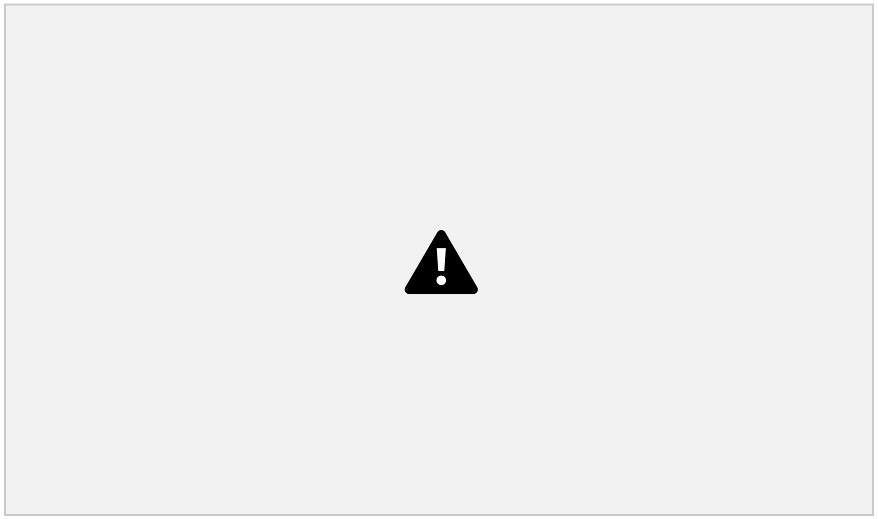
is converted into following productions

$$\mathbf{A \rightarrow \beta A'}$$
$$\mathbf{A' \rightarrow \alpha A' \mid \varepsilon}$$

This does not impact the strings derived from the grammar, but it removes immediate left recursion.



This type of recursion is also called **Immediate Left Recursion**.
In Left Recursive Grammar, expansion of A will generate Aα, Aαα, Aααα at each step, causing it to enter into an infinite loop.



**Example:**
Consider the Left Recursion from the Grammar.
E → E + T | T
T → T * F | F
F → (E) | id
Eliminate immediate left recursion from the Grammar.

**Solution:**

Comparing $E \rightarrow E + T \mid T$ with $A \rightarrow A\,\alpha \mid \beta$

$\therefore A = E,\ \alpha = +T,\ \beta = T$

$A \rightarrow A\,\alpha \mid \beta$ is changed to
$A \rightarrow \beta A'$ and
$A' \rightarrow \alpha\, A' \mid \varepsilon$

$A \rightarrow \beta A'$ means $E \rightarrow TE'$
$A' \rightarrow \alpha\, A' \mid \varepsilon$ means $E' \rightarrow +TE' \mid \varepsilon$

## LEFT FACTORING

If more than one grammar production rules have a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand. Basically, left factoring is used when it is not clear that which of the two alternatives is used to expand the non-terminal.

**Example:**
If a top-down parser encounters a production like
$A \Longrightarrow \alpha\beta \mid \alpha\boldsymbol{\gamma}$
Then it cannot determine which production to follow to parse the string as both productions are starting from the same terminal (or non-terminal). To remove this confusion, we use a technique called left factoring. Left factoring transforms the grammar to make it useful for top-down parsers. In this technique, we make one production for each common prefixes and the rest of the derivation is added by new productions.

**Example:**
The above productions can be written as
$A \Rightarrow \alpha A'$
$A' \Rightarrow \beta \mid \boldsymbol{\gamma}$
Now the parser has only one production per prefix which makes it easier to take decisions.

Consider the following grammar:

A → aAB | aBc | aAc

**Solution:**

**Step 1:**

A → aA’

A’ → AB | Bc | Ac

Again, this is a grammar with common prefixes.

**Step 2:**

A → aA’

A’ → AB | Bc |Ac // a grammar with common prefixes

A’ → AB | Ac

A’ → AD | Bc

D → B | c

This is a left factored grammar.

