Chapter 2: Syntax Analysis

What are we studying in this chapter?

- The Role of the Parser
- Context-free Grammars
- Writing a Grammar
- Parsing techniques
  - Top-down Parsing
  - Bottom-up Parsing

- 6 hours

2.1 Introduction

Every programming language such as C or PASCAL has rules that prescribe the syntactic structure of well-formed programs. The syntax of programming language constructs can be described by CFG or BNF (Backus Naur Form) notation. The parser determines the syntax or structure of a program. That is, it checks whether the input is syntactically correct or not. Before proceeding further, let us see what is a context free grammar, what is derivation and some other important terms that are used in coming chapters.

2.2 Context-free Grammars

Now, let us see "What is a context free grammar?"

Definition: The context free grammar in short a CFG is 4-tuple G = (V, T, P, S) where

- V is set of variables. The variables are also called non-terminals.
- T is set of terminals.
- P is set of productions. All productions in P are of the form $A \rightarrow \alpha$ where A is a non-terminal and α is string of grammar symbols.
- S is the start symbol.

Ex 1: Grammar to generate one of more a's is shown below:

$A \rightarrow a \mid aA$

Ex 2: Grammar to recognize an if-statement is shown below:

$S \rightarrow i\ C\ t\ S \mid i\ C\ t\ S\ e\ S \mid a$

$C \rightarrow b$

where *i* – stands for **if** keyword
t – stands for **then** keyword
e – stands for **else** keyword
a – stands for a statement
b – stands for a statement

Now, let us see "What are the various notations used when we write the grammars?" The various notations used in a context free grammar are shown below:

1. The following symbols are terminals
   - a) The keywords such as **if**, **for**, **while**, **do-while** etc.
   - b) Digits from 0 to 9
   - c) Symbols such as +, -, *, / etc
   - d) The lower case letters near the beginning of alphabets such as *a*, *b*, *c*, *d* etc.
   - e) The bold faced letters such as **id**

2. The following symbols are non-terminals
   - a) The lower case names such as *expression*, *operator*, *operand*, *statement* etc
   - b) The capital letters near the beginning of the alphabets such as A, B, C, D etc
   - c) The letter S is the start symbol

3. The lower case letters near the end of the alphabets such as u, v, w, x, y, z represents string of terminals.

4. The capital letters near the end of the alphabets such as U, V, W, X, Y, Z etc represent grammar symbols. A grammar symbol can be a terminal or a non-terminal.

5. The Greek letters such as α, β, γ, δ etc. represent string of grammar symbols.

2.3 Derivation

Now, let us see "What is derivation?"

Definition: The process of obtaining string of terminals and/or non-terminals from the start symbol by applying some set of productions (it may include all productions) is called *derivation.*

For example, if $A \rightarrow \alpha B\gamma$ and $B \rightarrow \beta$ are the productions, the string $\alpha\beta\gamma$ can be obtained from A- production as shown below:

$A \Rightarrow \alpha B\gamma$ [Apply the production $A \rightarrow \alpha B\gamma$]
$\quad \Rightarrow \alpha\beta\gamma$ [Replace B by β using the production $B \rightarrow \beta$]

The above derivation can also be written as shown below:

$$A \overset{+}{\Rightarrow} \alpha\beta\gamma$$

Observe the following points:

- If a string is obtained by applying only one production, then it is called one-step derivation and is denoted by the symbol ‘⇒ ‘.
- If one or more productions are applied to get the string αβγ from A, then we write

$$A \overset{+}{\Rightarrow} \alpha\beta\gamma$$

- If zero or more productions are applied to get the string αβγ from A, then we write

$$A \overset{*}{\Rightarrow} \alpha\beta\gamma$$

Example 2.1: Consider the grammar shown below from which any arithmetic expression can be obtained.

$$\begin{aligned}
E &\rightarrow E + E \\
E &\rightarrow E - E \\
E &\rightarrow E * E \\
E &\rightarrow E / E \\
E &\rightarrow \textbf{id}
\end{aligned}$$

Obtain the string **id + id * id** and show the derivation for the same.

Solution: The derivation to get the string **id + id * id** is shown below.

$$\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow \textbf{id} + E \\
&\Rightarrow \textbf{id} + E * E \\
&\Rightarrow \textbf{id} + \textbf{id} * E \\
&\Rightarrow \textbf{id} + \textbf{id} * \textbf{id}
\end{aligned}$$

Thus, the above sequence of steps can also be written as:

$$E \overset{+}{\Rightarrow} \textbf{id + id * id}$$

which indicates that the string **id + id * id** is obtained in one or more steps by applying various productions.

Now, let us see “What are the two types of derivations?” The two types of derivations are:

- Leftmost derivation
- Rightmost derivation

2.3.1 Leftmost derivation

Now, let us see “What is leftmost derivation?”

Definition: The process of obtaining a string of terminals from a sequence of replacements such that only leftmost non-terminal is replaced at each and every step is called leftmost derivation.

For example, consider the following grammar:

$$
\begin{aligned}
E &\rightarrow E + E\\
E &\rightarrow E * E\\
E &\rightarrow (E)\\
E &\rightarrow \mathbf{id}
\end{aligned}
$$

The leftmost derivation for the string **id + id * id** can be obtained as shown below:

$$
\begin{aligned}
E &\underset{lm}{\Rightarrow} E + E\\
&\Rightarrow \mathbf{id} + E\\
&\Rightarrow \mathbf{id} + E * E\\
&\Rightarrow \mathbf{id} + \mathbf{id} * E\\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$

2.3.2 Rightmost derivation

Now, let us see "What is rightmost derivation?"

Definition: The process of obtaining a string of terminals from a sequence of replacements such that only right most non-terminal is replaced at each and every step is called rightmost derivation.

For example, consider the following grammar:

$$
\begin{aligned}
E &\rightarrow E + E\\
E &\rightarrow E * E\\
E &\rightarrow (E)\\
E &\rightarrow \mathbf{id}
\end{aligned}
$$

The rightmost derivation for the string **id + id * id** can be obtained as shown below:

$$
\begin{aligned}
E &\underset{rm}{\Rightarrow} E + E\\
&\Rightarrow E + E * E\\
&\Rightarrow E + E * \mathbf{id}\\
&\Rightarrow E + \mathbf{id} * \mathbf{id}\\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$

2.4 Sentence

Now, let us see "What is a sentence?"

Definition: Let G = (V, T, P, S) be a CFG. Any string $w \in (V \cup T)^*$ which is derivable from the start symbol S such that $S \overset{*}{\Rightarrow} w$ is called a sentence or *sentential form* of G. For example, consider the derivation:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \mathbf{id} + E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

The final string of terminals i.e., **id + id * id** is called sentence of the grammar.

Now, let us see "What the different sentential forms?" The two sentential forms are:

- Left sentential form
- Right sentential form

2.4.1 Left sentential form

Now, let us see "What is left sentential form?"

Definition: If there is a derivation of the form $S \overset{*}{\Rightarrow} \alpha$, where at each step in the derivation process only a left most variable is replaced, then α is called *left-sentential* form of G.

For example, consider the following grammar and its leftmost derivation:

Grammar

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \\ E &\rightarrow \mathbf{id} \end{aligned}$$

leftmost derivation

$$\begin{aligned} E &\underset{lm}{\Rightarrow} E + E \\ &\Rightarrow \mathbf{id} + E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

In the above leftmost derivation, the string of grammar symbols obtained in each step such as:

$$\{ E + E,\ \mathbf{id} + E,\ \mathbf{id} + E * E,\ \mathbf{id} + \mathbf{id} * E,\ \mathbf{id} + \mathbf{id} * \mathbf{id} \}$$

are various *left-sentential forms* of the given grammar.

2.4.2 Right sentential form

Now, let us see "What is right sentential form?"

Definition: If there is a derivation of the form $S \overset{*}{\Rightarrow} \alpha$, where at each step in the derivation process only a right most non-terminal is replaced, then α is called *right-sentential* form of G.

For example, consider the following grammar and its rightmost derivation:

Grammar

$$\begin{array}{lll} E & \rightarrow & E + E \\ E & \rightarrow & E * E \\ E & \rightarrow & (E) \\ E & \rightarrow & \mathbf{id} \end{array}$$

rightmost derivation

$$\begin{array}{lll} E & \underset{rm}{\Rightarrow} & E + E \\ & \Rightarrow & E + E * E \\ & \Rightarrow & E + E * \mathbf{id} \\ & \Rightarrow & E + \mathbf{id} * \mathbf{id} \\ & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id} \end{array}$$

In the above rightmost derivation, the string of grammar symbols obtained in each step such as:

$$\{ E + E,\ E + E * E,\ E + E * \mathbf{id},\ E + \mathbf{id} * \mathbf{id},\ \mathbf{id} + \mathbf{id} * \mathbf{id} \}$$

are various *right-sentential forms* of the given grammar.

Example 2.2: Obtain the leftmost derivation for the string **aaabbabbba** using the following grammar.

$$\begin{array}{lll} S & \rightarrow & aB \mid bA \\ A & \rightarrow & aS \mid bAA \mid a \\ B & \rightarrow & bS \mid aBB \mid b \end{array}$$

The leftmost derivation for the string **aaabbabbba** is shown below:

$$\begin{array}{llll} S & \overset{lm}{\Rightarrow} & aB & \text{(Applying } S \rightarrow aB) \\ & \Rightarrow & aaBB & \text{(Applying } B \rightarrow aBB) \\ & \Rightarrow & aaaBBB & \text{(Applying } B \rightarrow aBB) \\ & \Rightarrow & aaabBB & \text{(Applying } B \rightarrow b) \\ & \Rightarrow & aaabbB & \text{(Applying } B \rightarrow b) \\ & \Rightarrow & aaabbaBB & \text{(Applying } B \rightarrow aBB) \\ & \Rightarrow & aaabbabB & \text{(Applying } B \rightarrow b) \\ & \Rightarrow & aaabbabbS & \text{(Applying } B \rightarrow bS) \\ & \Rightarrow & aaabbabbbA & \text{(Applying } S \rightarrow bA) \\ & \Rightarrow & aaabbabbba & \text{(Applying } A \rightarrow a) \end{array}$$

2.4.3 Language

Now, let us see "What is the language generated by grammar?" The formal definition of the language accepted by a grammar is defined as shown below.

Definition: Let G = (V, T, P, S) be a grammar. The language L(G) generated by the grammar G is

$$L(G) = \{w \mid S \overset{*}{\Rightarrow} w \text{ and } w \in T^{*}\}$$

i.e., w is a string of terminals obtained from the start symbol S by applying various productions.

For example, for the grammar A → a | aA the various strings that are generated are *a*, *aa*, *aaa*, ……and so on.

So, L = { a, aa, aaa, aaaa, …….}

2.4.4 Derivation Tree (Parse tree)

The derivation can be shown in the form of a tree. Such trees are called derivation or parse trees. The leftmost derivation as well as the right most derivation can be represented using derivation trees. Now, let us see "What is derivation tree or parse tree?" The derivation tree can be defined as shown below.

Definition: Let G = (V, T, P, S) be a CFG. The tree is derivation tree (parse tree) with the following properties.

1. The root has the label S.
2. Every vertex has a label which is in (V U T U ε).
3. Every leaf node has label from T and an interior vertex has a label from V.
4. If a vertex is labeled A and if $X_1, X_2, X_3, \ldots. X_n$ are all children of A from left, then $A \rightarrow X_1X_2X_3\ldots.X_n$ must be a production in P.

For example, consider the following grammar and its rightmost derivation along with parse tree:

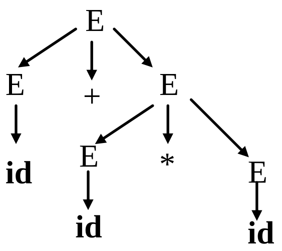
Grammar

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \\ E &\rightarrow \textbf{id} \end{aligned}$$

rightmost derivation

$$\begin{aligned} E \;\underset{rm}{\Rightarrow}\; & E + E \\ \Rightarrow\; & E + E * E \\ \Rightarrow\; & E + E * \textbf{id} \\ \Rightarrow\; & E + \textbf{id} * \textbf{id} \\ \Rightarrow\; & \textbf{id} + \textbf{id} * \textbf{id} \end{aligned}$$

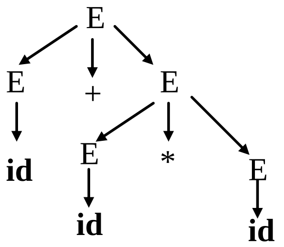
Parse tree



Now, let us see "What is the yield of the tree?" The *yield* of a tree can be formally defined as follows:

Definition: The *yield* of a tree is the string of symbols obtained by only reading the leaves of the tree from *left* to *right* without considering the ε-symbols. The yield of the tree is derived always from the root and the yield of the tree is always a terminal string.

For example, consider the derivation tree (or parse tree) shown below:



If we read only the terminal symbols in the above parse tree from left to right we get **id + id * id** and **id + id * id** is the yield of the given parse tree.

2.5 Ambiguous grammar

In this section, let us see "What is ambiguous grammar?"

Definition: Let G = (V, T, P, S) be a context free grammar. A grammar G is *ambiguous* if and only if there exists at least one string $w \in T^*$ for which two or more left derivations exist or two or more right derivations exist. That is, the ambiguous grammar has two or more meanings or interpretations.

Since, for every derivation a parse tree exist, the ambiguous grammar can also be defined as the one which has two or more different parse trees for the string *w* derived from start symbol S.

Example 2.3: Consider the following grammar from which an arithmetic expression can be obtained:

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E - E \\ E &\rightarrow E * E \\ E &\rightarrow E / E \\ E &\rightarrow (E) \mid I \\ I &\rightarrow \textbf{id} \end{aligned}$$

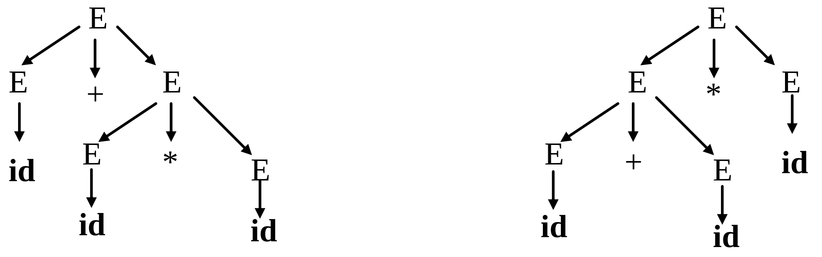
Show that the grammar is ambiguous.

Solution: The sentence **id** + **id** * **id** can be obtained from leftmost derivation in two ways as shown below.

$$\begin{array}{lll} E & \Rightarrow & E + E \\ & \Rightarrow & \textbf{id} + E \\ & \Rightarrow & \textbf{id} + E * E \\ & \Rightarrow & \textbf{id} + \textbf{id} * E \\ & \Rightarrow & \textbf{id} + \textbf{id} * \textbf{id} \end{array}$$

$$\begin{array}{lll} E & \Rightarrow & E * E \\ & \Rightarrow & E + E * E \\ & \Rightarrow & \textbf{id} + E * E \\ & \Rightarrow & \textbf{id} + \textbf{id} * E \\ & \Rightarrow & \textbf{id} + \textbf{id} * \textbf{id} \end{array}$$

The corresponding derivation trees for the two leftmost derivations are shown below:



Since the two parse trees are different for the same sentence **id** + **id** * **id** by applying leftmost derivation, the grammar is ambiguous.

Example 2.4: Is the following grammar ambiguous? (if-statement or if-then-else)

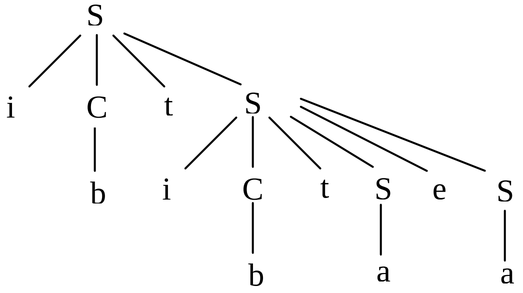
$$\begin{array}{lll} S & \rightarrow & iCtS \mid iCtSeS \mid a \\ C & \rightarrow & b \end{array}$$

The string **ibtibtaea** can be obtained by applying the leftmost derivation as shown below along with parse.

Leftmost derivation

$$\begin{array}{lll} S & \Rightarrow & iCtS \\ & \Rightarrow & ibtS \\ & \Rightarrow & ibtiCtSeS \\ & \Rightarrow & ibtibtSeS \\ & \Rightarrow & ibtibtaeS \\ & \Rightarrow & ibtibtaea \end{array}$$

Parse tree



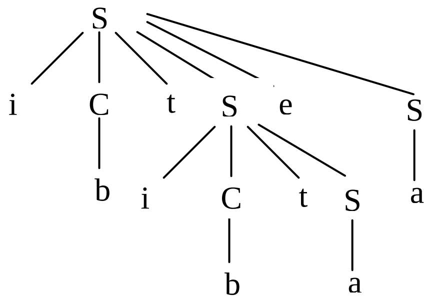
The string **ibtibtaea** can be obtained again by applying the leftmost derivation but using different sets of productions as shown below along with parse tree.

Note: i –if, t –then, e – else , b – other statement, a – other statement

Leftmost derivation

$$\begin{aligned} S &\Rightarrow iCtSeS \\ &\Rightarrow ibtSeS \\ &\Rightarrow ibtiCtSeS \\ &\Rightarrow ibtibtSeS \\ &\Rightarrow ibtibtaeS \\ &\Rightarrow ibtibtaea \end{aligned}$$

Parse tree



Since there are two different parse trees for the string ‘ibtibtaea’ by applying leftmost derivation the given grammar is ambiguous. The grammar has two interpretations or two meanings.

2.6 Eliminating ambiguity

Some grammars that are ambiguous can be converted into unambiguous grammars. This can be done using two methods:

- Dis-ambiguity rule
- Using precedence and associativity of operators

2.6.1 Dis-ambiguity rule

We have already seen in the previous problem that the grammar corresponding to if-statement is ambiguous. This is due to dangling-else. The dangling else problem can be eliminated and thus ambiguity of the grammar can also be eliminated.

