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**

o **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.

o **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.

o **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.

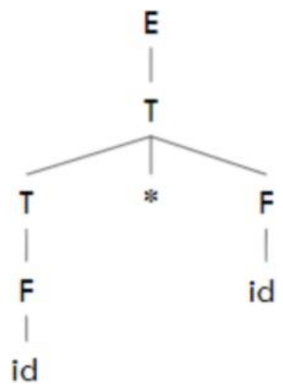
o **Explain with examples.**

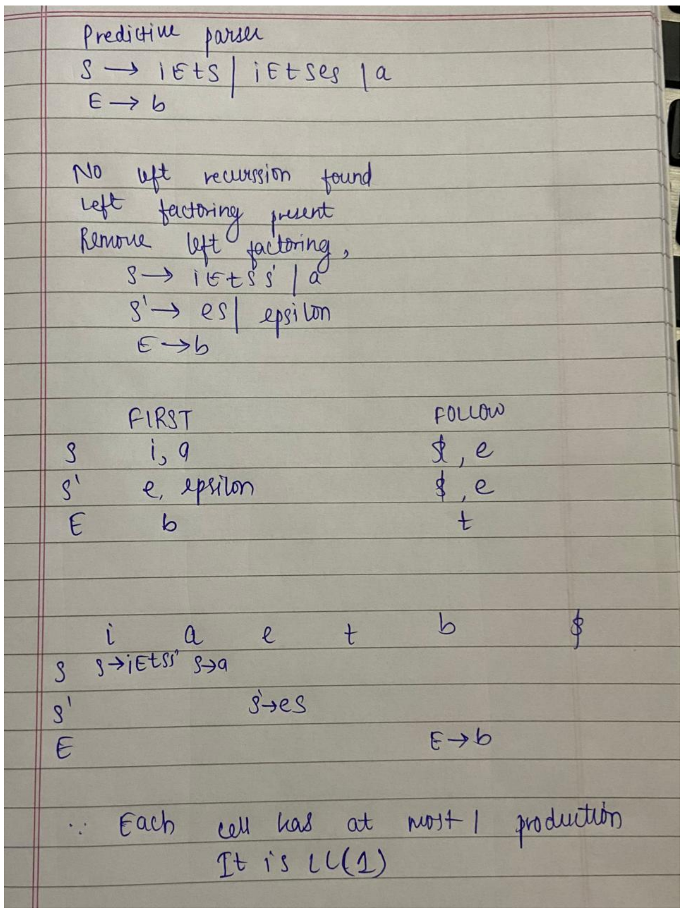
Top-Down parser for id*id

E → E + T | T

T → T * F | F

F → (E) | id





Predictive parser

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

$E \rightarrow b$

No left recursion found

Left factoring present

Remove left factoring,

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid$ epsilon

$E \rightarrow b$

	FIRST	FOLLOW
S	i, a	$, e
S'	e, epsilon	$, e
E	b	t

	i	a	e	t	b	$
S	S→iEtSS'	S→a				
S'			S'→eS			
E					E→b	

∵ Each cell has at most 1 production

It is LL(1)