## COMPUTATION OF FIRST:

Remove left recursion and left factoring from the production grammar.

**Rules For Calculating First Function:**

1. If X is a terminal, then First(X) is just {X}.
2. If there is a Production X → ε then add ε to First(X)
3. If there is a Production X → $Y_1Y_2$ ...$Y_k$ then add first ($Y_1Y_2$ ...$Y_k$) to first(X)
   If First ($Y_1Y_2$ ...$Y_k$) then, choose
   a. First($Y_1$) doesn't contain ε, First(X)=First($Y_1$),.
   b. First($Y_1$) does contain ε, include everything in First($Y_1$) (except for ε) as well as everything in First($Y_2$),.
   c. If First($Y_1$) First($Y_2$) ... First ($Y_k$) all contain ε then add ε to First ($Y_1Y_2$ ...$Y_k$) as well.

## COMPUTATION OF FOLLOW:

**Rules For Calculating Follow Function:**

1. First put $ (the end of input marker) in Follow(S), where S is the start symbol.
2. If there is a production A → αBβ, β≠∈ then everything in FIRST(β) except for ε is placed in FOLLOW(B).
   Follow (B) = First (β) - ε
3. If there is a production A → αB, then everything in FOLLOW(A) is in FOLLOW(B)
   Follow (B) = Follow (A)
4. If there is a production A → αBβ, where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)
   Follow (B) = Follow (A)

## PROBLEMS RELATED TO FIRST AND FOLLOW:

Consider the expression gramma, repeated below:

E → T E'
E' → +T E' | ∈
T → F T'
T'→ * F T' | ∈
F → ( E ) | id

**Solution:**

**First Functions:**

First(E) = First (TE') = First (T) = First (FT') = First (F) = {( , id} //Rule 3a and 1
First(T) = First (FT') = First (F) = {( , id} //Rule 3a
First(F) = {( , id} // Rule 1
First(E') = {+, ∈} //Rule 1 and 2
First(T') = {*, ∈} //Rule 1 and 2

**Follow Functions:**

1. First put $ (the end of input marker) in Follow(S), where S is the start symbol.
E → T E' //E is the start symbol
Follow(E) = {$

2. If there is a production A → αBβ, β≠∈ then everything in FIRST(β) except for ε is placed in FOLLOW(B).
Follow (B) = First (β) - ε
E → T E'
A → αBβ, Where α = 1, B = T, β = E'

So, check First (β) has ε. It has.
First(E’) = {+, ∈}
Follow (T) = First(E’) – ε
= {+

E’ →+T E’ | ∈
A → αBβ, Where α = +, B = T, β = E’, not applicable for second part of the production rule.
So, check First (β) has ε. It has.
First(E’) = {+, ∈}
Follow (T) = First(E’) – ε
= {+

T → F T’
A → αBβ, Where α = 1, B = F, β = T’.
So, check First (β) has ε. It has.
First(T’) = {*, ∈}
Follow (F) = First(T’) – ε
= {*

T’→ * F T’ | ∈
A → αBβ, Where α = *, B = F, β = T’, not applicable for second part of the production rule.
So, check First (β) has ε. It has.
First(T’) = {*, ∈}
Follow (F) = First(T’) – ε
= {*

F → ( E ) | id
A → αBβ, Where α = (, B = E, β = ), not applicable for second part of the production rule.
So, check First (β).
First ( ) ) = { )
Follow (E) = { )
Follow(E) = {$, ) // Already found Follow (E)

3. If there is a production A → αB, then everything in FOLLOW(A) is in FOLLOW(B)
E → T E’
A → αB, Where α = T, B = E’
Follow (B) = Follow (A)
Follow(E’) = Follow (E)
Follow(E) = {$, )

Follow (E') = { \$, )

E' →+T E' | ∈
A → αB, Where α = +T, B = E'
Follow(E') = Follow (E') //Both are same
Follow (E') = { \$, )

T → F T'
A → αB, Where α = F, B = T'
Follow(T') = Follow (T)
Follow (T) = {+
Follow (T') = {+

T'→ * F T' | ∈
A → αB, Where α = *F, B = T'
Follow(T') = Follow (T')
Follow (T') = {+

F → ( E ) | id
A → αB, cannot be applicable, because B should have non terminal symbol.