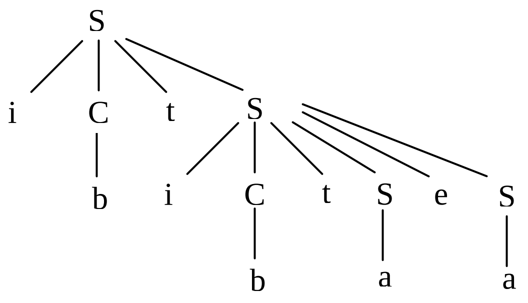
Now, let us see “What is dangling else problem?” Consider the following grammar:

$$\begin{aligned} S &\rightarrow iCtS \mid iCtSeS \mid a \\ C &\rightarrow b \end{aligned}$$

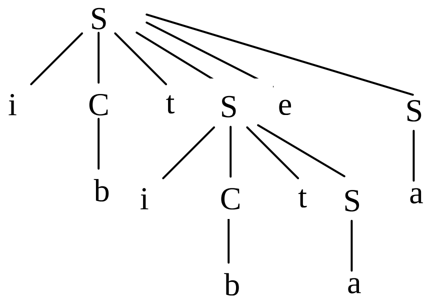
where

- i stands for keyword **if**
- C stands for **Condition** to be satisfied. Here C is a non-terminal
- t stands for keyword **then**
- S stands **statement** for non-terminal
- e stands for keyword **else**
- a stands for other statement
- b stands for other statement

Since the above grammar is ambiguous, we get two different parse trees for the string **ibtibtaea** (Look at solution for previous problem for details) as shown below:



Parse tree 1



Parse tree 2

Since there are two parse trees for the same string **ibtibtaea** the given grammar is ambiguous. Observe the following points:

- The first parse tree associates **else** with 2nd if-statement
- The second parse tree associates **else** with first if-statement.

This ambiguity whether to associate **else** with first if-statement or second if-statement is called "dangling else problem".

Now, let us see "How dangling else problem can be solved?" The dangling else problem can be solved by constructing unambiguous grammar as shown below:

Example 2.5: Eliminate ambiguity from the following ambiguous grammar:

$$S \rightarrow iCtS \mid iCtSeS \mid a$$
$$C \rightarrow b$$

Solution: In all programming languages when if-statements are nested, the first parse tree is preferred. So, the general rule is "Match each **else** with closest unmatched **then**". This rule can be directly incorporated into grammar and ambiguity can be eliminated as shown below:

Step 1: The matched statement **M** is an if-else statement where the statement S before **else** and after **else** keyword is matched. This can be expressed as:

$$M \rightarrow i\ C\ t\ M\ e\ M$$

Step 2: An unmatched statement **U** is the one consisting of:

a) simple if-statement where the statement S is matched statement or unmatched statement. The equivalent production is:

$$U \rightarrow i\ C\ t\ S$$

b) if-else statement where the statement before **else** is matched and statement after **else** is unmatched. The equivalent production is:

$$U \rightarrow i\ C\ t\ M\ e\ U$$

Step 3: The matched statement **M** and un-matched statement **U** can obtained using the statement S as shown below:

S → M | U

So, the final grammar which is un-ambiguous is shown below:

S → M | U
M → i C t M e M
U → i C t S
U → i C t M e U

Observe that the above grammar associates **else** with closest **then** and eliminates ambiguity from the grammar.

2.6.2. Eliminating ambiguity using precedence and associativity

This method is explained using the following example:

Example 2.6: Convert the following ambiguous grammar into unambiguous grammar

E → E * E | E - E
E → E ^ E | E / E
E → E + E
E → (E) | **id**

The grammar can be converted into unambiguous grammar using the precedence of operators as well as associativity operators as shown below:

Step 1: Arrange the operators in increaing order of the precedence along with associativity as shown below:

Operators	Associativity	non-terminal used
+ , –	LEFT	E
*, /	LEFT	T
^	RIGHT	P

Since there are three levels of precedence, we associate three non-terminals: E, T and P. Also an extra non-terminal F, generating basic units in an arithmetic expression.

Step 2: The basic units in expression are **id** (identifier) and parenthesized expressions. The production corresponding to this can be written as:

F → (E) | **id**

Step 3: The next highest priority operator is ^ and it is right associative. So, the production must start from the non-terminal P and it should have right recursion as shown below:

$$P \rightarrow F \;\hat{}\; P \mid F$$

Step 4: The next highest priority operators are * and / and they are left associative. So, the production must start from the non-terminal T and it should have left recursion as shown below:

$$T \rightarrow T * P \mid T / P \mid P$$

Step 5: The next highest priority operators are + and – and they are left associative. So, the production must start from the non-terminal E and it should have left recursion as shown below:

$$E \rightarrow E + T \mid E - T \mid T$$

Step 6: The final grammar which is unambiguous can be written as shown below:

$$\boxed{\begin{array}{l} E \rightarrow E + T \mid E - T \mid T \\ T \rightarrow T * P \mid T / P \mid P \\ P \rightarrow F \;\hat{}\; P \mid F \\ F \rightarrow (E) \mid \mathbf{id} \end{array}}$$

Example 2.7: Convert the following ambiguous grammar into unambiguous grammar

$$\begin{array}{l} E \rightarrow E + E \\ E \rightarrow E - E \\ E \rightarrow E \;\hat{}\; E \\ E \rightarrow E * E \\ E \rightarrow E / E \\ E \rightarrow (E) \mid \mathbf{id} \end{array}$$

by considering * and – operators lowest priority and they are left associative, / and + operators have the highest priority and are right associative and ^ operator has precedence in between and it is left associative.

The grammar can be converted into unambiguous grammar using the precedence of operators as well as associativity operators as shown below:

Step 1: Arrange the operators in increasing order of the precedence along with associativity as shown below:

Precedence	Operators	Associativity	non-terminal used
(lowest)	* , –	LEFT	E
	^	LEFT	P
(highest)	/, +	RIGHT	T

Since there are three levels of precedence we associate three non-terminals: E, P and T. Also use an extra non-terminal F generating basic units in an arithmetic expression.

Step 2: The basic units in expression are **id** (identifier) and parenthesized expressions. The production corresponding to this can be written as:

$$F \rightarrow (E) \mid \mathbf{id}$$

Step 3: The next highest priority operators are + and / and they are right associative. So, the production must start from the non-terminal T and it should be right recursive in RHS of the production as shown below:

$$T \rightarrow F + T \mid F \,/\, T \mid F$$

Step 4: The next highest priority operator is ^ and it is left associative. So, the production must start from the non-terminal P and it should be left recursive in RHS of the production as shown below:

$$P \rightarrow P \,\hat{}\, T \mid T$$

Step 5: The next highest priority operators are * and – and they are left associative. So, the production must start from the non-terminal E and it should be left recursive in RHS of the production as shown below:

$$E \rightarrow E + P \mid E - P \mid P$$

Step 6: The final grammar which is unambiguous can be written as shown below:

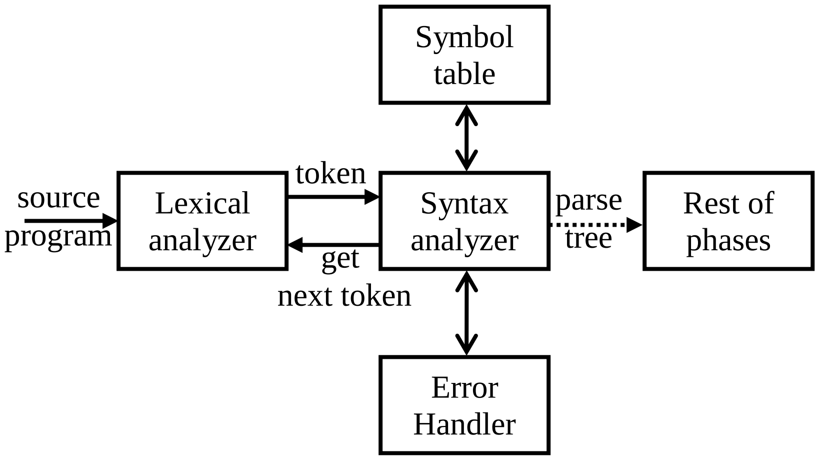
$$\boxed{\begin{array}{l} E \rightarrow E + P \mid E - P \mid P \\ P \rightarrow P \,\hat{}\, T \mid T \\ T \rightarrow F + T \mid F \,/\, T \mid F \\ F \rightarrow (E) \mid \mathbf{id} \end{array}}$$

2.7 The Role of the Parser

First, let us see "What is parsing?"

Definition: Parsing is the process of getting tokens from the lexical analyzer and obtains a derivation for the sequence of tokens and builds a parse tree. Thus, if the program is syntactically correct, the parse tree is generated. If a derivation for the sequence of tokens does not exist i.e., if the program is syntactically wrong, it results in syntax error and the parser displays the appropriate error messages. The parse trees are very important in figuring out the meaning of a program or part of the program. The parse tree is also called syntax tree. **Parser** also called syntax analyzer is the one which does parsing.

The block diagram that shows the interaction of parser with other modules and phases is shown below:



The role of the parser or the various activities that are performed by the parser are shown below:

- Parser reads sequence of tokens from the lexical analyzer
- The parser checks whether the tokens obtained from lexical analyzer can be successfully generated. This is done by obtaining a derivation for the sequence of tokens and builds the parse tree.
- If a derivation is obtained using the sequence of tokens it indicates that program is syntactically correct and the parse tree is generated.
- If a derivation is not obtained using the sequence of tokens it indicates that program is syntactically wrong and the parse tree is not generated. Now, the parser reports appropriate error messages clearly and accurately along with line numbers
- Parser also recovers from each error quickly so that subsequent errors can be detected and displayed so that the user can correct the programs.

2.7.1 Error Recovery strategies

The following activities are performed whenever errors are detected by the parser:

- Detect the syntax errors accurately and produce appropriate error messages so that the programmer can correct the program.
- It has to recover from the errors quickly and detect subsequent errors in the program.
- Error handler should take all the actions very fast and should not slowdown the compilation process.

Now, let us see "What are error recovery strategies of the parser (or syntax analyzer)?" The various error recovery techniques are:

- Panic mode recovery
- Error productions
- Phrase level recovery
- Global correction

Panic Mode Recovery: It is the simplest and most popular error recovery method. When an error is detected, the parser discards symbols one at a time until next valid token (called synchronizing token) is found. The typical synchronizing tokens are:

- statement terminators such as semicolon
- Expression terminators such as \n

It often skips a considerable amount of input without checking it for additional errors. Once the synchronizing token found, the parser will continue from that point onwards to identify the subsequent errors. In situations where multiple errors are in the same statement, this method is not useful.

For example, consider the erroneous expression:

(5 ** 2) +8

- The parser scans the input from left to right and finds no mistake after reading (, 5 and *.
- After reading the second *, it knows that no expression has consecutive * operators and it displays an error "Extra * in the input"
- Now, it has to recover from the error. In panic mode recovery, it skips all input symbols till the next integer 2 is encountered. Here, 2 is the synchronizing token.
- Thus, error is detected and recovered from the error in panic mode recovery

Error Productions: In this type of error recovery strategy, we introduce error productions. The error productions specify commonly known mistakes in the grammar. When we implement the parser, when an error production is used, it displays the appropriate error message. For example, consider the expression 10x. Mathematically it means multiply 10 with x. But, in a programming language we should write 10*x. Such errors can be identified very easily by incorporating error productions and within the body of the function, display appropriate error messages.

Disadvantages

- Can resolve many errors, but not all potential errors.
- The introduction of error productions will complicate the grammar

Phrase Level Recovery: It is an error correcting method. On discovering an error, a parser may perform local correction on the remaining input. This is normally done by inserting, deleting or/and replacing the input and enable the parser to continue parsing.

Ex: Replacing a comma by a semicolon, deleting an extraneous semicolon, or inserting a missing semicolon.

Disadvantages

- Very difficult to implement
- Slows down the parsing of correct programs
- Proper care must be taken while choosing replacements as they may lead to infinite loops.

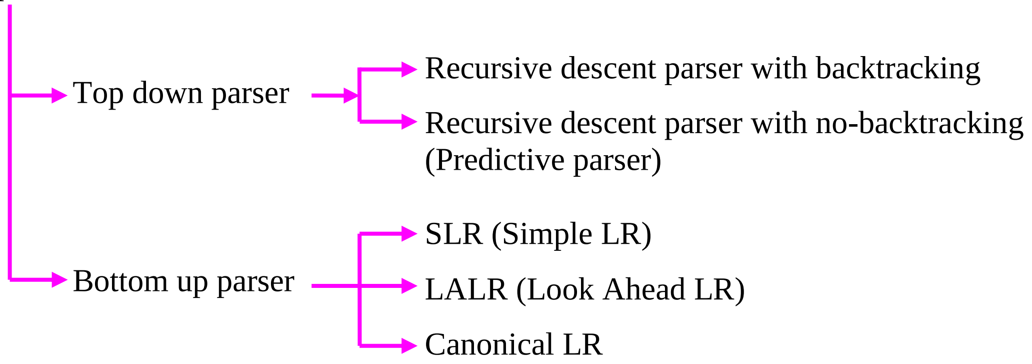
Global Correction: This is also one of the error correction strategies. The various points to remember in this error correction method are:

- These methods replace incorrect input with correct input using least-cost-correction algorithms.
- These algorithms take an incorrect input string x and grammar G, and 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.
- These methods are costly to implement in terms of time and space and hence are only of theoretical interest.

2.7.2 Parsing techniques

In this section, let us see "What are the different types of parsers?"

The parsers are classified as shown below:



- Top down parser
  - Recursive descent parser with backtracking
  - Recursive descent parser with no-backtracking (Predictive parser)
- Bottom up parser
  - SLR (Simple LR)
  - LALR (Look Ahead LR)
  - Canonical LR

2.8 Top-down Parsing

Now, let us see "What is top down parser?"

Definition: The process of constructing a parse tree for the string of tokens (obtained from the lexical analyzer) from top i.e., starting from the root node and creating the nodes of the parse tree in preorder in depth-first-search manner is called top down parsing technique. Thus, top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string and constructing the parse tree for that derivation. The parser that uses this approach is called top down parser. Since the parsing starts from top (i.e., root) down to the leaves, it is called top down parser.

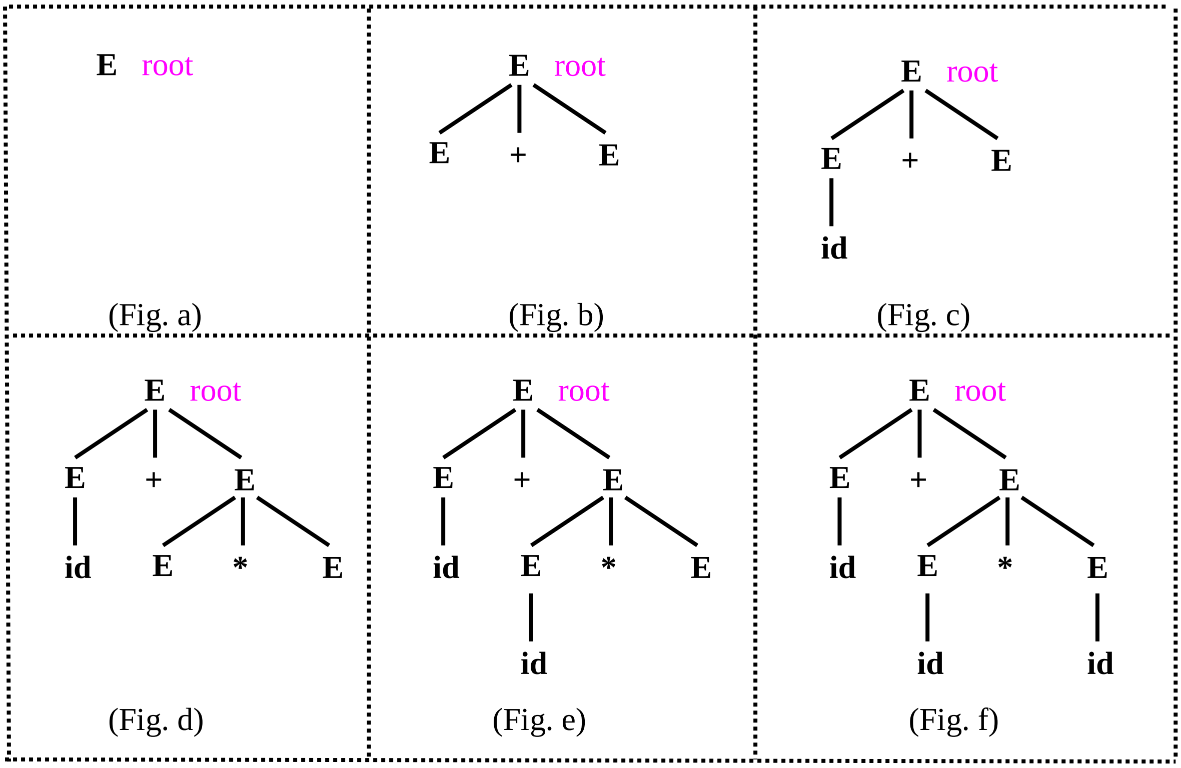
Example 2.8: Show the top-down parsing process for the string id + id * id for the grammar

$$
\begin{aligned}
E &\rightarrow E + E\\
E &\rightarrow E * E\\
E &\rightarrow (E)\\
E &\rightarrow id
\end{aligned}
$$

Solution: The string **id + id * id** can be obtained by the grammar by applying leftmost derivation as shown below:

$$
\begin{aligned}
E &\underset{lm}{\Rightarrow} E + E && \text{(fig a) step 1}\\
&\Rightarrow \textbf{id} + E && \text{(fig b) step 2}\\
&\Rightarrow \textbf{id} + E * E && \text{(fig c) step 3}\\
&\Rightarrow \textbf{id} + \textbf{id} * E && \text{(fig d) step 4}\\
&\Rightarrow \textbf{id} + \textbf{id} * \textbf{id} && \text{(fig e) step 5}
\end{aligned}
$$

The above derivation can be written in the form of a parse tree from the start symbol using top-down approach as shown below:



(Fig. a) (Fig. b) (Fig. c) (Fig. d) (Fig. e) (Fig. f)

Initial: Start from root node E

Step 1: Replace E with E + E using E → E + E. It is shown in figure (b)

Step 2: Replace E with **id** using E → id. It is shown in figure (c).

Step 3: Replace E with E * E using E → E * E. It is shown in figure (d).

Step 4: Replace E with **id** using E → id. It is shown in figure (e).

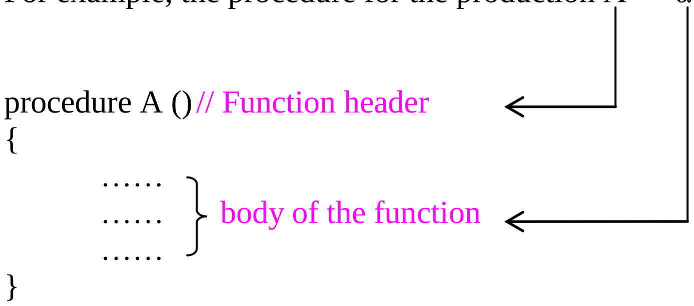
Step 5: Replace E with **id** using E → id. It is shown in figure (f).

2.8.1 Recursive descent parser

Now, let us see “What is recursive descent parser?”

