# Experiment No. 06

Class: TECMPNA Date:07/04/21

Name: Rebecca Dias Rollno: 19

**AIM: TO REMOVE LEFT FACTORING FROM A GIVEN GRAMMAR**

**THEORY:**

* **Non Deterministic Grammars**
* **What is a Deterministic and Non- Deterministic Grammar?**

In deterministic algorithm, for a given particular input, the computer will always produce the same output going through the same states but in case of non-deterministic algorithm, for the same input, the compiler may produce different output in different runs. In fact non-deterministic

algorithms can’t solve the problem in polynomial time and can’t determine what the next step is. The non-deterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it.

# Need for Deterministic Grammars.

Non-deterministic algorithms are not reliable as it can’t solve the problem in polynomial time and can’t determine what is the next step. The nondeterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it. In deterministic algorithm, for a given particular input, the computer will always produce the same output going through the same states and it can solve problems in polynomial time.

# How Top-Down parsers handle Left Factoring?

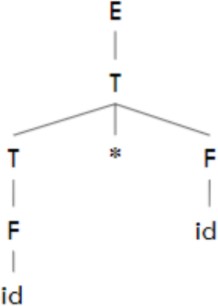
Top down parsing attempts to build the parse tree from root to leaf. Top

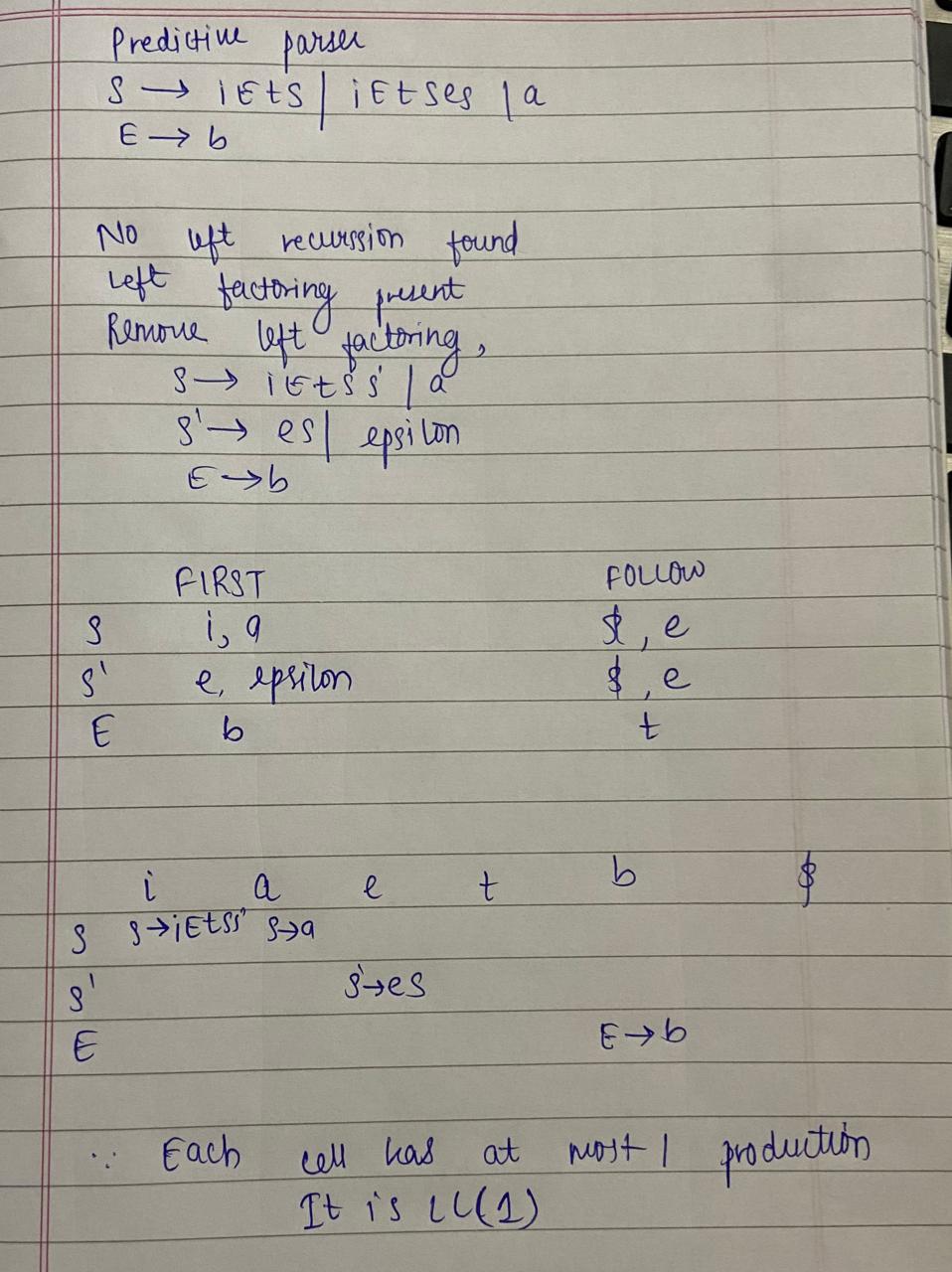
Down parser will start from start symbol and proceeds to string. It follows leftmost derivation. In leftmost derivation, the leftmost non-terminal in each sentential