4. If there is a production A → αBβ, where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)
If FIRST(β) has ε, then Follow (B) = Follow (A)

E → T E'
A → αBβ, where α = 1, B = T, β = E', check First (β) has ∈, it has.
First (E') = {+, ∈}, then
Follow (T) = Follow (E)
Follow (E) = {\$, )
Follow (T) = {+, \$, )

E' →+T E' | ∈
A → αBβ, where α = +, B = T, β = E'
First (E') = {+, ∈}, then
Follow (T) = Follow (E')
Follow (E') = { \$, )
Follow (T) = {+, \$, )

T → F T'

A → αBβ, where α = 1, B = F, β = T'
First (T') = {*, ∈}, then
Follow (F) = Follow (T)
Follow (T) = {+, $, )
Follow (F) = {*,+, $, )

T'→ * F T' | ∈
A → αBβ, where α = *, B = F, β = T'
First (T') = {*, ∈}, then
Follow (F) = Follow (T')
Follow (T') = {+
Follow (F) = {*, +, $, )

F → ( E ) | id
A → αBβ, where α = (, B = E, β = ), First(β) = First ( ) ) = ), **β has terminal symbol, does not has ∈**

So, Finally
Follow(E) = Follow(E') = {) , $}
FOLLOW (T) = FOLLOW(T′) = {+, ), $}
Follow(F) = {+, *, ), $}

## CONSTRUCTION OF A PREDICTIVE PARSING TABLE:

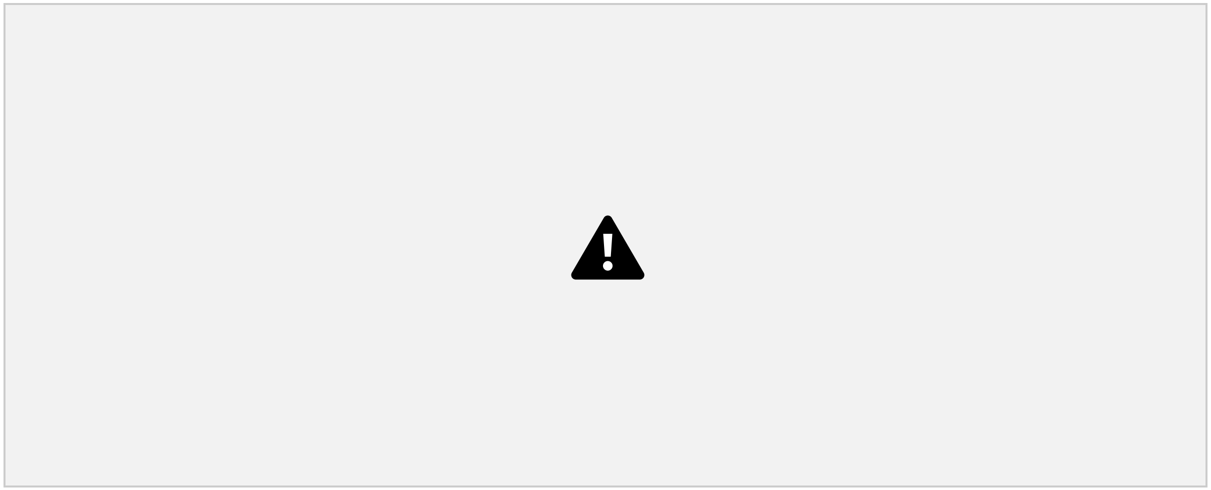
Consider the following grammar & also check whether string id + id * id is accepted or not.
E → TE′
E′ → +TE′ | ε
T → FT′
T′ → FT′ | ε
F → (E) | id

**Step 1: Elimination of Left Recursion & perform Left Factoring**
There is no left recursion and left factoring in Grammar.