Definition: A recursive descent parser is a top down parser in which parse tree is constructed from the top starting from root node and selecting the productions from left to right (if two or more alternative productions exists). For every non-terminal there exists a recursive procedure and the right hand of the production of that non-terminal is implemented as the body of the procedure. The sequence of terminals and non-terminals on the right hand side of the production correspond to matching with input symbols and calls to other procedures while selecting the alternate production is implemented using switch or if-statements. Thus, the syntax or structure of the resulting program closely mirrors that of the grammar it recognizes.

For example, the procedure for the production $A \rightarrow \alpha$ can be written as shown below:



```
procedure A ()// Function header
{
        ......
        ......  } body of the function
        ......
}
```

Observe the following points:

- For the variable A on the left hand side of the production, we write the function header
- For the string of grammar symbols denoted by α on the right hand side of the production corresponds to the function body.
- Thus, for every non-terminal in the grammar we write the procedure or function as in previous two steps.

Working: The recursive descent parser works as shown below:

- Execution starts from the function that corresponds to the start symbol of the grammar
- If the body of the function scans the entire input string, then parsing is successful. Otherwise, the input string is not proper and parsing halts.
- Each non-terminal is associated with a parsing procedure or a function that can recognize any sequence of tokens generated by that non-terminal
- Within the function, both non-terminals and terminals are matched
- To match the non-terminal A, we call the function/procedure A which corresponds to non-terminal A. These calls may be recursive and hence the name recursive descent parser.

- To match the terminal *a*, we compare current input symbol with *a*. If there is match, it is syntactically correct and we increment the input pointer and get the next token
- If the current input symbol is not *a*, it is syntactically wrong and appropriate error message is displayed
- Some error correction may be done to recover from each error quickly so that subsequent errors can be detected and displayed so that the user can correct the programs.

The general procedure for a recursive-descent parsing that uses top-down parser is shown below:

Example 2.9: Algorithm for recursive descent parser (backtracking is not supported)

```
procedure A()          // $A \rightarrow X_1X_2X_3 \ldots\ldots X_k$
{
      for i = 1 to k do
            if ($X_i$ is a non-terminal)
                  call procedure $X_i$();
            else if ($X_i$ is same as current input symbol $a$)
                  advance the input to the next symbol
            else
                  error();
      end for
}
```

Now, let us write the recursive parsers for some of the grammars.

Example 2.10: Write the recursive descent parser for the following grammar

$E \rightarrow T$
$T \rightarrow F$
$F \rightarrow (E) \mid \textbf{id}$

```
// function corresponding to the production $E \rightarrow T$
procedure E()
{
     T();
}
```

```
// function corresponding to the production T → F
procedure T()
{
        F();
}
```

```
// function corresponding to the production F → (E)  | id
//                                                  F → (        E        )         |           id
procedure F()
{
         if (input_symbol == '(' )

                  advance input pointer
                  E();
                  if (input_symbol == ')')
                           advance input pointer
                  else
                           error()
                  end if

         else if (input_symbol == id)
                  advance input pointer
         else
                  error();
         end if
}
```

Now, let us see "What are the different types of recursive descent parsers?" The recursive descent parser can be classified into two types:

- Recursive descent parser with backtracking
- Recursive descent parser without backtracking (predictive parser)

2.8.2 Recursive descent parser with backtracking

Now, let us see "What is the need for backtracking in recursive descent parser?" The backtracking is necessary for the following reasons:

- During parsing, the productions are applied one by one. But, if two or more alternative productions are there, they are applied in order from left to right one at a time.

- When a particular production applied fails to expand the non-terminal properly, we have to apply the alternate production. Before trying alternate production, it is necessary undo the activities done using the current production. This is possibly only using backtracking.

But, the recursive descent parsers with backtracking are not frequently used. So, we just concentrate on how they work with example.

Example 2.11: Show the steps involved in recursive descent parser with backtracking for the input string *cad* for the following grammar

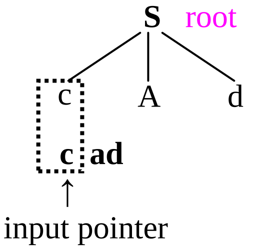
S → cAd
A → ab | a

Solution: The three parts that are used while parsing the string are:

Given grammar	**String to be parsed**	**Parse tree**
S → cAd A → ab | a	cad ↑ input pointer	**S** (root)

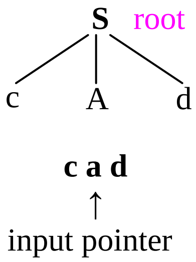
Observe that input-pointer points to the next character to be read.

Step 1: The only unexplored node is S and we apply the production S → cAd to expand the non-terminal S as shown below:

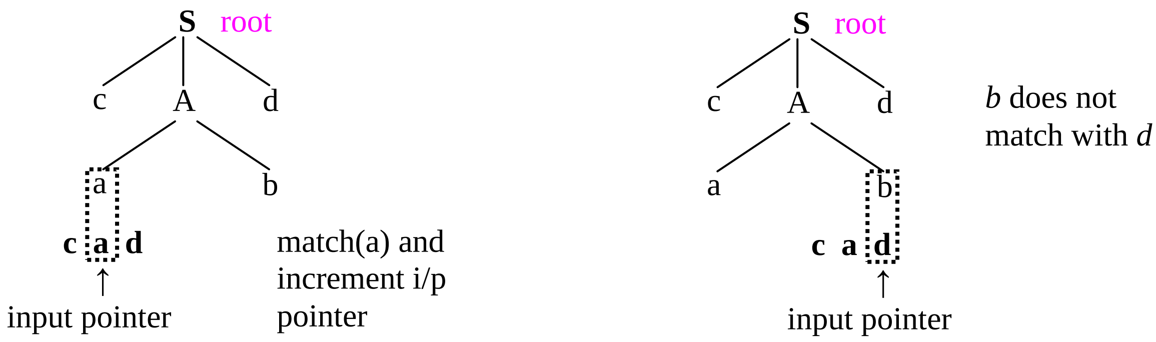


match(c) and increment i/p pointer

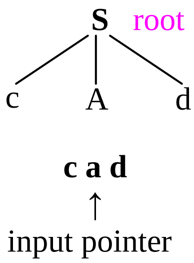
Step 2: Now, the next node to be expanded is A and input pointer points to *a* as shown below:



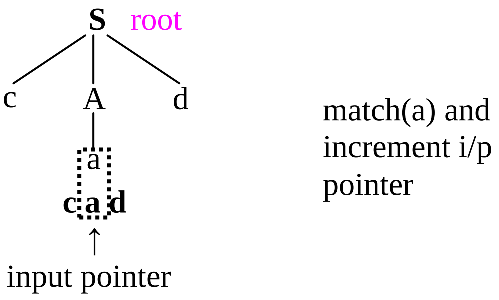
Step 3: Since two productions are there from A, the first production A → ab is selected and non-terminal A is expanded as shown below:



Step 4: Observe that by selecting A → ab the input string is not matched. So, we have to reset the pointer to input symbol *a* (so as to get the parse tree shown in step 2). This is done using backtracking and is shown in (fig a). After backtracking try expanding A using the second production A → a and then proceed comparing as shown in (fig b).



(Fig a.)



Step 5: Now, the next symbol *d* in grammar is compared with *d* in the input and they match. Finally, we halt and announce successful completion of parsing.

Now, let us see "For what type of grammars recursive descent parser cannot be constructed? What is the solution?"

The recursive descent parser cannot be constructed for a grammars having:

- Ambiguity. The solution is to eliminate ambiguity from the grammar.
- Left recursion. The solution is to eliminate left recursion from the grammar.
- Two or more alternatives having a common prefix. The solution is to left factor the grammar.

2.8.3 Left recursion

Now, let us see "What is left recursion? What problems are encountered if a recursive descent parser is constructed for a grammar having left recursion?"

Definition: A grammar G is said to be left recursive if it has non-terminal A such that there is a derivation of the form:

$$A \overset{+}{\Rightarrow} A\alpha \qquad \text{(Obtained by applying one or more productions)}$$

where α is string of terminals and non-terminals. That is, whenever the first symbol in a partial derivation is same as the symbol from which this partial derivation is obtained, then the grammar is said to be left-recursive grammar. A grammar may have:

- immediate left recursion
- indirect left recursion

Immediate left recursion: A grammar G is said to have immediate left recursion if it has a production of the form:

$$A \rightarrow A\alpha$$

For example, consider the following grammar:

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

In the above grammar consider the first two productions:

$$E \rightarrow E + T$$
$$T \rightarrow T * F$$

Observe that in the above two productions, the first symbol on the right hand side of the production is same as the symbol on the left hand side of the production. So, the given grammar has immediate left recursion in two productions.

Indirect left recursion: A left recursion involving derivations of *two or more steps* so that the first symbol on the right hand side of the partial derivation is same as the symbol from which the derivation started is called indirect left recursion. For example, consider the following grammar:

$$E \rightarrow T$$
$$T \rightarrow F$$
$$F \rightarrow E + T \mid id$$

Consider the following derivation:

$$E \Rightarrow T \Rightarrow F \Rightarrow E + T$$

In the above derivation, note that in the partial derivation to get the string E + T, the first symbol is E which is same as the symbol from which the derivation started. But, the string E+T is obtained from E by applying two or more productions. So, the given grammar even though it is not having immediate left recursion, it has indirect left recursion and hence the grammar is left recursive.

Now, let us write the recursive descent parser for the grammar having left recursion.

Example 2.12: Consider the production E→ E + T . Write the recursive descent parser.

Solution: The recursive descent parser for the production E→ E + T can be written as shown below:

```
// function corresponding to the production E → E  +  T
procedure E()
{
        E();

        if (input_symbol = '+')
                advance input pointer
        else
                error
        end if

        T();
}
```

Now, let us see "What is the problem in constructing recursive descent parser for the grammar having left recursion?" Observe the following points (with respect to the above procedure which has left recursion)

- When a procedure is invoked, the parameter values along with return address will be pushed on to the stack and hence stack size decreases
- The procedure E() is called recursively infinitely without consuming any input and hence the size of the stack grows very fast and stack will be full soon.
- Since there is no space left on the stack to push parameter values and return address, the system crashes.
- So. the recursive descent parser that is built using left-recursive grammar can cause a parser to go into an infinite loop eventually crashing the system and hence the left recursive grammar is not suitable for recursive descent parser. Hence, we have to eliminate left recursion from the grammar and then parse the string.

2.8.4 Procedure to eliminate left recursion

Consider the production of the form:

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

where β do not start with A. Note that the above grammar has left recursion. Now, let us see how to eliminate left recursion. The various strings that can be generated by above grammar are shown below:

Derivation: 1st	2nd	3rd	4th and so on.
$A \Rightarrow \beta$	$A \Rightarrow A\alpha$	$A \Rightarrow A\alpha$	$A \Rightarrow A\alpha$
	$\Rightarrow \beta\,\alpha$	$\Rightarrow A\alpha\alpha$	$\Rightarrow A\alpha\alpha$
		$\Rightarrow \beta\,\alpha\alpha$	$\Rightarrow A\alpha\alpha\alpha$
			$\Rightarrow \beta\,\alpha\alpha\alpha$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
{ β,	β α,	β αα,	β ααα........}

Observe from above derivations that the language L consists of β followed zero or more α's. The same language can be represented and generated using different grammar as shown below:

$$L = \{\, \beta\, \alpha^i \mid i \ge 0 \}$$

$$\downarrow \;\; \downarrow$$

$$A \rightarrow \beta\, A'$$ where A' should generate zero or more α's

From A' we can get zero or more α's using the following productions:

$$A' \rightarrow \epsilon \mid \alpha A'$$

So the final grammar that generate β followed by zero or more α's which do not have left recursion is shown below:

$$A \rightarrow \beta\, A'$$
$$A' \rightarrow \epsilon \mid \alpha A'$$

Thus, the grammar which has left recursion can be written in the form of another grammar that does not have left recursion as shown below:

Left recursive grammar	**Right recursive grammar**
$A \rightarrow A\alpha \mid \beta$	$A \rightarrow \beta\, A'$
	$A' \rightarrow \epsilon \mid \alpha A'$

In general,

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots\ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots\ldots \mid \beta_m$$

$$\Downarrow$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots\ldots\ldots \mid \beta_m\, A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots\ldots \mid \alpha_n A' \mid \epsilon$$

Example 2.13: Eliminate left recursion from the following grammar

$$\begin{array}{l} E \rightarrow\ E + T \mid T \\ T \rightarrow\ T * F \mid F \\ F \rightarrow\ (E) \mid id \end{array}$$

Solution: The given grammar is shown below:

$$\begin{array}{l} E \rightarrow\ E + T \mid T \\ T \rightarrow\ T * F \mid F \\ F \rightarrow\ (E) \mid id \end{array}$$

Since the first symbol on the right hand side of E-production and T-production is same as the symbol on the left hand side of the production, the grammar has immediate left recursion. Now, the immediate left recursion can be eliminated from the grammar as shown below:

Left recursive productions	**Right recursive productions**
$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots\ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots.. \mid \beta_m$	$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots\ldots.. \mid \beta_m A'$ $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots\ldots \mid \alpha_n A' \mid \epsilon$
1) $\begin{array}{cccccc} E & \rightarrow & E & \underbrace{+T} & \mid & T \\ \downarrow & & \downarrow & \downarrow & & \downarrow \\ A & \rightarrow & A & \alpha_1 & \mid & \beta_1 \end{array}$	$E \rightarrow TE'$ $E' \rightarrow +TE' \mid \epsilon$
2) $\begin{array}{cccccc} T & \rightarrow & T & \underbrace{*F} & \mid & F \\ \downarrow & & \downarrow & \downarrow & & \downarrow \\ A & \rightarrow & A & \alpha_1 & \mid & \beta_1 \end{array}$	$T \rightarrow FT'$ $T' \rightarrow *FT' \mid \epsilon$
3) $F \rightarrow\ (E) \mid id$	$F \rightarrow\ (E) \mid id$

The final grammar obtained after eliminating left recursion can be written as shown below:

$$\boxed{\begin{array}{l} E \rightarrow TE' \\ E^1 \rightarrow +TE' \mid \epsilon \\ T \rightarrow FT' \\ T^1 \rightarrow *FT' \mid \epsilon \\ F \rightarrow\ (E) \mid id \end{array}}$$

Now, we can write the recursive descent parser for the above grammar which is obtained after eliminating left recursion.

Example 2.14: Write the recursive descent parser for the following grammar:

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

The recursive descent parser for the above grammar is shown below. Note that for each non-terminal there is a procedure and the right hand side of the production is implemented as the body of the procedure as shown below:

```
// function corresponding to the production:  E → T  E'
procedure E()
{
        T();
        EDASH();
}
```

```
// function corresponding to the production:  E' → + T  E' | ∈
procedure EDASH()
{
        if ( inputsymbol == '+')
        {
                Advance input pointer
                T();
                EDASH();
        }
}
```

```
// function corresponding to the production:  T → F  T'
procedure T()
{
        F();
        TDASH();
}
```

```
// function corresponding to the production:  T' → * F T'| ϵ
procedure TDASH()
{
        if ( inputsymbol == '*')
        {
                advance input pointer
                F();
                TDASH();
        }
}
// function corresponding to the production:  F → ( E ) | id
procedure F()
{
        if ( inputsymbol == '(')
        {
                advance input pointer
                E();
                if ( inputsymbol == ')')
                        advance input pointer
                else error();
        }
        else
        {
                if (inputsymbol == id )
                        advance input pointer
                else    error();
        }
}
```

Example 2.15: Obtain top-down parse for the string **id+id*id** for the following grammar

$E \rightarrow TE'$

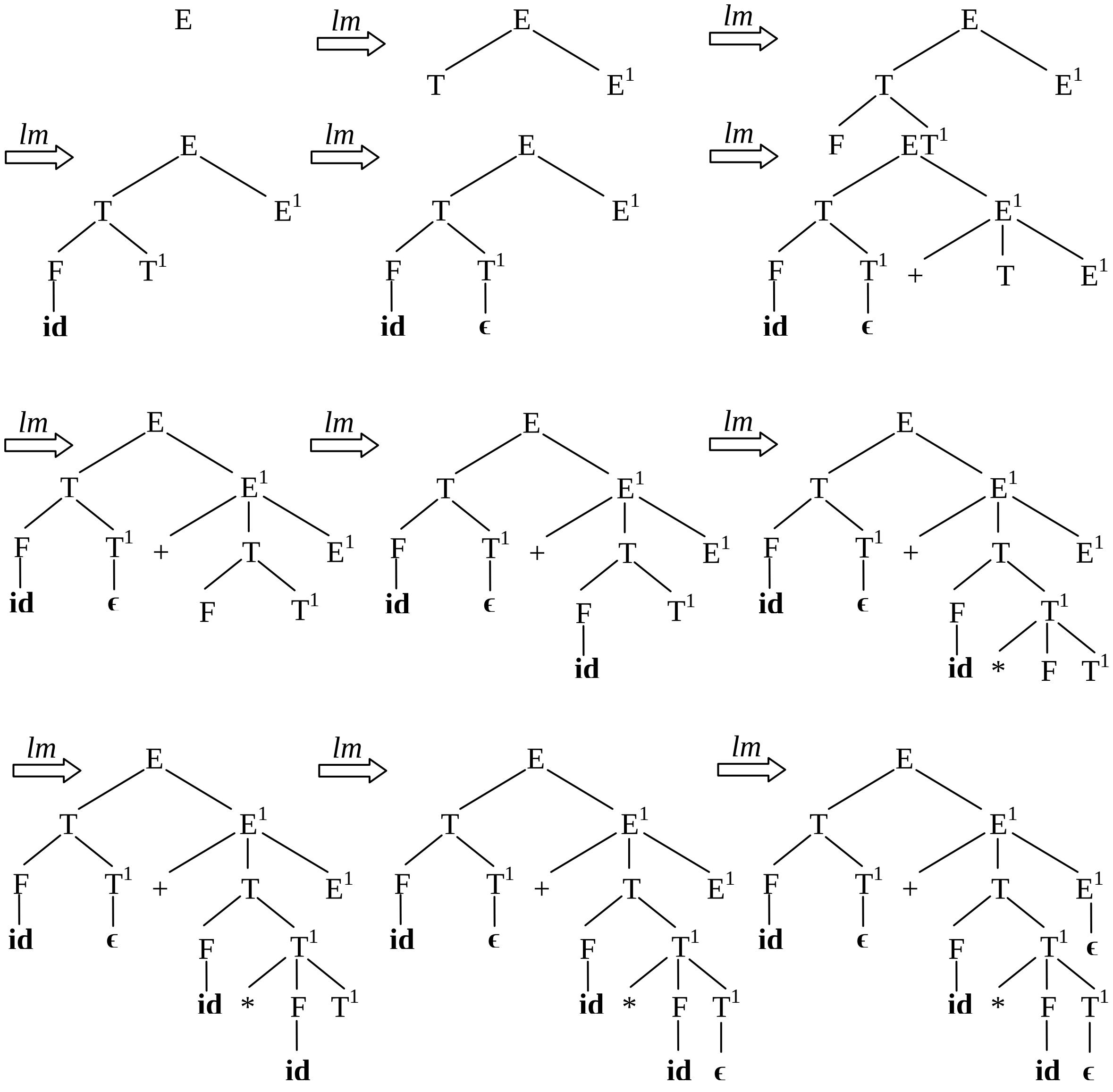
$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