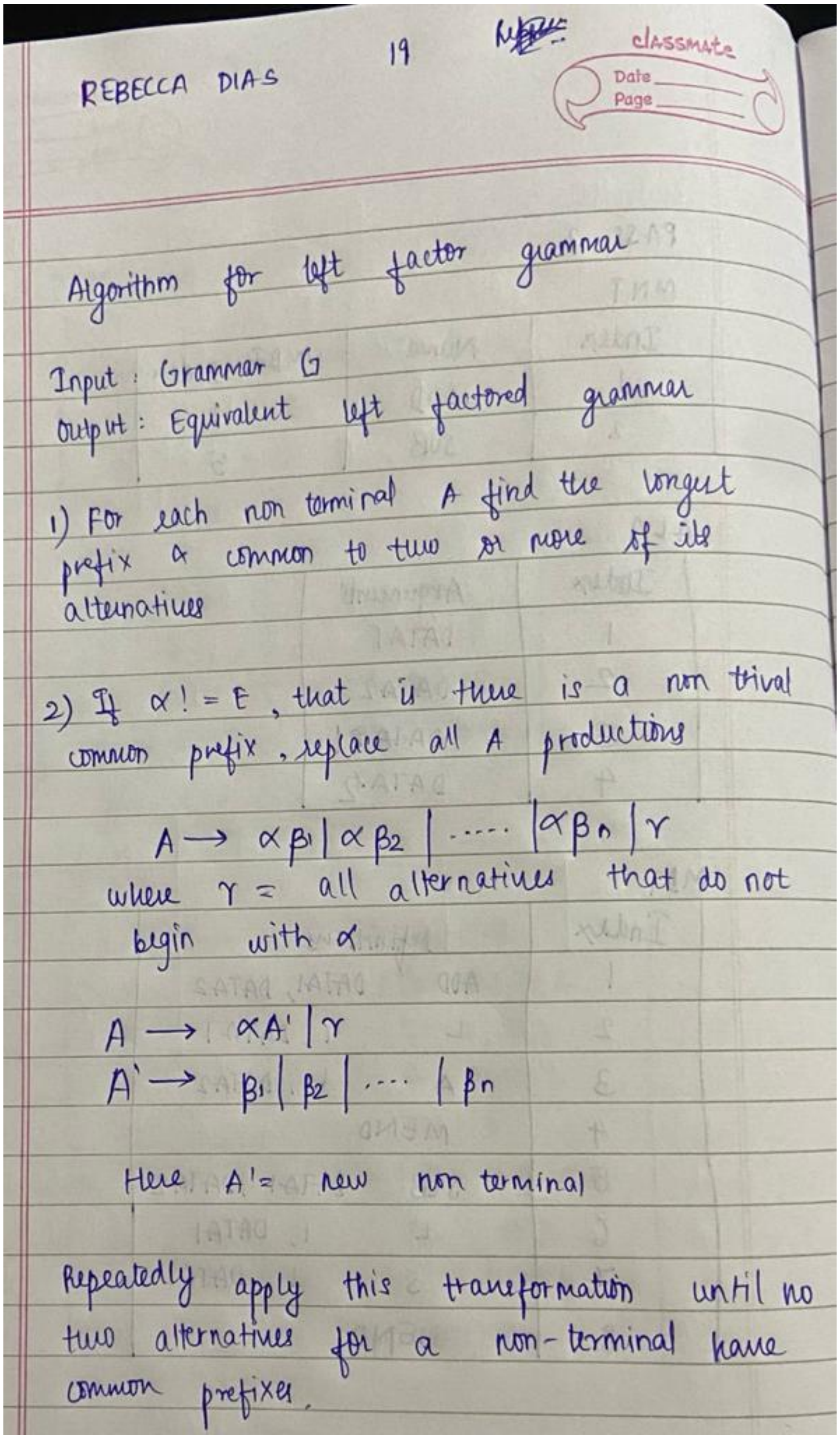
Steps to eliminate left Factoring

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)



REBECCA DIAS   19

Algorithm for left factor grammar

Input : Grammar G
Output : Equivalent left factored grammar

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

2) If $\alpha != \epsilon$, that is there is a non trival common prefix , replace all A productions

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \cdots | \alpha\beta_n | \gamma$$

where γ = all alternatives that do not begin with α

$$A \rightarrow \alpha A' | \gamma$$
$$A' \rightarrow \beta_1 | \beta_2 | \cdots | \beta_n$$

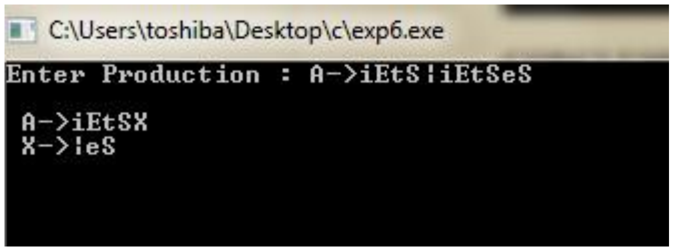
Here A' = new non terminal

Repeatedly apply this transformation until no two alternatives for a non-terminal have common prefixes.

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]='\O';
for(j=++i,i=0;gram[j]!='\O';j++,i++)
     part2[i]=gram[j];
part2[i]='\O';
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]!='\O';i++,j++){
     newGram[j]=part1[i];
}
newGram[j++]='|';
for(i=pos;part2[i]!='\O';i++,j++){
     newGram[j]=part2[i];
}
modifiedGram[k]='X';
modifiedGram[++k]='\O';
newGram[j]='\O';
printf("\n A->%s",modifiedGram);
printf("\n X->%s\n",newGram);
getch();
}
```

OUTPUT:


C:\Users\toshiba\Desktop\c\exp6.exe

```
Enter Production : A->iEtS|iEtSeS

 A->iEtSX
 X->|eS
```

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.