**Step 2: Computation of FIRST**
FIRST(E) = FIRST(T) = FIRST(F) = {(, id}
FIRST (E′) = {+, ε}
FIRST (T′) = {*, ε}

**Step 3: Computation of FOLLOW**
FOLLOW (E) = FOLLOW(E′) = {), $}
FOLLOW (T) = FOLLOW(T′) = {+, ), $}
FOLLOW (F) = {+, *, ), $}

**Step 4: Construction of Predictive Parsing Table**
Create the table, i.e., write all non-terminals row-wise & all terminal column-wise.



**INPUT:** Grammar G
**OUTPUT:** Parsing table M
**METHOD:**

1. For each production A → α of the grammar, do step 2 and 3.
2. For each terminal a in First (α), Add A→ α to M[A,a].
3. If ε is in First (α), Add A→ ε to M [A, b] for each terminal b in Follow(A). If ε is in First (α) and $ is in Follow (A), add A→ α to M [A, $]
4. Make each of the undefined entry of M as error.

Now, fill the table by applying rules from the construction of the Predictive Parsing Table.

**E → TE′**
Comparing E → TE′ with A → α    //Rule 1: For each production A → α of the grammar

| | |
|---|---|
| E → | TE′ |

|  |  |
|---|---|
| A → | α |

∴ A = E, α = TE′
∴ FIRST(α) = FIRST(TE′) = FIRST(T) = {(, id}　　　//As FIRST(T) does not contain ε, need not to consider E’
∴ ADD E → TE′ to M[E, (] and M[E, id]　　//Rule 2: For each terminal a in First (α), Add A→ α to M[A,a].
∴ write E → TE′ in front of Row (E) and Columns {(, id}

|  | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E → TE′ |  |  | E → TE′ |  |  |

**E′ → +TE′ | ε**
Comparing it with A → α

|  |  |
|---|---|
| E’ → | +TE′ |
| A → | α |

∴ A = E′, α = +TE′
∴ FIRST(α) = FIRST(+TE′) = {+}
∴ ADD E’ → +TE′ to M[E′, +]
∴ write production E′ → +TE′ in front of Row (E′) and Column (+)

|  | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| E’ |  | E′ → +TE′ |  |  |  |  |

**E′ → ε**
Comparing it with A → α

|  |  |
|---|---|
| E’ → | ε |
| A → | α |

∴ α = ε
FIRST(α) = {ε}　　　//Rule 3: If ε is in First (α), Add A→ ε to M [A, b] for each terminal b in Follow(A). If ε is in First (α) and $ is in Follow (A), add A→ α to M [A, $]

Find FOLLOW (E′) = { ), $}
∴ ADD Production E′ ⟶ ε to M[E′, )] and M[E′, $]
∴ write E′ ⟶ ε in front of Row (E′) and Column {$, )}

| | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| E′ | | E′ ⟶ +TE′ | | | E′ ⟶ ε | E′ ⟶ ε |

**T ⟶ FT′**
Comparing it with A ⟶ α

| | |
|---|---|
| T ⟶ | FT′ |
| A ⟶ | α |

∴ A = T, α = FT′
∴ FIRST(α) = FIRST (FT′) = FIRST (F) = {(, id}        //As FIRST(F) does not contain ε, need not to consider T’
∴ ADD Production T ⟶ FT′ to M[T, (] and M[T, id]
∴ write T ⟶ FT′ in front of Row (T)and Column {(, id}

**T′ ⟶*FT′| ε**
Comparing it with A ⟶ α

| | |
|---|---|
| T ⟶ | *FT′ |
| A ⟶ | α |

∴ FIRST(α) = FIRST (* FT′) = {*}
∴ ADD Production T ⟶ *FT′ to M[T′,*]
∴ write T′ ⟶* FT′ in front of Row (T′) and Column {*}