The top-down parse for the string **id+id*id** for the above grammar can be written as shown below:



Example 2.16: Eliminate left recursion from the following grammar:

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \epsilon$

Solution: Left recursion can be eliminated as shown below:

Step 1: The S-production does not have immediate left recursion. So, let us not consider the production $S \rightarrow Aa \mid b$

Step 2: Consider the production: $A \rightarrow Ac \mid Sd \mid \epsilon$. Replacing the non-terminal S by the production $S \rightarrow Aa \mid b$ we get following A-production:

$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

Step 3: Now, the grammar obtained after eliminating indirect left recursion is shown below:

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

Now, immediate left recursion can be eliminated as shown below:

Left recursive productions	**Right recursive productions**
$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots\ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots\ldots \mid \beta_m$	$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots\ldots\ldots \mid \beta_m A'$ $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots\ldots \mid \alpha_n A' \mid \epsilon$
1) $S \rightarrow Aa \mid b$	$S \rightarrow Aa \mid b$
2) $\underset{\underset{A}{\downarrow}}{A} \rightarrow \underset{\underset{A}{\downarrow}}{A}\, \underbrace{c}_{\downarrow\, \alpha_1} \mid A\, \underbrace{ad}_{\downarrow\, \alpha_2} \mid \underbrace{bd}_{\downarrow\, \beta_1} \mid \underbrace{\epsilon}_{\beta_2}$	$A \rightarrow bd A' \mid \epsilon A'$ $A' \rightarrow cA' \mid adA' \mid \epsilon$

So, the final grammar obtained after eliminating left recursion is shown below:

$S \rightarrow Aa \mid b$		$S \rightarrow Aa \mid b$
$A \rightarrow bd A' \mid \epsilon A'$	can be written as	$A \rightarrow bd A' \mid A'$
$A' \rightarrow cA' \mid adA' \mid \epsilon$		$A' \rightarrow cA' \mid adA' \mid \epsilon$

Now, let us "Write the algorithm to eliminate left recursion" The algorithm to eliminate left recursion is shown below:

Example 2.17: Algorithm to eliminate left recursion (including indirect left recursion)

Input : Grammar G without ϵ-productions and with no cycles

Output: Grammar without left recursion. It may have ϵ-productions

Arrange the non-terminals in the order $A_1, A_2, A_3, \ldots\ldots\ldots A_n$

```
for i = 1 to n do
        for j = 1 to i-1 do
                Let $A_j \rightarrow \beta_1 | \beta_2 | \beta_3 |….. \beta_k$
                Replace $A_i \rightarrow A_j\alpha$ by $A_i \rightarrow \beta_1\alpha | \beta_2\alpha | \beta_3\alpha |….. \beta_k\alpha$
        end for
        Eliminate immediate left recursion among $A_i$ productions
end for
```

2.8.5 Left factoring

Now, let us see "What is left factoring? What is the need for left factoring?"

Definition: A grammar in which two or more productions from a non-terminal A do not have a common prefix of symbols on the right hand side of the A-productions is called left factored grammar. The left-factored grammar is suitable for top-down parser such as recursive descent parser with or without backtracking.

Ex 1: The grammar to generate string consisting of at least one 'a' followed by at least one 'b' can be written as shown below:

$S \rightarrow aAbB$
$A \rightarrow aA \mid \epsilon$ **Left factored grammar**
$B \rightarrow bB \mid \epsilon$

Observe the following points:

- The S-production has only one production and it cannot have common prefix on the right side of the production.
- The two A-productions do not have any common prefix on the right side.
- Finally two B-productions do not have any common prefix on the right side of that production

Ex 2: The grammar that generates string consisting of at least one 'a' followed by at least one 'b' can also be written as shown below:

$S \rightarrow AB$
$A \rightarrow aA \mid a$ **Non Left factored grammar**
$B \rightarrow bB \mid b$

Observe the following points:

- The S-production has only one production and it is not having common prefix on the right side of the production.
- The two A-productions have a common prefix "a" on the right side of the production.

- The two B-productions have a common prefix “b” on the right side of the production.
- Since common prefix is present in both A-productions and B-productions, it is not left-factored grammar.

Note: If two or more productions starting from same non-terminal have a common prefix, the grammar is not left-factored.

Now, let us see “What is the use of left factoring?” Left factoring is must for top down parser such as recursive descent parser with backtracking or predictive parser which is also recursive descent parser without backtracking. This is because, if A-production has two or more alternate productions and they have a common prefix, then the parser has some confusion in selecting the appropriate production for expanding the non-terminal A

Ex 1: Consider the following grammar that recognizes the if-statement:

S → **if** E **then** S **else** S | **if E then** S

Observe the following points:

- Both productions starts with keyword **if.**
- So, when we get the input ‘**if**’ from the lexical analyzer, we cannot tell whether to use the first production or to use the second production to expand the non-terminal S.
- So, we have to transform the grammar so that they do not have any common prefix. That is, left factoring is must for parsing using top-down parser.

Now, the question is “How to do left factoring?” The left-factoring can be done as shown below:

1) Consider two A-productions with common prefix α:

   $A \rightarrow \alpha\beta_1 |\ \alpha\beta_2$

2) Let the input begins with string derived from α. Since α is the common prefix, we retain α and we replace either β_1 or β_2 by the non-terminal A'. So, we can write the above production as:

   $A \rightarrow \alpha\ A'$

   where A' can produce either β_1 or β_2 using the production:

   $A' \rightarrow \beta_1 |\ \beta_2$

Now, after seeing the input derived from α, we can expand A' either to β_1 or to β_2. So, the given grammar is converted into left-factored grammar as shown below:

$$\boxed{A \rightarrow \alpha\beta_1 |\ \alpha\beta_2}$$

Non left-factored grammar

$$\boxed{\begin{array}{l} A \rightarrow \alpha\, A' \\ A' \rightarrow \beta_1 |\ \beta_2 \end{array}}$$

Left-factored grammar

Now, let us "Write the algorithm for doing left-factoring" The algorithm for doing left-factoring is shown below:

Example 2.18: The algorithm for left-factoring

Algorithm LEFT_FACTOR(G)

Input: Grammar G

Output: An equivalent left-factored grammar

Method: The following procedure is used:

1) For each non-terminal A, find the longest prefix α which is common to two or more of its alternatives.

2) If there is a production of the form:

   $$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots.. \alpha\beta_n \mid \gamma$$

   where γ do not start with α, then the above A-production can be written as shown below:

   $$A \rightarrow \alpha\, A' \mid \gamma$$
   $$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots.. \beta_n$$

   Here, A' is a new non-terminal.

3) Repeatedly apply the transformation in step 2 as long as two alternatives for a non-terminal have a common prefix

4) Return the final grammar which is left-factored

Example 2.19: Do the left-factoring for the following grammar:

$$\begin{aligned} S &\rightarrow iCtS \mid iCtSeS \mid a \\ C &\rightarrow b \end{aligned}$$

Solution: The given grammar is shown below:

$$\begin{aligned} S &\rightarrow iCtS \mid iCtSeS \mid a \\ C &\rightarrow b \end{aligned}$$

Since S-production has common prefix **iCtS** in more than one production, left factoring is necessary. Left factoring the above grammar can be done using the algorithm shown below:

Given productions	Left-factored productions
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots.. \alpha\beta_n \mid \gamma$	$A \rightarrow \alpha A' \mid \gamma$ $A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots.. \beta_n$
1) $\underbrace{S}_{A} \rightarrow \underbrace{iCtS}_{\alpha}\ \underbrace{\epsilon}_{\beta_1} \mid \underbrace{iCtS}_{\alpha}\ \underbrace{eS}_{\beta_2} \mid \underbrace{a}_{\gamma}$	$S \rightarrow iCtSS' \mid a$ $S' \rightarrow \epsilon \mid eS$
2) $C \rightarrow b$	$C \rightarrow b$

So, the final grammar which is obtained after doing left-factoring is shown below:

$$\begin{aligned} S &\rightarrow iCtSS' \mid a \\ S' &\rightarrow \epsilon \mid eS \\ C &\rightarrow b \end{aligned}$$

2.8.6 Problems with top down parser

Now, let us "Briefly explain the problems associated with top-down parser?" **(JUY-AUG-2009)** The various problems associated with top down parser are:

- Ambiguity in the grammar
- Left recursion
- Non-left factored grammar
- Backtracking

- **Ambiguity in the grammar:** A grammar having two or more left most derivations or two or more right most derivations is called ambiguous grammar. For example, the following grammar is ambiguous:

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

  The ambiguous grammar is not suitable for top-down parser. So, ambiguity has to be eliminated from the grammar. (For details refer section 2.6)

- **Left-recursion:** A grammar G is said to be left recursive if it has non-terminal A such that there is a derivation of the form:

  $A \overset{+}{\Rightarrow} A\alpha$ (Obtained by applying one or more productions)

  where α is string of terminals and non-terminals. That is, whenever the first symbol in a partial derivation is same as the symbol from which this partial derivation is obtained, then the grammar is said to be left-recursive grammar. For example, consider the following grammar:

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

  The above grammar is unambiguous but, it is having left recursion and hence, it is not suitable for top down parser. So, left recursion has to be eliminated (For details refer section 2.8.3 and 2.8.4)

- **Non-left factored grammar:** If A-production has two or more alternate productions and they have a common prefix, then the parser has some confusion in selecting the appropriate production for expanding the non-terminal A. For example, consider the following grammar that recognizes the if-statement:

  S → **if** E **then** S **else** S | **if E then** S

  Observe the following points:
  - Both productions starts with keyword **if.**
  - So, when we get the input ‘**if**’ from the lexical analyzer, we cannot tell whether to use the first production or to use the second production to expand the non-terminal S.
  - So, we have to transform the grammar so that they do not have any common prefix. That is, left factoring is must for parsing using top-down parser.

A grammar in which two or more productions from every non-terminal A do not have a common prefix of symbols on the right hand side of the A-productions is called left factored grammar. *(Refer previous section for doing left-factoring)*

- **Backtracking:** The backtracking is necessary for top down parser for following reasons:

  1) During parsing, the productions are applied one by one. But, if two or more alternative productions are there, they are applied in order from left to right one at a time.

  2) When a particular production applied fails to expand the non-terminal properly, we have to apply the alternate production. Before trying alternate production, it is necessary undo the activities done using the current production. This is possibly only using backtracking.

  Even though backtracking parsers are more powerful than predictive parsers, they are also much slower, requiring exponential time in general and therefore, backtracking parsers are not suitable for practical compilers.

2.8.7 Recursive descent parser with no-backtracking (Predictive parser)

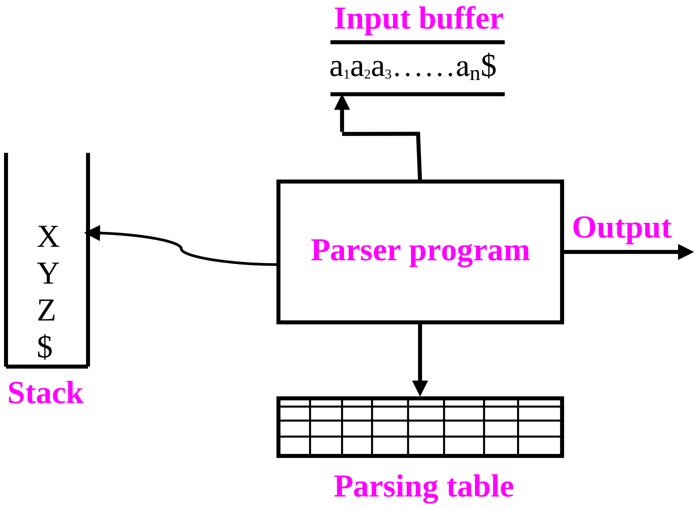
Now, let us see "What is a predictive parser? Explain the working of predictive parser. "

Definition: Predictive parser is a top down parser. It is an efficient way of implementing a recursive descent parser by maintaining a stack explicitly rather than implicitly via recursive calls. The predictive parser can correctly guess or predict which production to use if two or more alternative productions are there. This is done using two ways:

- By looking at the next few tokens (often called lookahead) it selects the correct production out of two or more alternatives productions and expand the non-terminal
- Without backtracking. So, there is no question of undoing bad choices using backtracking. In fact, bad choices will never occur.

Since, it can predict which production to use while parsing, it is called predictive parser. The predictive parsers accepts a restricted grammar called LL(k) grammars (defined in section 2.10)

Now, let us see "What are the various components of predictive parser? How it works?" The working of predictive parser can be explained easily by knowing the various components of the predictive parser. The block diagram showing the various parts of predictive parser are shown below:



The predictive parser has four components namely:

- Input
- Stack
- Parsing table
- Parsing program
- Output

- **Input :** The input buffer contains the string to be parsed and the input string ends with '\$'. Here, \$ indicates the end of the input.

- **Stack :** It contains sequence of grammar symbols and '\$' is placed initially on top of the stack. When \$ is on top of the stack, it indicates that stack is empty.

- **Parsing table :** It is a two dimensional array M[A, a] where A is a non-terminal and *a* is terminal or \$. The non-terminal A represent the row index and terminal *a* represent the column index. The entry in M[A, a] contains either a production or blank entry.

- **Parser :** It is a program which takes different actions based on X which is the symbol on top of the stack and the current input symbol *a*.

- **Output:** As output, the productions that are used are displayed using which the parse tree can be constructed.

Working of the parser: The various actions performed by the parser are shown below:

1) If X = a = \$, that is, if the symbol on top of the stack and the current input symbol is \$, then parsing is successful.

2) If X = a ≠ \$, that is, if the symbol on top of the stack is same as the current input symbol but not equal to \$, then pop X from the stack and advance the input pointer to point to next symbol.

3) If X is a terminal and $\neq$ a, that is, the symbol on top of the stack is not equal to the current input symbol, then error()

4) If X is a non-terminal and *a* is the input symbol, the parser consults the parsing table M[X, a] which contains either an X production or an error entry. If X → UVW is the corresponding production, the parser pops X from the stack and pushes U, V and W in reverse order onto the stack.

Now, before seeing how the parser parses the string, let us "Explain parsing table and how to use the parsing table? or "What information is given in the predictive parsing table?" The parsing table details and how it can be used can be explained using the example.

Example 2.20: Consider the following grammar and the corresponding predictive parsing table:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid id \end{aligned}$$

GRAMMAR

lookahead tokens

M ⇒ 2d parsing table

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT^1$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

leftmost variable

The various information that we get from the above parsing table are shown below:

- The symbols present in the first column of table M i.e., E, E', T, T' and F represent left most non-terminals in the derivation. Let us denote the non-terminal in general by **A**

- The symbols present in the first row such as **id**, +, *, (,) and $ represent next input tokens obtained from the lexical analyzer. Let us denote the terminal in general by **'a'.**

- The entry in a particular row **A** and column '*a'* denoted by M[A, a] may be either blank or a production. This is the production predicted for a variable A when the input symbol is '**a**'. Now, parsing is done as shown below:

  1) If E is on top of the stack and **id** is the input symbol, the parser consults the parsing table M[E, **id**], gets the production E → TE'. Now, the parser removes E from the stack and push TE' in reverse order. So, the entry M[E, **id**] = E → TE' indicates that in the current leftmost derivation, E is the left most non-terminal. When the token **id** is read from the input, we expand the non-terminal E using the production E → TE'.

  2) If E' is on top of the stack and input is ')', the parser consults the parsing table M[E',)] and gets the production E' → ϵ. Now, the parser removes E' from the stack. But, nothing is there on the right side of the production to push. That is, the entry M[E',)] = E' → ϵ indicates that in the current leftmost derivation, E' is the leftmost non-terminal and it is replaced by ϵ. Thus, only the leftmost variable is replaced at each step when the input symbol (lookahead token) is read from the input buffer which results in leftmost derivation. Thus, we say that predictive parsing will mimic the leftmost derivation.

  3) The entry in row E and column '+' is blank. This indicates an error entry and the parser should display appropriate error messages.

Now, the various actions performed by the parser are can be implemented using algorithm. The complete algorithm to parse the string using predictive parser is shown below:

Example 2.21: The predicative parsing algorithm

Input: The string *w* ending with $ (end of the input) and the parsing table

Output: If $w \in L(G)$ i.e., if the input string is generated successfully from the parser, the parse tree using leftmost derivation is constructed. Otherwise, the parser displays error message.

Method: Initially the $ and S are placed on the stack and the input buffer contains input string *w* ending with $. The algorithm shown below uses the parsing table and produce the parse tree. But, instead of displaying the parse tree, we generate the productions that are used to generate the parse tree.

```
Let input pointer points to the first symbol of w

Let X = S be the symbol on top of the stack.

while (X ≠ $)                                  // Stack is not empty
    If ( X == a)                               // stack symbol = input symbol
        Pop X from the stack
        Advance the input pointer.
    else if X is a terminal
        Error()
    else if M[X, a] is blank
        Error()
    else if M[X, a] = X → $Y_1Y_2Y_3$.......$Y_k$
        Output the production X → $Y_1Y_2Y_3$.......$Y_k$
        Remove X from the stack
        Push $Y_1$, $Y_2$,$Y_3$,.......$Y_k$ in reverse order
    endif
    Let X = top stack symbol
end while
```

The initial configuration of the parser

Stack	**Input**
$S	w$

Final configuration of parser, if parsing is successful

Stack	**Input**
$	$

Example 2.22: Consider the following grammar and the corresponding predictive parsing table:

GRAMMAR

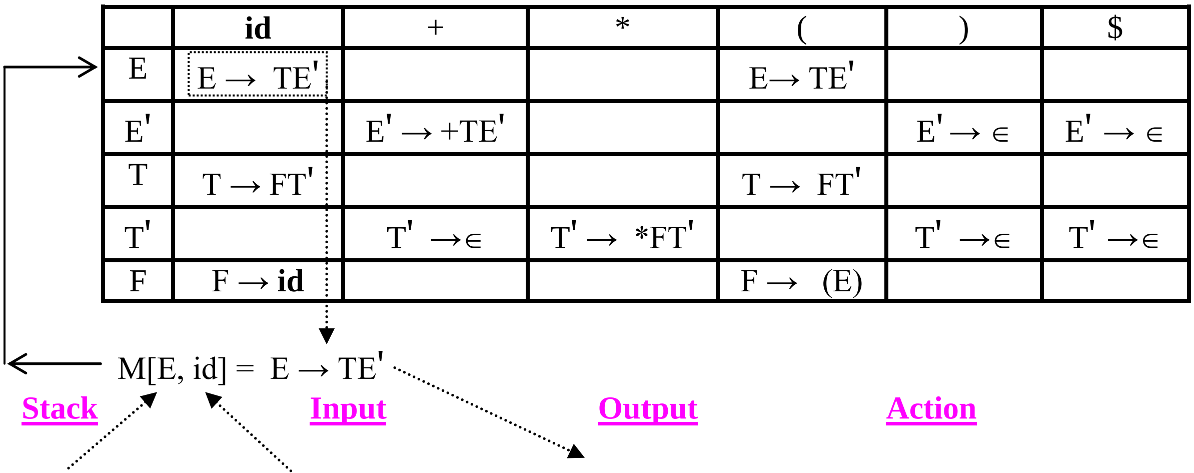
$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \in$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \in$$
$$F \rightarrow (E) \mid id$$

M **Parsing Table**

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E^1 \rightarrow +TE'$			$E' \rightarrow \in$	$E' \rightarrow \in$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \in$	$T^1 \rightarrow *FT'$		$T' \rightarrow \in$	$T' \rightarrow \in$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

Show the sequence of moves made by the predictive parser for the string **id+id*id** during parsing.