is always chosen. At each step of a top-down parser, the key problem is that of determining the production to be applied for a nonterminal Once production is chosen, rest of parsing consists of matching terminal symbols in production body with the input string cannot handle left-recursive and left-factored grammars.

* **Explain with examples.** Top-Down parser for id\*id E → E + T | T

T → T \* F | F F → (E) | id





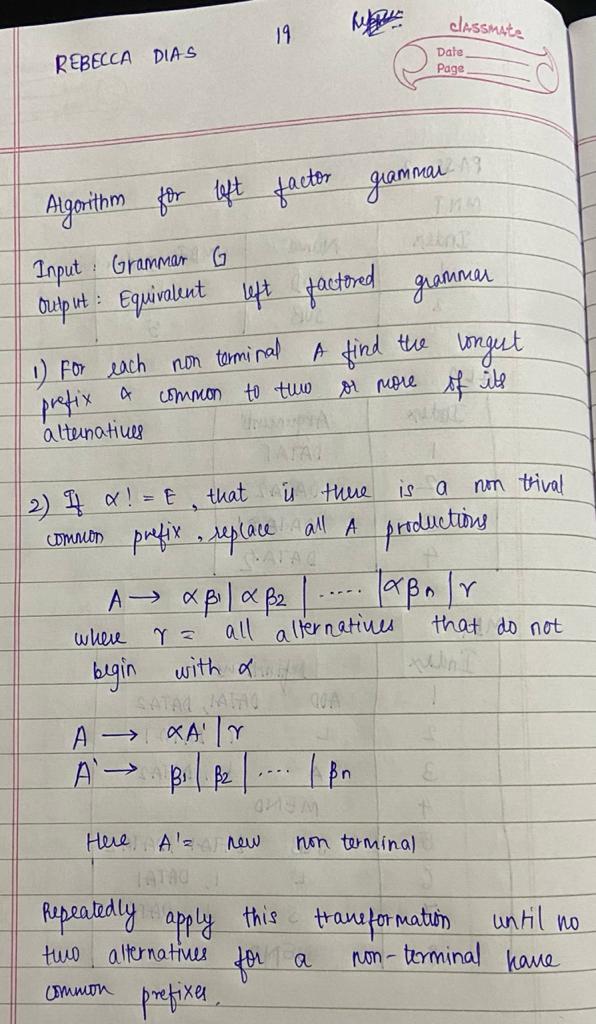
# Steps to eliminate left Factoring

In left factoring,

* 1. We make one production for each common prefixes.
  2. The common prefix may be a terminal or a non-terminal or a combination of both.
  3. Rest of the derivation is added by new productions.
  4. The grammar obtained after the process of left factoring is called as Left Factored Grammar.

# IMPLEMENTATION:

* Pseudo code (handwritten algorithm)



# CODE (C):

#include<stdio.h> #include<string.h> void main()

{

char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20]; int i,j=0,k=0,l=0,pos;

printf("Enter Production : A->"); gets(gram); for(i=0;gram[i]!='|';i++,j++)

part1[j]=gram[i]; part1[j]='\0';

for(j=++i,i=0;gram[j]!='\0';j++,i++) part2[i]=gram[j];

part2[i]='\0'; for(i=0;i<strlen(part1)||i<strlen(part2);i++)

{

if(part1[i]==part2[i])

{

modifiedGram[k]=part1[i]; k++;

pos=i+1;

}

}

for(i=pos,j=0;part1[i]!='\0';i++,j++){ newGram[j]=part1[i];

}

newGram[j++]='|'; for(i=pos;part2[i]!='\0';i++,j++){

newGram[j]=part2[i];

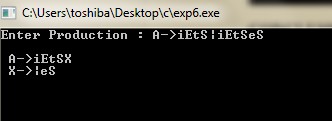
}

modifiedGram[k]='X'; modifiedGram[++k]='\0'; newGram[j]='\0';

printf("\n A->%s",modifiedGram); printf("\n X->%s\n",newGram); getch();

}

**OUTPUT:**



**CONCLUSION:**

Left factoring is a process by which the grammar containing common prefixes is transformed. This transformation makes the grammar useful for top-down parsers. We successfully Implemented Left Factoring for any grammar using C language.