**T′ ⟶ ε**
Comparing it with A ⟶ α

| | |
|---|---|
| T′ ⟶ | ε |
| A ⟶ | α |

∴ A = T′
α = ε
∴ FIRST(α) = FIRST {ε} = {ε}

Find FOLLOW (A) = FOLLOW (T′) = {+, ), $}
∴ ADD T′ ⟶ ε to M[T′, +], M[T′, )] and M[T′, $]
∴ write T′ ⟶ ε in front of Row (T′)and Column {+, ), $}

**F ⟶(E)**
Comparing it with A ⟶ α

|  |  |
|---|---|
| F ⟶ | (E) |
| A ⟶ | A |

∴ A = F, α = (E)
∴ FIRST(α) = FIRST ((E)) = FIRST (( ) = {(}
∴ ADD F ⟶ (E) to M[F, (]
∴ write F ⟶ (E) in front of Row (F)and Column (( )

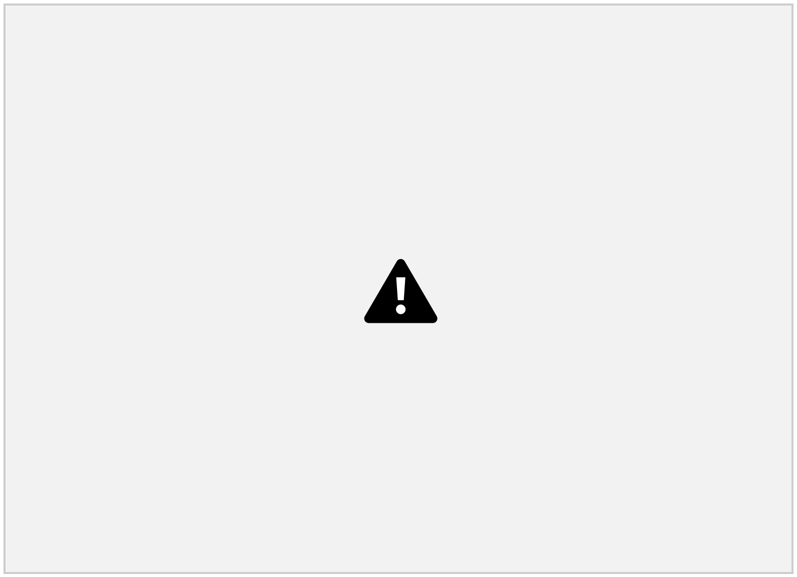
**F ⟶ id**
Comparing it with A ⟶ α

|  |  |
|---|---|
| F ⟶ | Id |
| A ⟶ | A |

∴ FIRST(α) = FIRST (id) = {id}
∴ ADD F ⟶ id to M[F, id]
∴ write F ⟶ id in front of Row (F)and Column (id)
Combining all statements will generate the following Predictive or LL (1) Parsing Table.

|  | 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) |  |  |

**Step 5: Checking Acceptance of String id + id * id using Predictive Parsing Program (NON-RECURSIVE PREDICTIVE PARSER)**
A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing. It is also called as LL(1) parsing table technique since we would be building a table for string to be parsed. It has capability to predict which production is to be used to replace input string. To accomplish its tasks, predictive parser uses a look-ahead pointer, which points to next input symbols. To make parser back-tracking free, predictive parser puts some constraints on grammar and accepts only a class of grammar known as LL(k) grammar. Predictive parsing uses a stack and a parsing table to parse input and generate a parse tree. Both stack and input contain an end symbol $ to denote that stack is empty and input is consumed. The parser refers to parsing table to take any decision on input and stack element combination. There might be instances where there is no production matching input string, making parsing procedure to fail.
Initially, the stack will contain the starting symbol E and $ at the bottom of the stack. Input Buffer will contain a string attached with $ at the right end.
If the top of stack = Current Input Symbol, then symbol from the top of the stack will be popped, and also input pointer advances or reads next symbol.
The sequence of Moves by Predictive Parser.