Solution: The sequence of moves made by the parser for the string **id+id*id** is shown below:



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

$M[E, id] = E \rightarrow TE'$

Stack	**Input**	**Output**	**Action**
$ E	**id+id*id$**	$E \rightarrow T E'$	[Remove E and push TE' in reverse]
$ E' T	**id+id*id$**	$T \rightarrow F T'$	[Remove T and push FT' in reverse]

$ E' T' F	id+id*id$	F → **id**	[Remove F and push **id**]
$ E' T' **id**	id+id*id$	match(id)	[Remove **id** increment i/p ptr]
$ E' T'	+id*id$	T' → ∈	[Remove T' from stack]
$ E'	+id*id$	E' → +TE'	[Remove E' and push +TE' in reverse]
$ E' T+	+id*id$	match(+)	[Remove + increment i/p ptr]
$ E' T	id*id$	T → FT'	[Remove T and push FT' in reverse]
$ E' T' F	id*id$	F → **id**	[Remove F and push **id]**
$ E' T' **id**	id*id$	match(id)	[Remove id and increment i/p ptr]
$ E' T'	*id$	T^1 → *FT'	[Remove T' and push *FT' in reverse]
$ E' T' F *	*id$	match (*)	[Remove * increment i/p ptr]
$ E' T' F	id$	F → **id**	[Remove F and push **id]**
$ E' T' **id**	id$	match(id)	[Remove id and increment i/p ptr]
$ E' T'	$	T' → ∈	[Remove T' from stack]
$ E'	$	E' → ∈	[Remove E' from stack]
$	$	**ACCEPT**	

Since the stack contains $ and the input pointer points to $, the string **id+id*id** is parsed successfully.

2.9 FIRST and FOLLOW

The predictive parser can be easily constructed once we know FIRST and FOLLOW sets. These sets of symbols help us to construct the predictive parsing table very easily.

2.9.1 Computing FIRST symbols

Now, let us “Define FIRST(α)”

Definition: FIRST(α) is defined as set of terminals that appear in the beginning of derivation derived from α. Formally, FIRST(α) is defined as shown below:

$$\text{FIRST}(\alpha) = \begin{cases} \epsilon & \text{if } \alpha = \epsilon \quad \text{Definition 1} \\ \epsilon & \text{if } \alpha \stackrel{*}{\Rightarrow} \epsilon \quad \text{Definition 2} \\ a & \text{if } \alpha \stackrel{*}{\Rightarrow} a\beta \quad \text{Definition 3} \end{cases}$$

Example 2.23: Compute FIRST sets for each non-terminal in the following grammar

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid id \end{aligned}$$

Solution: The FIRST sets for the given grammar can be computed by obtaining various derivations as shown below:

$$E \Rightarrow T\,E' \Rightarrow F\,T'\,E' \Rightarrow \boxed{(}\,E\;)\,T'E'$$

$$E \Rightarrow T\,E' \Rightarrow F\,T'\,E' \Rightarrow \boxed{id}\,T'E'$$

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

Consider the derivations not used in previous derivation:

$$E' \Rightarrow \boxed{+}\,T\,E' \qquad\qquad T' \Rightarrow \boxed{*}\,F\,T'$$

$$E' \Rightarrow \boxed{\epsilon} \qquad\qquad T' \Rightarrow \boxed{\epsilon}$$

So, **FIRST(E')** = {ε, + }   So, **FIRST(T')** = { ε, * }

Now, the final FIRST sets are written as shown below:

	E	E'	T	T'	F
FIRST	(, id	ε, +	(, id	ε, *	(, id

Now, the question is "What is the use of FIRST sets?" The FIRST sets can be used during predictive parsing while creating the predictive parsing table as shown below:

- Consider the A-production $A \rightarrow \alpha \mid \beta$ and assume FIRST(α) and FIRST(β) are disjoint i.e., FIRST(α) ∩ FIRST(β) = {} which is an empty set.

- If the input symbol obtained from lexical analyzer is *a* and if *a* is in FIRST(α) then use the production $A \rightarrow \alpha$ during parsing.

- If the input symbol obtained from lexical analyzer is *b* and if *b* is in FIRST(β) then use the production $A \rightarrow \beta$ during parsing.

Thus, using FIRST sets we can choose what production to use between the two productions $A \rightarrow \alpha \mid \beta$ when input symbol is *a* or *b*.

Now, let us see "What are the rules to be followed to compute FIRST(X)?" or "What is the algorithm to compute FIRST(X)?" The algorithm or the rules to compute FIRST(X) are shown below:

ALGORITHM FIRST(X)

Rule 1: If $X \rightarrow a\alpha$ where *a* is a terminal, then FIRST(X) ← *a*

Rule 2: If $X \rightarrow \epsilon$, then FIRST(X) ← ϵ

Rule 3: If $X \rightarrow Y_1Y_2Y_3\ldots\ldots\ldots Y_n$ and if $Y_1Y_2Y_3\ldots\ldots\ldots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$, then
FIRST(X) ← non-ϵ symbols in FIRST(Y_i).

Rule 4: If $X \rightarrow Y_1Y_2Y_3\ldots\ldots\ldots Y_n$ and $Y_1Y_2Y_3\ldots\ldots\ldots Y_n \overset{*}{\Rightarrow} \epsilon$, then FIRST(X) ← ϵ

Rule 5: If X is a terminal or , ϵ then FIRST(X) ← X

Now, let us see how FIRST sets are computed by taking some specific examples:

1) Rule 1 is applied if the first symbol on the right hand side of the production is a terminal. If so, then add only the first symbol.
   - Ex 1: if $A \rightarrow aBC$, then FIRST(A) = {a}
   - Ex 2: if $E \rightarrow +TE^1$ then FIRST(E) = {+}
   - Ex 3: if $A \rightarrow abc$, then FIRST(A) = {a}

2) Rule 2 is applied only for ϵ-productions
   - Ex 1: if $A \rightarrow \epsilon$, then FIRST(A) = { ϵ }
   - Ex 2: if $E^1 \rightarrow \epsilon$, then FIRST(E^1) = { ϵ }

3) Rule 3 is applied for all productions not considered in first two steps

Ex : Consider the productions:

$S \rightarrow ABCd$
$A \rightarrow \epsilon \mid +B$
$B \rightarrow \epsilon \mid *B$
$C \rightarrow \epsilon \mid \%B$

FIRST(A), FIRST(B), FIRST(C) are computed using rules 1 and 2 as shown below:

	S	A	B	C
FIRST		ϵ, +	ϵ, *	ϵ, %

To compute FIRST(S) consider the production S → ABCd and apply rule 3 as shown below:

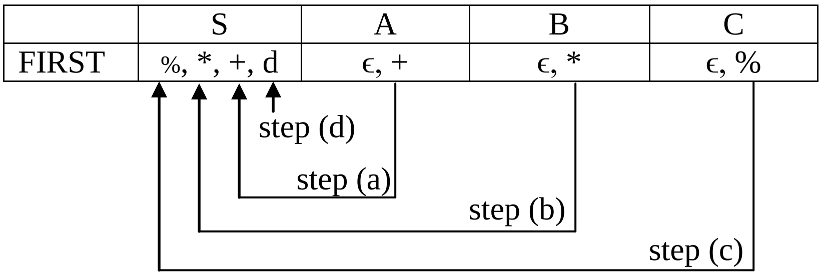
a) $S \rightarrow ABCd$ Add non- ϵ symbols of FIRST(A) to FIRST(S)

b) $S \rightarrow \boxed{A}BCd$ Since $A \overset{*}{\Rightarrow} \epsilon$, add non- ϵ symbols of FIRST(B) to FIRST(S)

c) $S \rightarrow \boxed{AB}Cd$ Since $AB \overset{*}{\Rightarrow} \epsilon$, add non-ϵ symbols of FIRST(C) to FIRST(S)

d) $S \rightarrow \boxed{ABC}d$ Since $ABC \overset{*}{\Rightarrow} \epsilon$, add non-ϵ symbols of FIRST(d) to FIRST(S)

So, all the above actions are pictorially represented as shown below:



	S	A	B	C
FIRST	%, *, +, d	ϵ, +	ϵ, *	ϵ, %

4) Rule 4 is applied for all productions whose RHS gives ϵ

Ex : Consider the productions:

$S \rightarrow ABC$
$A \rightarrow \epsilon \mid +B$
$B \rightarrow \epsilon \mid *B$
$C \rightarrow \epsilon \mid \%B$

FIRST(A), FIRST(B), FIRST(C) are computed using rules 1 and 2 as shown below:

	S	A	B	C
FIRST		ϵ, +	ϵ, *	ϵ, %

To compute FIRST(S) consider the production S → ABC and apply rule 3 as shown below:

a) $S \rightarrow ABC$ Add non- ϵ symbols of FIRST(A) to FIRST(S)

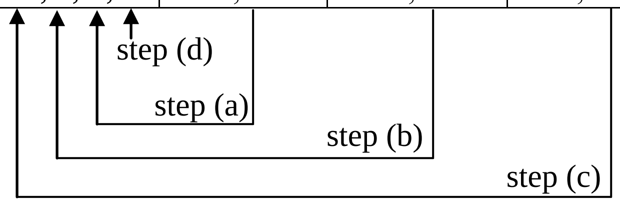
b) $S \rightarrow \boxed{A}BC$ Since A $\overset{*}{\Rightarrow}$ ϵ, add non- ϵ symbols of FIRST(B) to FIRST(S)

c) $S \rightarrow \boxed{AB}C$ Since AB $\overset{*}{\Rightarrow}$ ϵ, add non-ϵ symbols of FIRST(C) to FIRST(S)

d) $S \rightarrow \boxed{ABC}$ Since ABC $\overset{*}{\Rightarrow}$ ϵ, add ϵ to FIRST(S)

So, all the above actions are pictorially represented as shown below:

	S	A	B	C
FIRST	%, *, +, ϵ	ϵ, +	ϵ, *	ϵ, %



5) Rule 5 is applied only for terminals.

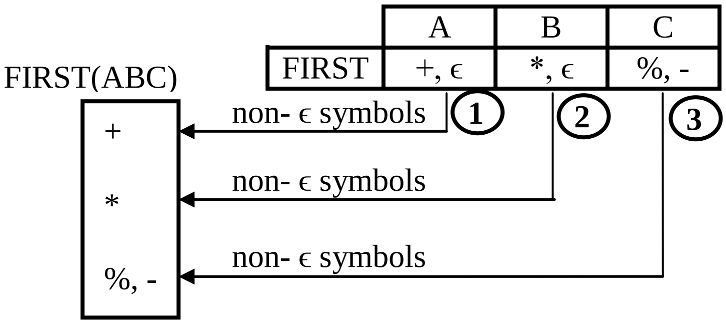
- Ex 1: + is terminal. So, FIRST(+) = { + }
- Ex 2: *a* is a terminal. So, FIRST(a) = {a}
- Ex 3: **id** is a terminal. So, FIRST(**id**) = {**id**}

Note: FIRST($X_1X_2X_3$.........X_n) can be computed as follows :

1) FIRST($X_1X_2X_3$.........X_n) ← Non- ϵ symbols of FIRST(X_1)
2) if FIRST(X_1) = ϵ, then FIRST($X_1X_2X_3$.........X_n) ← FIRST(X_2) - ϵ
3) if FIRST(X_1) and FIRST(X_2) = ϵ then FIRST($X_1X_2X_3$.........X_n) ← FIRST(X_3) - ϵ
   ……..
   ……..
4) If FIRST(X_1), FIRST(X_2),….. and FIRST(X_n) = ϵ, then FIRST(X_1X_2...X_n) ← ϵ

Example 2.24: Let FIRST(A) = {+, ϵ }, FIRST(B) = { *, ϵ } and FIRST(C) = { %, - } Compute FIRST(ABC)

Solution: Using FIRST(A), FIRST(B) and FIRST(C), the FIRST(ABC) can be obtained as shown below:



① Add non- ϵ symbols of FIRST(A)

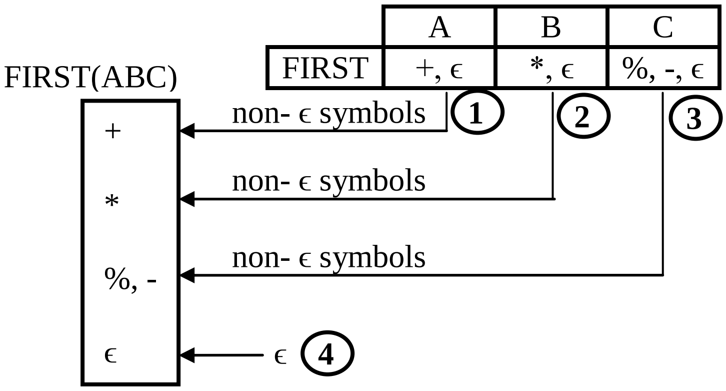
② Since FIRST(A) contains ϵ, we add non- ϵ symbols of FIRST(B)

③ Since FIRST(A) and FIRST(B) contains ϵ, we add non- ϵ symbols of FIRST(C)

So, $\boxed{\text{FIRST(ABC)} = \{+, *, \%, -\}}$

Example 2.25: Let FIRST(A) = {+, ϵ }, FIRST(B) = { *, ϵ } and FIRST(C) = { %, -, ϵ } Compute FIRST(ABC)

Solution: Using FIRST(A), FIRST(B) and FIRST(C), the FIRST(ABC) can be obtained as shown below:



① Add non- ϵ symbols of FIRST(A)

② Since FIRST(A) contains ϵ, we add non- ϵ symbols of FIRST(B)

③ Since FIRST(A) and FIRST(B) contains ϵ, we add non- ϵ symbols of FIRST(C)

④ Since FIRST(A), FIRST(B) and FIRST(C) contains ϵ, we add ϵ symbol

So, $\boxed{\text{FIRST(ABC)} = \{+, *, \%, -, \epsilon\}}$

2.9.2 Computing FOLLOW symbols

Once we know, how to compute FIRST sets, let us concentrate on how to compute FOLLOW sets. Before proceeding further, let us "Define FOLLOW(A)?"

Definition: The FOLLOW(A) for a non-terminal A is defined as the set of terminals *a* that will appear immediately to the right of A in some sentential form. That is, the set of terminals *a* such that there exists a derivation of the form:

$$S \overset{*}{\Rightarrow} \alpha A a \beta$$

for some α and β. If A is appeared as the last symbol in some sentential form, then place $ into FOLLOW(A) where the symbol $ is treated as "endmarker" symbol.

Now, let us see "What is the algorithm to compute FOLLOW(A)?" The algorithm to compute FOLLOW(A) is shown below:

ALGORITHM FOLLOW(A)

Rule 1: FOLLOW(S) ← $ where S is the start symbol.

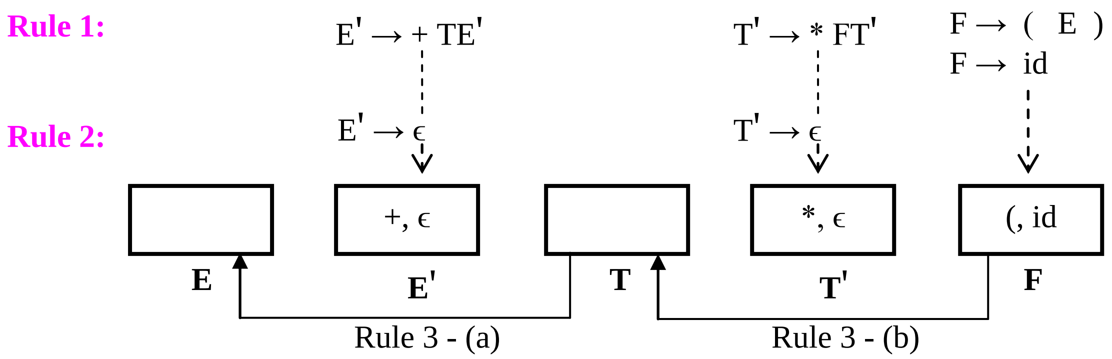
Rule 2: If $A \rightarrow \alpha B \beta$ is a production and $\beta \neq \in$ then FOLLOW(B) ← non-∈ symbols in FIRST(β)

Rule 3: If $A \rightarrow \alpha B \beta$ is a production and $\beta \overset{*}{\Rightarrow} \in$, then FOLLOW(B) ← FOLLOW(A)

Example 2.26: Compute FIRST and FILLOW sets for the following grammar:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \in \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \in \\ F &\rightarrow (E) \mid id \end{aligned}$$

a) **Computing FIRST sets:** The FIRST sets can be computed as shown below:

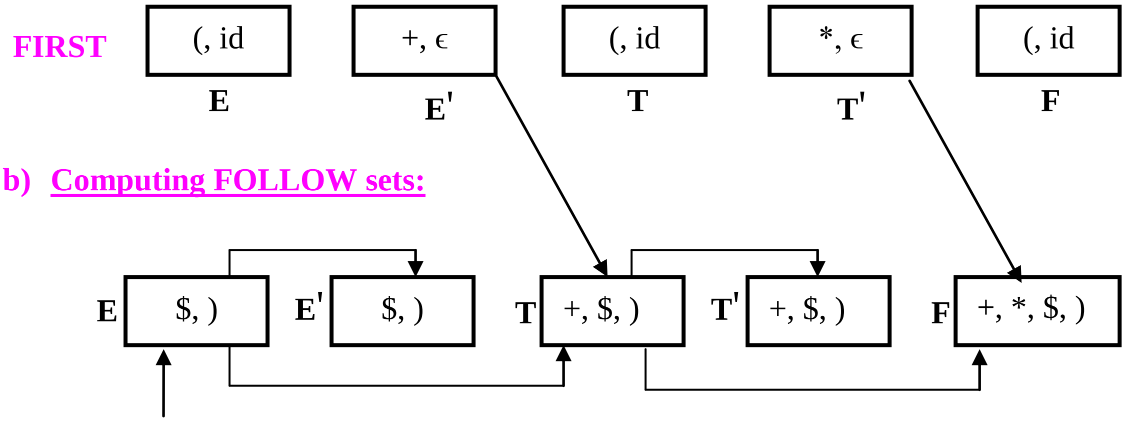


Rule 3: Consider the productions not considered earlier and obtain FIRST sets as shown below:

a) E → T E' Add "FIRST(T) - ϵ" to FIRST(E) i.e., draw an edge from T to E in above figure.

b) T → F T' Add "FIRST(F) - ϵ" to FIRST(T) i.e., draw an edge from F to T in above figure.

In the above figure, transfer FIRST(T) to FIRST(E) and from FIRST(F) to FIRST(T). So, the final FIRST sets are shown below:



b) **Computing FOLLOW sets:**

Rule 1: $ is placed in FOLLOW(E) since E is the start symbol.

Rule 2 &3 : Apply rule 2 and 3 for every production of the form A → αBβ where B is a non-terminal. In the first column shown below, copy from FIRST(β) to FOLLOW(B) and in the second column copy from FOLLOW(A) to FOLLOW(B).

Rule 2 (β ≠ ϵ) FOLLOW(B) ← FIRST(β) - ϵ Copy from right to left (Put arrow from right left on RHS of the production)	**Rule 3** (β ≛ ϵ) FOLLOW(A) → FOLLOW(B) Copy from left to right (Put arrow from LHS of production to RHS)
E → T E' A → α B β	E → T E' A → α B β
E → T E' A → α B β Rule 2 not applicable	E → T E' A → α B β
E^1 → + T E' A → α B β	E' → + T E' A → α B β
E^1 → $\boxed{+\ T}$ E' A → α B β Rule 2 not applicable	E' → $\boxed{+\ T}$ E' A → α B β
T → F T' A → α B β	T → F T' A → α B β
T → F T' A → α B β Rule 2 not applicable	T → F T' A → α B β
T' → * F T' A → α B β	T' → * F T' A → α B β
T' → $\boxed{*\ F}$ T' A → α B β Rule 2 not applicable	T' → $\boxed{*\ F}$ T' A → α B β
F → (E) A → αB β	F → (E) A → αB β Rule 3 not applicable

Now, let us see "What are the steps to be followed while constructing the predictive parser?" The various steps to be followed while constructing the predictive parser are shown below:

- If the grammar is ambiguous, eliminate ambiguity from the grammar
- If the grammar has left recursion, eliminate left recursion
- If the grammar has two or more alternatives having common prefix, then do left-factoring
- The resulting grammar is suitable for constructing predictive parsing table

2.9.3 Constructing predictive parsing table

Now, using FIRST and FOLLOW sets, we can easily construct the predictive parsing table and the productions are entered into the table M[A, a] where

- M is a 2-dimensional array representing the predictive parsing table
- A is a non-terminal which represent the row values
- *a* is a terminal or $ which is endmarker and represent the column values

Now, let us "Write the algorithm to construct the predictive parsing table" The complete algorithm is shown below:

ALGORITHM Predictive_Parsing_Table(G, M)
Input : Grammar G

Output : Predictive parsing table M

Procedure : For each production A → α of grammar G apply the following rules

1) For each terminal *a* in FIRST(α), add A→ α to M[A, a]

2) If FIRST(α) contains ∈, for each symbol *b* in FOLLOW(A), add A→ α to M[A, b]

Example 2.27: Obtain the predictive parsing table for the following grammar

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \in$$
$$T \rightarrow FT'$$
$$T^1 \rightarrow *FT' \mid \in$$
$$F \rightarrow (E) \mid id$$

Solution: The FIRST and FOLLOW sets of each non-terminal of the given grammar are shown below: (See example 2.26 for details)

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

For every production of the form A → α, we compute FIRST(α) and entries of the parsing table can be done as shown below:

Productions **A → α**	**a = FIRST(α)**	**M[A, a] = A → α**	**Rule**
$E \rightarrow \underbrace{TE'}$ $A \quad \alpha$	(, id	M [E, ( ] = E → TE' M [E, id] = E → TE'	1
$E' \rightarrow \underbrace{+TE'}$ $A \quad \alpha$	+	M [E', +] = E' → +TE'	1
$E' \rightarrow \in$ $A \quad \alpha$	∈	M [E',)] = E' → ∈ M [E', $] = E' → ∈	2
$T \rightarrow \underbrace{FT'}$ $A \quad \alpha$	(,id	M [T, ( ] = T → FT' M [T, id] = T → FT'	2
$T' \rightarrow \underbrace{*FT'}$ $A \quad \alpha$	*	M [T', *] = T' → *FT'	2
$T' \rightarrow \in$ $A \quad \alpha$	∈	M [T', +] = T' → ∈ M [T',)] = T' → ∈ M [T', $] = T' → ∈	3
$F \rightarrow \underbrace{(E)}$ $A \quad \alpha$	(	M [F, ( ] = F → (E)	2
F → **id** $A \quad \alpha$	**id**	M [F, **id**] = F → **id**	2

The parsing table is shown below:

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

Note: Since there are no multiple entries in the parsing table, the given grammar is called LL(1) grammar. If multiple entries are present in the parsing table, the grammar is not LL(1). The predictive parser accepts only the language generated from LL(1) grammar.

2.10 LL (1) Grammars

In this section, let us see "What is LL (1) grammar?"

Definition: The grammar from which a predictive parser, that is, recursive descent parser without backtracking is constructed is called LL(1) grammar where

- The first L stands for left-to-right scan of the input
- The second L stands for leftmost derivation. So, the predictive parsers always mimic the leftmost derivation.
- The digit 1 indicates number of tokens to lookahead.

In LL(1) parsing technique or predictive parsing if two or more alternative productions are there, the predictive parser also called LL(1) parser chooses the correct production by guessing using one lookahead token.

Now, let us see "What grammars are not LL(1)?" The following grammars are not LL(1) grammars:

- Ambiguous grammar is not LL(1)
- Left recursive grammar is not LL(1)
- The grammar which is not left factored (that is, if two or more alternative productions have common prefix), the grammar is not LL(1)
- The grammar that results in multiple entries in the parsing table is not LL(1).

Now, the question is "How to check whether a given grammar is LL(1) or not without constructing the predictive parser?" The grammar is said to be LL(1) if following two conditions are satisfied:

- For every production of the form $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \ldots\ldots.\alpha_n$:

$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) \text{ must be empty for all } i, j \geq n \text{ where } i \neq j$$

- For every non-terminal A such that FIRST(A) contains ϵ:

$$\text{FIRST}(A) \cap \text{FOLLOW}(A) \text{ must be empty}$$

Example 2.28: Compute FIRST and FOLLOW symbols and predictive parsing table for the following grammar:

$$S \rightarrow iCtS \mid iCtSeS \mid a$$
$$C \rightarrow b$$

Is the following grammar LL(1)?

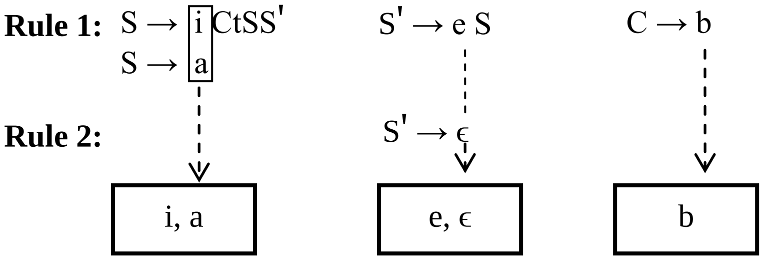
Solution: We know that the grammar is not left-factored since, two productions have common prefix "iCtS". So, it is necessary to do the left-factoring for the given grammar. The left-factored grammar (for details refer section 2.8.5, example 2.19) is shown below:

$S \rightarrow iCtSS' \mid a$
$S' \rightarrow \epsilon \mid eS$
$C \rightarrow b$

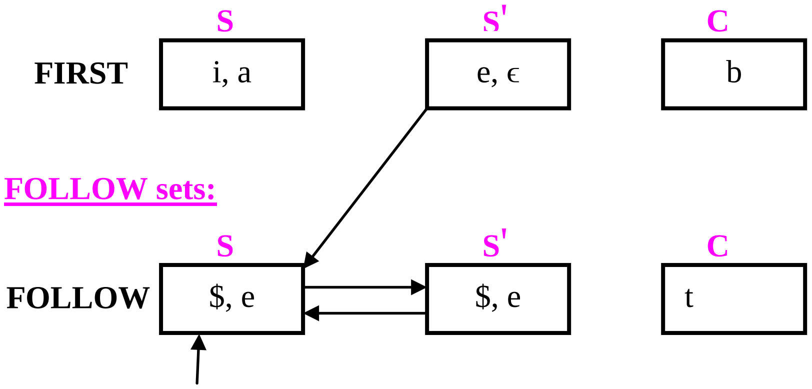
The following procedure is used:

- Compute FIRST sets and FOLLOW sets
- Check whether the grammar is LL(1) or not
- Obtain the parsing table

Step 1: The first symbols can be computed as shown below:



Rule 3: This rule is not applied, since all productions are already considered when we apply first two rules. So, the final FIRST sets are shown below:



Rule 1: $ is placed in FOLLOW(S) since S is the start symbol.

Rule 2 &3 : Apply rule 2 and 3 for every production of the form A → αBβ where B is a non-terminal. In the first column shown below, copy from FIRST(β) to FOLLOW(B) and in the second column copy from FOLLOW(A) to FOLLOW(B).

Rule 2 ($\beta \neq \epsilon$) FOLLOW(B) ← FIRST(β) - ϵ		**Rule 3** ($\beta \overset{*}{\Rightarrow} \epsilon$) FOLLOW(A) → FOLLOW(B)	
$S \rightarrow i\ \overset{\downarrow t}{C}\ \boxed{t\ S\ S'}$ $A \rightarrow \alpha\ B\ \ \beta$		rule 3 not applicable	
$S \rightarrow \boxed{i\ C\ t}\ \overset{\downarrow}{S}\ S'$ $A \rightarrow \alpha\ B\ \beta$		$S \rightarrow \boxed{i\ C\ t}\ \overset{\downarrow}{S}\ S'$ $A \rightarrow \alpha\ B\ \beta$	
$S \rightarrow \boxed{i\ C\ t\ S}\ S'$ $A \rightarrow \alpha\ B\ \beta$	rule 2 not applicable	$S \rightarrow \boxed{i\ C\ t\ S}\ \overset{\downarrow}{S'}$ $A \rightarrow \alpha\ B\ \beta$	
$S' \rightarrow e\ S$ $A \rightarrow \alpha\ B\ \beta$	rule 2 not applicable	$S' \rightarrow e\ \overset{\downarrow}{S}$ $A \rightarrow \alpha\ B\ \beta$	

Note: The productions S → a and C → b are not considered while computing FOLLOW since there are no variables in those productions. So, the FIRST and FOLLOW sets for the left-factored grammar are shown below:

	S	S'	C
FIRST	a, i	e, ϵ	b
FOLLOW	$, e	$, e	t

To check whether the grammar is LL(1) or not: Without constructing the predictive parser also we can check whether the grammar is LL(1) or not. If the grammar is LL(1), the following two conditions must be satisfied:

Condition 1:

For a given production $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \ldots\ldots.\alpha_n$	Condition to be satisfied $\text{FIRST}(\alpha_1) \cap \text{FIRST}(\alpha_2) \cap \ldots.\text{FIRST}(\alpha_n) = \phi$
$S \rightarrow i\ C\ t\ S\ S' \mid a$	$\text{FIRST}(iCtSS') \cap \text{FIRST}(a)$ $\{i\} \cap \{a\} = \phi$
$S' \rightarrow \epsilon \mid eS$	$\text{FIRST}(\epsilon) \cap \text{FIRST}(eS)$ $\{\epsilon\} \cap \{e\} = \phi$

Note: *Condition 1 is satisfied*

Condition 2:

If FIRST(A) contains ϵ	Condition to be satisfied is FIRST(A) ∩ FOLLOW(A) = ϕ

FIRST(S') has ϵ	FIRST(S') ∩ FOLLOW(S')
	{ e, ϵ } ∩ {$, e} = {e}

Note: *Condition 2 is not satisfied:*

Since one of the condition fails, the given grammar is not LL(1). For a grammar to be LL(1), the both the conditions must be satisfied.

Construction of predictive parsing table: For every production of the form A → α, we compute FIRST(α) and entries of the parsing table can be done as shown below:

Productions **A → α**	**a = FIRST(α)**	**M[A, a] = A → α**	**Rule**
$\underset{A}{S} \rightarrow \underbrace{iCtSS'}_{\alpha}$	i	M [S, i] = S → iCtSS'	1
$\underset{A}{S} \rightarrow \underset{\alpha}{a}$	a	M [S, a] = S → a	1
$\underset{A}{S'} \rightarrow \underbrace{eS}_{\alpha}$	e	M [S', e] = S' → eS	1
$\underset{A}{S'} \rightarrow \underset{\alpha}{\epsilon}$	ϵ	M [S', e] = S' → ϵ M [S', $] = S' → ϵ	2
$\underset{A}{C} \rightarrow \underset{\alpha}{b}$	b	M [C, b] = C → b	1

The parsing table is shown below:

	a	**b**	**e**	**i**	**t**	**$**
S	S → a			S → iCtSS'		
S^1			S' → eS S' → ϵ			S' → ϵ
C		C → b				

Example 2.29: Given the following grammar:

$$S \rightarrow a \mid (L)$$
$$L \rightarrow L\,,\,S \mid S$$

a) Is the grammar suitable for predictive parser?
b) Do the necessary changes to make it suitable for LL(1) parser
c) Compute FIRST and FOLLOW sets for each non-terminal
d) Obtain the parsing table and check whether the resulting grammar is LL(1) or not.
e) Show the moves made by the predictive parser on the input "(a , (a , a))"

Solution: The given grammar is shown below:

$$S \rightarrow a \mid (L)$$
$$L \rightarrow L\,,\,S \mid S$$

a) Consider the production: $L \rightarrow L\,,\,S$

Since the first symbol on RHS of the production is same as the symbol on LHS of the production, the given grammar is having left-recursion and hence, *it is not suitable for predictive parser.*

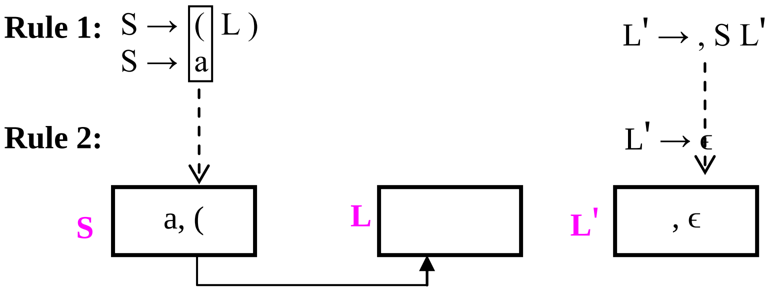
b) To make it suitable for LL(1) parser or predictive parser, we need to eliminate left-recursion as shown below: (For details refer section 2.8.4)

Left recursive productions	**Right recursive productions**
$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots\ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots.. \mid \beta_m$	$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots\ldots.. \mid \beta_m A'$ $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots\ldots \mid \alpha_n A' \mid \epsilon$
1) $S \rightarrow a \mid (L)$	$S \rightarrow a \mid (L)$
2) $L \rightarrow L\boxed{,\,S} \mid S$ $\downarrow \quad \downarrow \; \downarrow \quad \downarrow$ $A \rightarrow A \;\; \alpha_1 \mid \beta_1$	$L \rightarrow SL'$ $L' \rightarrow \,,\,S\,L' \mid \epsilon$

The final grammar obtained after eliminating left recursion can be written as shown below:

$$\boxed{\begin{array}{l} S \rightarrow a \mid (L) \\ L \rightarrow SL' \\ L' \rightarrow \,,\,S\,L' \mid \epsilon \end{array}}$$

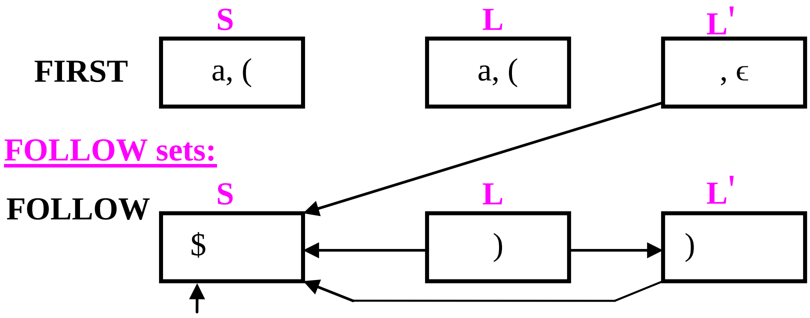
c) **Computing FIRST and FOLLOW:** The first set can be computed as shown below:



Rule 3: Consider the productions not considered earlier and obtain FIRST sets as shown below:

a) L → S L' Add “FIRST(S) - ϵ” to FIRST(L)

In the above figure, transfer FIRST(S) to FIRST(L). So, the final FIRST sets are shown below:



FOLLOW sets:

Rule 1: $ is placed in FOLLOW(S) since S is the start symbol.

Rule 2 &3 : Apply rule 2 and 3 for every production of the form A → αBβ where B is a non-terminal. In the first column shown below, copy from FIRST(β) to FOLLOW(B) and in the second column copy from FOLLOW(A) to FOLLOW(B).