**Input:** A input string 'w' and a parsing table('M') for grammar G.
**Output:** If w is in L(G), an LMD of w; otherwise, an error indication.
**Method:**
Set input pointer to point to the first symbol of the string $
Repeat

begin
let X be the symbol pointed by the stack pointer, and a is the symbol pointed to by input pointer;

```
if X is a terminal or $ then
        if X=a then
        pop X from the stack and increment the input pointer;
        else error()
        end if
else /*if X is a non-terminal */
        if M[X,a] = X - > y1, y2…yk (production rule) then
        begin
        pop X from the stack;
        push yk, yk-1…y2, y1 onto the stack, with Y1 on top;
        output the production X- > y1, y2…yk;
        end
        else error()
        end if
end if
until X=$ /* stack is empty */
```

| Stack | Input | Action |
|---|---|---|
| $E (Start symbol) | id + id * id $ (Input string) | |
| $E | id + id * id $ | E → TE′ |
| $E′T | id + id * id $ | T → FT′ |
| $E′T′F | id + id * id $ | F → id |
| $E′T′id | id + id * id $ | Remove id |
| $E′T′ | + id * id $ | T′ → ε |
| $E′ | + id * id $ | E′ → +TE′ |
| $E′T + | + id * id $ | Remove + |

| | | |
|---|---|---|
| $E′T | id * id $ | T → FT′ |
| $E′T′F | id * id $ | F → id |
| $E′T′id | id * id $ | Remove id |
| $E′T′ | * id $ | T′ →* FT′ |
| $E′T′F * | * id $ | Remove * |
| $E′T′F | id $ | F → id |
| $E′T′id | id $ | Remove id |
| $E′T′ | $ | T′ → ε |
| $E′ | $ | E′ → ε |
| $ | $ | Accept |

## TRANSITION DIAGRAMS FOR PREDICTIVE PARSERS:

Consider the grammar,
E → E + T | T
T → T * F | F
F → (E) | id

The above grammar has left recursion. Eliminate it. And has ambiguity. Make it unambiguous.
Now, the grammar
E → TE′
E′ → +TE′ | ε
T → FT′
T′ →*FT′ | ε
F → (E) | id

## ERROR RECOVERY IN PREDICTIVE PARSING:

Recovery in non-recursive predictive parser is easier than in a recursive parser. An error occurs in predictive parsing when,

The error recovery in predictive parsing uses the following two methods:

- Panic Mode Recovery:
  - The terminal on the top of the stack does not match the current input symbol or
  - A non-terminal A is on the top of the stack with a as the current input symbol and the entry M[A, a] is empty.
- Phrase Level Recovery:
  - Carefully filling in the blank entries about what to do, whatever may be the action, care should be taken that it should not lead the parser into infinite loop.

## **DIFFERENCE BETWEEN RECURSIVE PREDICTIVE DESCENT PARSER AND NON-RECURSIVE PREDICTIVE DESCENT PARSER:**

| **Recursive Predictive Descent Parser** | **Non-Recursive Predictive Descent Parser** |
|---|---|
| It is a technique which may or may not require backtracking process. | It is a technique that does not require any kind of backtracking. |
| It uses procedures for every non-terminal entity to parse strings. | It finds out productions to use by replacing input string. |
| It is a type of top-down parsing built from a set of mutually recursive procedures where each procedure implements one of non-terminal of grammar. | It is a type of top-down approach, which is also a type of recursive parsing that does not uses technique of backtracking. |
| It contains several small functions one for each non- terminals in grammar. | The predictive parser uses a look ahead pointer which points to next input symbols to make it parser back tracking free, predictive parser puts some constraints on grammar. |
| It accepts all kinds of grammars. | It accepts only a class of grammar known as LL(k) grammar. |