Rule 2 ($\beta \neq \epsilon$) FOLLOW(B) ← FIRST(β) - ϵ	**Rule 3** ($\beta \overset{*}{\Rightarrow} \epsilon$) FOLLOW(A) → FOLLOW(B)
) S → (L) A → α B β	rule 3 not applicable
L → S L' A → α B β	L → S L' A → α B β

$L \to \quad S \;\; L'$ rule 2 not applicable $A \to \quad \alpha \;\; B \;\; \beta$	$L \to \quad S \;\; L'$ $A \to \quad \alpha \;\; B \;\; \beta$
$L^1 \to \;\; , S \;\; L'$ $A \to \;\; \alpha \, B \;\; \beta$	$L' \to \;\; , S \;\; L'$ $A \to \;\; \alpha \, B \;\; \beta$
$L^1 \to \boxed{, S} L'$ rule 2 not applicable $A \to \quad \alpha \;\; B \;\; \beta$	$L' \to \boxed{, S} L'$ results in self-loop and hence discard $A \to \quad \alpha \;\; B \;\; \beta$

Note: The productions S → a and L' → ϵ are not considered while computing FOLLOW since they do not have non-terminals in those productions. So, the FIRST and FOLLOW sets for the left-factored grammar are shown below:

	S	L	L'
FIRST	a (	a (	, ϵ
FOLLOW	, $)	)	)

d) **Construction of parsing table:** For every production of the form A → α, we compute FIRST(α) and entries of the parsing table can be done as shown below:

Productions **A → α**	**a = FIRST(α)**	**M[A, a]**	**=**	**A → α**	**Rule**
$S \to a$ $A \quad \alpha$	a	M [S, a]	=	S → a	1
$S \to \underbrace{(\,L\,)}$ $A \quad \alpha$	(	M [S, ( ]	=	S → (L)	1
$L \to \underbrace{SL'}$ $A \quad \alpha$	a (	M [L, a] M [L, ( ]	= =	L → SL' L → SL'	1
$L' \to \epsilon$ $A \quad \alpha$	ϵ	M [L',)]	=	L' → ϵ	2
$L' \to \underbrace{, SL'}$ $A \quad \alpha$	,	M [L', ‘,’]	=	L' → , SL'	1

The above entries can be entered into parsing table as shown below:

	(	)	a	,	\$
S	$S \rightarrow (L)$		$S \rightarrow a$		
L	$L \rightarrow SL^1$		$L \rightarrow SL^1$		
L^1		$L^1 \rightarrow \in$		$L^1 \rightarrow , SL^1$	

Since there are no multiple entries in the parse table, the resulting grammar obtained after eliminating left recursion is LL(1).

e) The moves made by the predictive parser on the input “(a , (a , a))” is shown below:

Stack	Input	Output	Action
$\$ S$	$(a , (a , a)) \$$	$S \rightarrow (L)$	[Remove S and push (L) in reverse]
$\$) L ($	$(a , (a , a)) \$$	Match (	Pop (and increment i/p pointer
$\$) L$	$a , (a , a)) \$$	$L \rightarrow SL'$	[Remove L and push SL' in reverse]
$\$) L' S$	$a , (a , a)) \$$	$S \rightarrow a$	[Remove S and push *a* in reverse]
$\$) L' a$	$a , (a , a)) \$$	Match a	Pop *a* and increment i/p pointer
$\$) L'$	$, (a , a)) \$$	$L' \rightarrow , SL'$	Remove L' and push **,SL'** in reverse
$\$) L' S ,$	$, (a , a)) \$$	Match ,	Pop ‘,’ and increment i/p pointer
$\$) L' S$	$(a , a)) \$$	$S \rightarrow (L)$	Remove S and push (L) in reverse
$\$) L') L ($	$(a , a)) \$$	Match (	Pop (and increment i/p pointer
$\$) L') L$	$a , a)) \$$	$L \rightarrow SL'$	Remove L and push SL' in reverse
$\$) L') L'S$	$a , a)) \$$	$S \rightarrow a$	Remove S and push *a* in reverse
$\$) L') L'a$	$a , a)) \$$	$S \rightarrow a$	Pop *a* and increment i/p pointer
$\$) L') L'$	$, a)) \$$	$L' \rightarrow , SL'$	Remove L' and push **,SL'** in reverse

$) L') L' S ,	, a)) $	Match ,	Pop ‘,’ and increment i/p pointer
$) L') L' S	a)) $	S → a	Remove S and push *a*
$) L') L' a	a)) $	Match *a*	Pop *a* and increment i/p pointer
$) L') L'	)) $	L' → ∈	Pop L'
$) L')	)) $	Match)	Pop) and increment i/p pointer
$) L'	) $	L' → ∈	Pop L'
$)	) $	Match)	Pop) and increment i/p pointer
$	$	**Accept**	

Note: Since stack is empty and i/p pointer also points to $ which is endmarker, parsing is successful

Example 2.30: Given the following grammar:

E → 5 + T | 3 – T
T → V | V*V | V+V
V → a | b

a) Is the grammar suitable for predictive parser?
b) What is the use of left-factoring? Do the left factoring for the above grammar
c) Compute FIRST and FOLLOW sets for each non-terminal
d) Without constructing the parsing table, check whether the grammar is LL(1) or not.
e) By constructing the parsing table, check whether the grammar is LL(1) or not.

Solution: The given grammar is shown below:

E → 5 + T | 3 – T
T → V | V*V | V+V
V → a | b

a) The E-productions and V productions are suitable for parsing. But, consider the production:

T → V | V*V | V+V

In the T-production, one or more productions have a common prefix V and hence the given grammar is not left-factored grammar. So, *the given grammar is not suitable for predictive parser.*

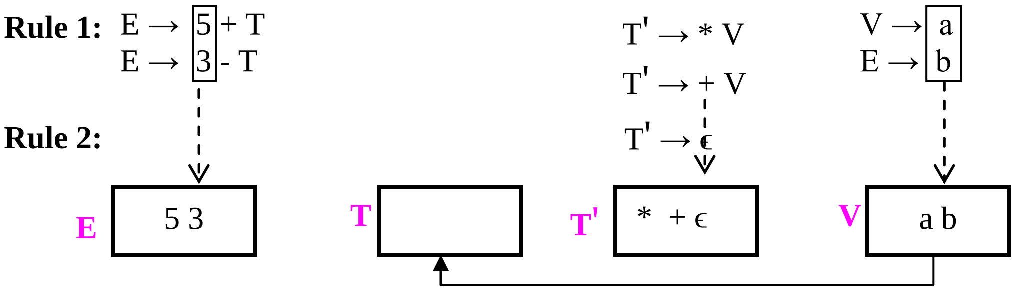
b) To make it suitable for LL(1) parser or predictive parser, we need to do left factoring (For details refer section 2.8.5). If an A-production has two or more alternate productions and they have a common prefix, then the parser has some confusion in selecting the appropriate production for expanding the non-terminal A. So, left factoring is must for top down parser. This can be done as shown below:

Given productions	Left-factored productions
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots\ldots \alpha\beta_n \mid \gamma$	$A \rightarrow \alpha A' \mid \gamma$ $A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots \beta_n$
1) $E \rightarrow 5 + T \mid 3 - T$	$E \rightarrow 5 + T \mid 3 - T$
2) $\underbrace{T}_{A} \rightarrow \underbrace{V}_{\alpha}\ \underset{\beta_1}{\epsilon} \mid \underbrace{V}_{\alpha}\ \underbrace{* V}_{\beta_2} \mid \underbrace{V}_{\alpha}\ \underbrace{+ V}_{\beta_3}$	$T \rightarrow V T'$ $T' \rightarrow \epsilon \mid * V \mid + V$
3) $V \rightarrow a \mid b$	$V \rightarrow a \mid b$

So, the final grammar which is obtained after doing left-factoring is shown below:

$$\begin{aligned}
&E \rightarrow 5 + T \mid 3 - T\\
&T \rightarrow V T'\\
&T' \rightarrow \epsilon \mid * V \mid + V\\
&V \rightarrow a \mid b
\end{aligned}$$

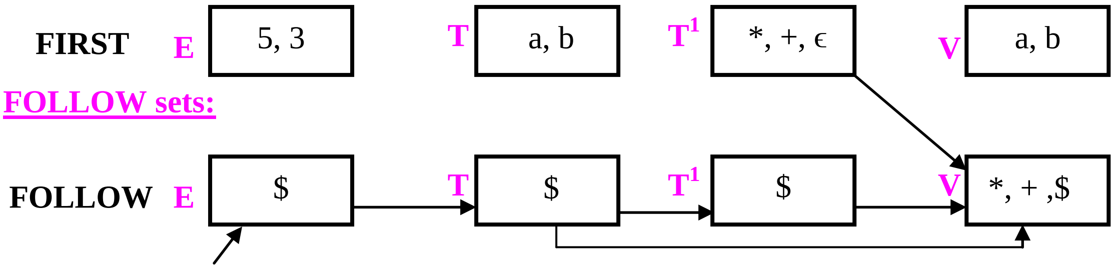
c) **Computing FIRST and FOLLOW:** The first set can be computed as shown below:



Rule 3: Consider the productions not considered earlier and obtain FIRST sets as shown below:

b) $T \rightarrow V\ T'$ Add “FIRST(V) - ϵ” to FIRST(T)

In the above figure, transfer FIRST(V) to FIRST(T). So, the final FIRST sets are shown below:



Rule 1: Place $ into FOLLOW(S) since S is the start symbol.

Rule 2 &3 : Apply rule 2 and 3 for every production of the form A → αBβ where B is a non-terminal. In the first column shown below, copy from FIRST(β) to FOLLOW(B) and in the second column copy from FOLLOW(A) to FOLLOW(B).

Rule 2 (β ≠ ϵ) FOLLOW(B) ← FIRST(β) - ϵ	**Rule 3** ($\beta \overset{*}{\Rightarrow} \epsilon$) FOLLOW(A) → FOLLOW(B)
E → $\boxed{5 +}$ T A→ α B β rule 2 not applicable	E → $\boxed{5 +}$ T A→ α B β
E → $\boxed{3 -}$ T A→ α B β rule 2 not applicable	E → $\boxed{3 -}$ T A→ α B β
T → V T^1 A → α B β	T → V T^1 A → α B β
T → V T^1 A → α B β rule 2 not applicable	T → V T^1 A → α B β
T^1 → * V A → α B β rule 2 not applicable	T^1 → * V A → α B β
T^1 → + V A → α B β rule 2 not applicable	T^1 → + V A → α B β

Note: The productions $T^1 \to \epsilon$ and $V \to a \mid b$ are not considered while computing FOLLOW since there are no non-terminals in those productions.

So, the FIRST and FOLLOW sets for the left-factored grammar are shown below:

	E	T	T^1	V
FIRST	5, 3	a, b	*,+ , ϵ	a,b
FOLLOW	$	$	$	*,+,$

d) Now, for the grammar to be LL(1) the following two conditions must be satisfied:

a. The first condition has to be satisfied:

Production $A \to \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \ldots\ldots.$	Condition to be satisfied $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \phi$
$E \to 5 + T \mid 3 - T$	$\text{FIRST}(5 + T) \cap \text{FIRST}(3 - T) = \phi$
$V \to a \mid b$	$\text{FIRST}(a) \cap \text{FIRST}(b) = \phi$

Observe that the first condition is satisfied

b. The second condition has to be satisfied:

If $\text{FIRST}(A) = \epsilon$	Condition to be satisfied $\text{FIRST}(A) \cap \text{FOLLOW}(A) = \phi$
If $\text{FIRST}(T^1) = \epsilon$	$\text{FIRST}(T^1) \cap \text{FOLLOW}(T^1)$ $\{*, +, \epsilon\} \cap \{\$\} = \phi$

Observe that the second condition is satisfied

Since, both conditions are satisfied, **the resulting grammar is LL(1)**

e) **Construction of Parsing table:** For every production of the form $A \to \alpha$, we compute $\text{FIRST}(\alpha)$ and entries of the parsing table can be done as shown below:

Productions **$A \to \alpha$**	**$a = \text{FIRST}(\alpha)$**	**$M[A, a] = A \to \alpha$**	**Rule**
$E \to 5 + T$ $A \quad \alpha$	5	$M[E, 5] = E \to 5 + T$	1

$E \to 3 - T$ A α	3	M [E, 3] = $E \to 3 - T$	1
$T \to \underbrace{VT^1}$ A α	a, b	M [T, a] = $T \to VT^1$ M [T, b] = $T \to VT^1$	1
$T^1 \to \in$ A α	$\in$	M [T^1, \$] = $T^1 \to \in$	2
$T^1 \to *V$ A α	*	M [T^1, *] = $T^1 \to *V$	1
$T^1 \to +V$ A α	+	M [T^1, +] = $T^1 \to +V$	1
$V \to a$ A α	a	M [V, a] = $V \to a$	1
$V \to b$ A α	b	M [V, b] = $V \to b$	1

The parsing table is shown below:

	5	3	a	b	*	+	\$
E	$E \to 5 + T$	$E \to 3 - T$					
T			$T \to VT^1$	$T \to VT^1$			
T^1					$T^1 \to *V$	$T^1 \to +V$	$T^1 \to \in$
V			$V \to a$	$V \to b$			

Since there are no multiple entries in the parse table, the resulting grammar obtained after doing left factoring is LL(1).

Example 2.31: Given the following grammar:

$Z \to d \mid XYZ$
$Y \to \epsilon \mid c$
$X \to Y \mid a$

a) Compute FIRST and FOLLOW sets for each non-terminal

b) Without constructing the parsing table, check whether the grammar is LL(1) or not.

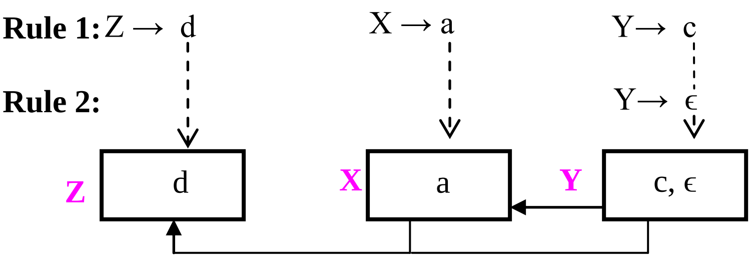
c) By constructing the parsing table, check whether the grammar is LL(1) or not.

Solution: The given grammar is shown below:

$Z \to d \mid XYZ$
$Y \to \epsilon \mid c$
$X \to Y \mid a$

a) The FIRST and FOLLOW sets are computed as shown below:

Step 1: The first symbols can be computed as shown below:



Rule 3: Consider the productions not considered earlier and obtain FIRST sets as shown below:

1) Z → XYZ — Add "FIRST(X) - ϵ" to FIRST(Z)

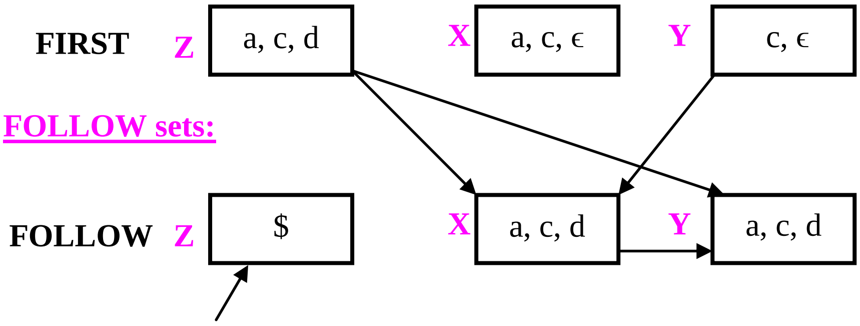
2) Z → XYZ — Since FIRST(X) has ϵ, add "FIRST(Y) - ϵ" to FIRST(Z)

3) Z → XYZ — Since FIRST(X) and FIRST(Y) has ϵ, add "FIRST(Z) - ϵ" to FIRST(Z)

4) X → Y — Add "FIRST(Y) - ϵ" to FIRST(X)

5) X → Y — Since $Y \overset{*}{\Rightarrow} \epsilon$, add ϵ to FIRST(X)

So, the final FIRST sets are shown below:



FOLLOW sets:

Rule 1: Place $ into FOLLOW(Z) since Z is the start symbol.

Rule 2 &3 : Apply rule 2 and 3 for every production of the form A → αBβ where B is a non-terminal. In the first column shown below, copy from FIRST(β) to FOLLOW(B) and in the second column copy from FOLLOW(A) to FOLLOW(B).

Rule 2 (β ≠ ϵ) FOLLOW(B) ← FIRST(β) - ϵ		**Rule 3** ($\beta \overset{*}{\Rightarrow} \epsilon$) FOLLOW(A) → FOLLOW(B)
$Z \to X\overset{\beta}{\boxed{Y\,Z}}$ A→ α B β	β = FIRST(Y) - ϵ + FIRST(Z) - ϵ	Rule 3 is not applicable
$Z \to X\,Y\overset{\beta}{\boxed{Z}}$ A→ α B β	β = FIRST(Z) - ϵ	Rule 3 is not applicable
$Z \to \boxed{X\,Y}Z$ A→ α B β	rule 2 not applicable	$Z \to \boxed{X\,Y}Z$ A→ α B β
X → Y A→ αB β	rule 2 not applicable	X → Y A→ α B β

Note: The productions Z → d, Y → ϵ | c and X → a are not considered while computing FOLLOW since there are no non-terminals in those productions. So, the FIRST and FOLLOW sets for the left-factored grammar are shown below:

	Z	X	Y
FIRST	a,c,d	a,c, ϵ	c , ϵ
FOLLOW	$	a,c,d	a,c,d

b) Now, for the grammar to be LL(1) the following two conditions must be satisfied:

a. The first condition to be satisfied:

Production $A \to \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \ldots\ldots.$	Condition to be satisfied $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \phi$
Z → d | XYZ	FIRST (d) ∩ FIRST(XYZ) {d} ∩ {a,c,d} = d

Condition 2 is not satisfied. Hence, the grammar is not LL(1).

c) **Construction of Parsing table:** It can be constructed as shown below:

Productions **A → α**	**a = FIRST(α)**	**M[A, a] = A → α**	**Rule**
Z → d A α	d	M [Z, d] = Z → d	1
Z → XYZ A α	a, c, d	M [Z, a] = Z → XYZ M [Z, c] = Z → XYZ M [Z, d] = Z → XYZ	1
Y → c A α	c	M [Y, c] = Y → c	1
Y → ∈ A α	∈	M [Y, a] = Y → ∈ M [Y, c] = Y → ∈ M [Y, d] = Y → ∈ ↑ FOLLOW (X)	2
X → a A α	a	M [X, a] = X → a	1
X → Y A α	c, ∈	M [X, c] = X → Y M [X, a] = X → Y M [X, c] = X → Y M [X, d] = X → Y ↑ FOLLOW (X)	1 2

The parsing table is shown below:

	a	c	d	$
Z	Z→ XYZ	Z→ XYZ	Z → d Z→ XYZ	
X	X → a X → Y	X → Y	X → Y	
Y	Y → ∈	Y → c Y → ∈	Y → ∈	

Since there are multiple entries in the parse table, the given grammar is not LL(1).

Example 2.32 : Left factor the following grammar and obtain LL(1) parsing table

E → T + E | T
T → float | float * T | (E)

Solution: Since the right hand side of E-production and T-production has common prefixes, this grammar is not suitable for parsing. So, we have to do left factoring and see that two or more productions do not have common prefix. Left-factoring can be done as shown below:

The left factoring can be done to the given grammar as shown below:

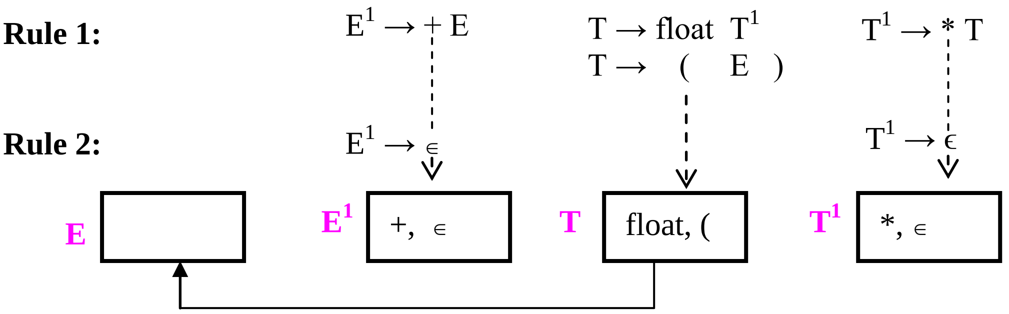
Given productions	Left-factored productions
$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots\ldots\ \alpha\beta_n \mid \gamma$	$A \to \alpha\ A^1 \mid \gamma$ $A^1 \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots\ \beta_n$
1) $E \to T\boxed{+\ E} \mid T$ $A \to \alpha\ \beta_1 \mid \alpha\ \beta_2$	$E \to T\ E^1$ $E^1 \to +\ E \mid \epsilon$
2) $T \to \text{float} \mid \text{float}\ *\boxed{T} \mid (E)$ $A \to \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid \gamma$	$T \to \text{float}\ T^1 \mid (\ E\ )$ $T^1 \to \epsilon \mid *T$

So, the grammar obtained after doing left factoring is shown below:

$$\boxed{\begin{array}{l} E \to T\ E^1 \\ E^1 \to +\ E \mid \epsilon \\ T \to \text{float}\ T^1 \mid (\ E\ ) \\ T^1 \to \epsilon \mid *T \end{array}}$$

a) The FIRST and FOLLOW sets can be computed as shown below:
FIRST sets: are computed as shown below:

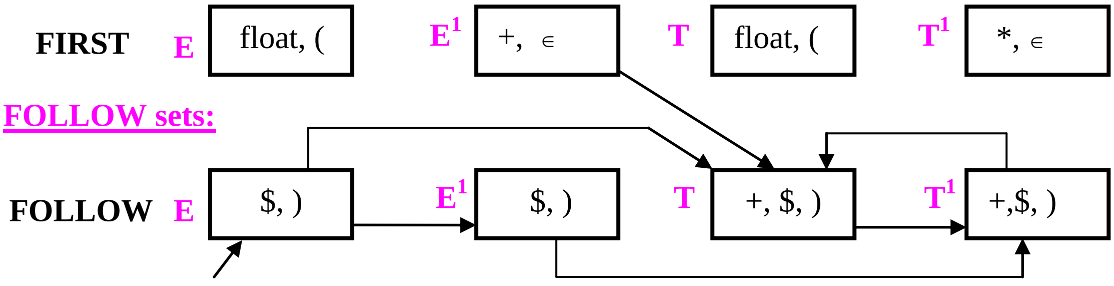
Step 1: The first symbols can be computed as shown below:



Rule 3: Consider the productions not considered earlier and obtain FIRST sets as shown below:

$E \to T\ E^1$ Add "FIRST(T) - ϵ" to FIRST(E)

In the above figure, transfer FIRST(T) to FIRST(E). So, the final FIRST sets are shown below:



Rule 1: Place $ into FOLLOW(S) since S is the start symbol.

Rule 2 &3 : Apply rule 2 and 3 for every production of the form A → αBβ where B is a non-terminal. In the first column shown below, copy from FIRST(β) to FOLLOW(B) and in the second column copy from FOLLOW(A) to FOLLOW(B).

Rule 2 (β ≠ ϵ) FOLLOW(B) ← FIRST(β) - ϵ	**Rule 3** (β ≛ ϵ) FOLLOW(A) → FOLLOW(B)
E → T E^1 A→ α B β	E → T E^1 A→ α B β
E → T E^1 A→ α B β rule 2 not applicable	E → T E^1 A→ α B β
E → + E^1 A→ α B β rule 2 not applicable	E → + E^1 A→ α B β
T → float T^1 A→ α B β rule 2 not applicable	T → float T^1 A→ α B β
T → (E) A → αB β	T → (E) A → αB β Rule 3 not applicable
T^1 → * T A → α B β rule 2 not applicable	T^1 → * T A→ α B β

Note: The productions $T^1 \to \in$ and $E^1 \to \in$ are not considered while computing FOLLOW since there are no non-terminals in those productions. So, the FIRST and FOLLOW sets for the left-factored grammar are shown below:

	E	E^1	T	T^1
FIRST	float, (	+, ∈	float, (	*, ∈
FOLLOW	$,)	$,)	+,$,)	+,$,)

b) **Construction of Parsing table:** For every production of the form A → α, we compute FIRST(α) and entries of the parsing table can be done as shown below:

Productions A → α	**a = FIRST(α)**	**M[A, a] = A → α**	**Rule**
$E \to T E^1$ A α	float, (	M [E, float] = $E \to T E^1$ M [E, '('] = $E \to T E^1$	1
$E^1 \to + E$ A α	+	M [E^1, +] = $E^1 \to + E$	1
$E^1 \to \in$ A α	∈	M [E^1, $] = $E^1 \to \in$ M [E^1, ')'] = $E^1 \to \in$	2
$T \to$ float T^1 A α	float	M [T, float] = $T \to$ float T^1	1
$T \to (E)$ A α	(	M [T, ( ] = $T \to (E)$	1
$T^1 \to \in$ A α	∈	M [T^1, $] = $T^1 \to \in$ M [T^1, ')'] = $T^1 \to \in$ M [T^1, +] = $T^1 \to \in$	2
$T^1 \to * T$ A α	*	M [T^1, *] = $T^1 \to * T$	1

The parsing table is shown below:

	float	*	+	(	)	$
E	$E \to T E^1$			$E \to T E^1$		
E^1			$E^1 \to + E$		$E^1 \to \in$	$E^1 \to \in$
T	$T \to$ float T^1			$T \to (E)$		
T^1		$T^1 \to * T$	$T^1 \to \in$		$T^1 \to \in$	$T^1 \to \in$

2.11 Error recovery in predictive parsing

Now, let us see "How error recovery is done in predictive parsing?" An error is detected during predictive parsing when the following two situations occur:

- The terminal on top of the stack does not match with the next input symbol
- When non-terminal A is on top of the stack, *a* is the next input symbol and M[A, a] has blank entry (blank denote an error)

The error recovery is done using *panic mode* and *phrase-level recovery* as shown below:

- **Panic mode:** In this approach, error recovery is done by skipping symbols from the input until a token matches with synchronizing tokens. The synchronizing tokens are selected such that the parser should quickly recover from the errors that are likely to occur in practice. Some of the recovery techniques are shown below:
  1) For a non-terminal A, consider the symbols in FOLLOW(A). These symbols can be considered as synchronizing tokens and are added into parsing table replacing only blank entries. Now, whenever there is a mismatch, keep skipping the tokens till we get one of the synchronizing character and remove A from the stack. It is likely that parsing can continue.
  2) For a non-terminal A, consider the symbols in FIRST(A). These symbols can also be considered as synchronizing characters and add to the parsing table replacing only blank entries. Now, whenever there is a mismatch, keep skipping the tokens till we get one of the synchronizing character and remove A from the stack. It is also likely that parsing can continue.
  3) If a terminal on top of the stack cannot be matched, pop the terminal from the stack and issue "Error message" and insert the corresponding terminal and continue parsing.

  For example, consider the parsing table containing synchronizing tokens and sequence of moves made by the parser in example 2.33 given later in this section.

Phrase level recovery: This recovery method is implemented by filling the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, replace or delete symbols from the input and issue appropriate error messages. They may also pop from the stack.

Example 2.33: Consider the following grammar

$$\begin{aligned} E &\rightarrow TE^1 \\ E^1 &\rightarrow +TE^1 \mid \in \\ T &\rightarrow FT^1 \\ T^1 &\rightarrow *FT^1 \mid \in \\ F &\rightarrow (E) \mid id \end{aligned}$$

and the parsing table (Refer example 2.27 for details)

	id	+	*	(	)	$
E	$E \rightarrow TE^1$			$E\rightarrow TE^1$		
E^1		$E^1 \rightarrow +TE^1$			$E^1 \rightarrow \in$	$E^1 \rightarrow \in$
T	$T \rightarrow FT^1$			$T \rightarrow FT^1$		
T^1		$T^1 \rightarrow\in$	$T^1 \rightarrow *FT^1$		$T^1 \rightarrow\in$	$T^1 \rightarrow\in$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

Add the synchronizing tokens for the above parsing table and show the sequence of moves made by parser for the string **") id * + id"**

Solution: The synchronizing characters are the characters present in FIRST or FOLLW sets of each non-terminal. In our example, let us add synchronizing characters by considering FOLLOW of each non-terminal replacing each blank entry in the parsing table. The FOLLOW sets of each non-terminal are shown below (Refer example 2.26 for details):

	E	E^1	T	T^1	F
FOLLOW	$,)	$,)	+, $,)	+, $,)	+, *, $,)

Now, FOLLOW(E) = { $,) }. So, M[E, $] = M[E,)] = synch only for blank entries. Similarly, FOLLOW(F) = {+, *, $,) }. So, M[F,+] = M[F,*] = M[F, $] = M[F,) = synch. On similar lines we add synchronizing characters to the parsing table as shown below:

	id	+	*	(	)	$
E	$E \rightarrow TE^1$			$E\rightarrow TE^1$	synch	synch
E^1		$E^1 \rightarrow +TE^1$			$E^1 \rightarrow \in$	$E^1 \rightarrow \in$
T	$T \rightarrow FT^1$	synch		$T \rightarrow FT^1$	synch	synch
T^1		$T^1 \rightarrow\in$	$T^1 \rightarrow *FT^1$		$T^1 \rightarrow\in$	$T^1 \rightarrow\in$
F	$F \rightarrow$ **id**	synch	synch	$F \rightarrow (E)$	synch	synch

Now, the sequence of moves made by the parser for the string **") id * + id"** is shown below:

Stack	Input	Output	Action
$ E	) id * + id$	**error, skip**	Remove) from the input
$ E	id * + id$	$E \rightarrow TE^1$	Remove E and push TE^1 in reverse
$ E^1 T	id * + id$	$T \rightarrow FT^1$	Remove T and push FT^1 in reverse

$\$ E^1 T^1 F$	$\text{id} * + \text{id}\$$	$F \rightarrow$ **id**	Remove F and push id in reverse
$\$ E^1 T^1$ **id**	$\text{id} * + \text{id}\$$	Match id	Pop id and increment i/p pointer
$\$ E^1 T^1$	$* + \text{id}\$$	$T^1 \rightarrow *FT^1$	Remove T^1 and push $*FT^1$ in reverse
$\$ E^1 T^1 F *$	$* + \text{id}\$$	Match *	Pop * and increment i/p pointer
$\$ E^1 T^1 F$	$+ \text{id}\$$	**error, skip**	Pop + from the input
$\$ E^1 T^1 F$	$\text{id}\$$	$F \rightarrow$ **id**	Remove F and push id in reverse
$\$ E^1 T^1$ **id**	$\text{id}\$$	Match id	Pop id and increment i/p pointer
$\$ E^1 T^1$	$\$$	Match id	Pop id and increment i/p pointer
$\$ E^1 T^1$	$\$$	$T^1 \rightarrow \in$	Remove T^1 from the stack
$\$ E^1$	$\$$	$E^1 \rightarrow \in$	Remove E^1 from the stack
$\$$	$\$$	**ACCEPT**	

Note: Observe that parsing is successful and the parser has also recognized two errors. By looking at these errors if the programmer corrects the program, parsing action is successful without any errors.

Computing FIRST sets: The first sets of LHS of the production is nothing but the terminals obtained from the first symbols on the RHS of the production.

So, $\boxed{\text{FIRST(E) = FIRST(T) = FIRST(F) = (, id}}$

Computing FOLLOW sets: The FOLLOW sets of any non-terminal A on RHS of the production are obtained the following rules:

1) Sets of terminals immediately following A or sets of first symbols obtained from the non-terminals immediately following A
2) If A is on LHS of the production and B is right most symbol on RHS of the production then FOLLOW(B) = FOLLOW(A)

Exercises

1) What is a context free grammar? What is derivation? What are the two types of derivations?
2) Define the terms: leftmost derivation, rightmost derivation, sentence

3) What the different sentential forms? What is left sentential form? What is right sentential form?

4) Define the terms: Language, derivation tree, yield of a tree, ambiguous grammar

5) Show that the following grammar is ambiguous

$$
\begin{aligned}
E &\rightarrow E + E\\
E &\rightarrow E - E\\
E &\rightarrow E * E\\
E &\rightarrow E / E\\
E &\rightarrow (E) \mid I\\
I &\rightarrow \textbf{id}
\end{aligned}
$$

6) Is the following grammar ambiguous? (if-statement or if-then-else)

$$
\begin{aligned}
S &\rightarrow iCtS \mid iCtSeS \mid a\\
C &\rightarrow b
\end{aligned}
$$

7) What is dangling else problem? How dangling else problem can be solved

8) Eliminate ambiguity from the following ambiguous grammar:

$$
\begin{aligned}
S &\rightarrow iCtS \mid iCtSeS \mid a\\
C &\rightarrow b
\end{aligned}
$$

9) Convert the following ambiguous grammar into unambiguous grammar using normal precedence and associativity of the operators

$$
\begin{aligned}
E &\rightarrow E * E \mid E - E\\
E &\rightarrow E \wedge E \mid E / E\\
E &\rightarrow E + E\\
E &\rightarrow (E) \mid \textbf{id}
\end{aligned}
$$

10) Convert the following ambiguous grammar into unambiguous grammar

$$
\begin{aligned}
E &\rightarrow E + E\\
E &\rightarrow E - E\\
E &\rightarrow E \wedge E\\
E &\rightarrow E * E\\
E &\rightarrow E / E\\
E &\rightarrow (E) \mid id
\end{aligned}
$$

by considering * and – operators lowest priority and they are left associative, / and + operators have the highest priority and are right associative and ^ operator has precedence in between and it is left associative.

11) What is parsing? What are the different types of parsers?

12) What are error recovery strategies of the parser (or syntax analyzer)?”

13) What is top down parser? Show the top-down parsing process for the string id + id * id for the grammar

    i. E → E + E
    ii. E → E * E
    iii. E → (E)
    iv. E → id

14) What is recursive descent parser? Write the algorithm for recursive descent parser

15) Write the recursive descent parser for the following grammar

    E → T
    T → F
    F → (E) | id

16) What are the different types of recursive descent parsers? What is the need for backtracking in recursive descent parser

17) Show the steps involved in recursive descent parser with backtracking for the input string *cad* for the following grammar

    S → cAd
    A → ab | a

18) For what type of grammars recursive descent parser cannot be constructed? What is the solution?

19) What is left recursion? What problems are encountered if a recursive descent parser is constructed for a grammar having left recursion?

20) Write the procedure to eliminate left recursion

21) Eliminate left recursion from the following grammar

    E → E +T | T
    T → T * F | F
    F → (E) | id

22) Write the recursive descent parser for the following grammar:

    $E \rightarrow TE^1$
    $E^1 \rightarrow +TE^1 \mid \epsilon$
    $T \rightarrow FT^1$
    $T^1 \rightarrow *FT^1 \mid \epsilon$
    $F \rightarrow (E) \mid id$

23) Obtain top-down parse for the string id+id*id for the following grammar

    $E \rightarrow TE^1$

$E^1 \rightarrow + TE^1 \mid \in$
$T \rightarrow FT^1$
$T^1 \rightarrow *FT^1 \mid \in$
$F \rightarrow (E) \mid id$

24) Eliminate left recursion from the following grammar:

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \epsilon$

25) Write the algorithm to eliminate left recursion

26) What is left factoring? What is the need for left factoring? How to do left factoring? Write the algorithm for doing left-factoring

27) Do the left-factoring for the following grammar:

$S \rightarrow iCtS \mid iCtSeS \mid a$
$C \rightarrow b$

28) Briefly explain the problems associated with top-down parser?

29) What is a predictive parser? Explain the working of predictive parser.

30) What are the various components of predictive parser? How it works?

31) Define FIRST and FOLLOW sets and write the rules to compute FIRST and FOLLOW sets

32) Consider the following grammar:

$E \rightarrow TE^1$
$E^1 \rightarrow + TE^1 \mid \in$
$T \rightarrow FT^1$
$T^1 \rightarrow *FT^1 \mid \in$
$F \rightarrow (E) \mid id$

a) Compute FIRST and FILLOW sets for the following grammar:
b) Obtain the predictive parsing table
c) Show the sequence of moves made by the parser for the string **id+id*id**
d) Add the synchronizing tokens for the above parsing table and show the sequence of moves made by parser for the string ") id * + id"

33) What is LL (1) grammar? How to check whether a given grammar is LL(1) or not without constructing the predictive parser

34) Compute FIRST and FOLLOW symbols and predictive parsing table for the following grammar and check whether the grammar is LL(1) or not.

S → iCtS | iCtSeS | a
C → b

35) Given the following grammar:

S → a | (L)
L → L , S | S

a. Is the grammar suitable for predictive parser?
b. Do the necessary changes to make it suitable for LL(1) parser
c. Compute FIRST and FOLLOW sets for each non-terminal
d. Obtain the parsing table and check whether the resulting grammar is LL(1) or not.
e. Show the moves made by the predictive parser on the input "(a , (a , a))"

36) Given the following grammar:

E → 5 + T | 3 – T
T → V | V*V | V+V
V → a | b

a. Is the grammar suitable for predictive parser?
b. What is the use of left-factoring? Do the left factoring for the above grammar
c. Compute FIRST and FOLLOW sets for each non-terminal
d. Without constructing the parsing table, check whether the grammar is LL(1)
e. By constructing the parsing table, check whether the grammar is LL(1).

37) Given the following grammar:

Z → d | XYZ
Y → ϵ | c
X → Y | a

a. Compute FIRST and FOLLOW sets for each non-terminal
b. Without constructing the parsing table, check whether the grammar is LL(1).
c. By constructing the parsing table, check whether the grammar is LL(1).

38) Left factor the following grammar and obtain LL(1) parsing table

E → T + E | T
T → float | float * T | (E)

39) How error recovery is done in predictive parsing