**Practical No. 4**

**Aim:** Consider the following grammar for arithmetic expressions. There is a single non terminal *expr* (which is the start symbol), and the terminals are **+, -, \*, /, ( , )** together with the symbol n, which we use to stand for the lexical class of nonnegative *integer literals* 0,1,2,…..

The productions are:

*expr ->n| expr*+*expr | expr*-*expr | expr*\**expr | expr*/*expr |* -*expr |* ( *expr* )

Which of the following problems does the grammar suffer from: ambiguity, left recursion, common prefixes?

Write a program to check whether the grammar is left recursive or not if it is then remove the left recursion

**Theory:**

A grammar consists of one or more variables that represent classes of strings (*i.e.,* languages). There are rules that say how the strings in each class are constructed. The construction can use:

1. Symbols of the alphabet
2. Strings that are already known to be in one of the classes or both

A Context Free Grammar (CFG) has four components or tuples G = (V, T, P, S) where

1. V is a finite set of variables (or non-terminals or syntactic categories). Each variable represents a language, *i.e.,* a set of strings.
2. T is a finite set of terminals, *i.e.,* the symbols that form the strings of the language being defined.
3. P is a set of production rules that represent the recursive definition of the language.
4. S is the start symbol that represents the language being defined.

Other variables represent auxiliary classes of strings that are used to help define the language of the start symbol. In CFG the production rule should be in the form of A → X (*i.e.,* ***head***→ ***body***) where A Є V & X Є (V U T)\* and each production rule consists of:

1. A variable that is being (partially) defined by the production. This **variable** is often called the **head** of the production.
2. The production symbol is “**→**”.
3. A string of zero or more **terminals** and variables. This string, called the **body** of the production, represents one way to form strings in the language of the variable of the head.

In doing so, we leave terminals unchanged and substitute for each variable of the body any string that is known to be in the language of the variable. We often refers to the production whose head is A as “productions for A” or “A-productions”

Moreover, the productions

A! α1, A! α2…A! αn

can be replaced by the notation

A! α1 | α2 | … | αn

**Example of CFG**

The grammar G= ({S}, {a, b, ε}, P, S) where set of variable is {S}, set of terminal is{a, b, ε}, set of productions:

*S* → *aSa*,

*S* → *bSb*,

*S* → *ε*

S is start symbol

This grammar is context-free.

**CFG for expressions in a typical programming language**

1. Operators: + (addition) and \*(multiplication)
2. Identifiers: must begin with a or b, which may be followed by any string in {a, b, 1, 0}\*

We need two variables:

E: represents expressions. It is the start symbol.

I: represents the identifiers. Its language is regular and is the language of the regular expression: **(a + b) (a + b + 0 + 1)\***

**Grammar** G1 = ({E,I} ,{+, \*, (, ), a, b, 1, 0,}, P, E) where P is the set of productions:

1. E→ I
2. E→ E + E
3. E→ E – E
4. E→ (E)
5. I → a
6. I → b
7. I → Ia
8. I → Ib
9. I → I0
10. I → I1

**Derivations Using a Grammar**

We apply the productions of a CFG to infer that certain strings are in the language of a certain variable.

Two inference approaches:

1. Recursive inference, using productions from body to head.
2. Derivations, using productions from head to body.

**Derivations**

Derivation refers to replacement of an instance of a non-terminal in a given string by the right hand side of the production rule, whose left hand side is the non-terminal to be replaced.

Applying productions from head to body requires the definition of a new relational symbol:

G = (V, T, P, S) be a CFG

A ∈ V

α, β ⊂ (V ∪ T)∗ and A → γ ∈ P

Then we write

αAβ ⇒G αγβ

or, if G is understood

αAβ ⇒ αγβ and say that αAβ derives αγβ.

**Zero or more derivation steps**

We define ∗⇒ to be the reflexive and transitive closure of ⇒ (i.e., to denote zero or more derivation steps):

**Basis**: Let α ∈ (V ∪ T) ∗. Then α ∗⇒ α.

**Induction**: If α ∗⇒ β and β ⇒ γ, then α ∗⇒ γ.

**Examples of derivation**

Derivation of a ∗ (a + b000) by G 1

E ⇒ E ∗ E ⇒ I ∗ E ⇒ a ∗ E ⇒ a ∗ (E) ⇒

a ∗ (E + E) ⇒ a ∗ (I + E) ⇒ a ∗ (a + E) ⇒ a ∗ (a + I) ⇒

a ∗ (a + I0) ⇒ a ∗ (a + I00) ⇒ a ∗ (a + b00)

**Note 1: At each step we might have several rules to choose from, e.g. I** ∗ **E** ⇒ **a** ∗ **E** ⇒ **a** ∗ **(E), versus I** ∗ **E** ⇒ **I** ∗ **(E)** ⇒ **a** ∗ **(E).**

**Note 2: Not all choices lead to successful derivations of a particular string, for instance E** ⇒ **E + E (at the first step) won’t lead to a derivation of a** ∗ **(a + b000).**

**Important: Recursive inference and derivation are equivalent. A string of terminals w is inferred to be in the language of some variable A iff A** ∗⇒ **w**

**Leftmost and Rightmost derivation**

In other to restrict the number of choices we have in deriving a string, it is often useful to require that at each step we replace the leftmost (or rightmost) variable by one of its production rules

Leftmost derivation ⇒lm : Always replace the left-most variable by one of its rule-bodies Rightmost derivation ⇒rm : Always replace the rightmost variable by one of its rule-bodies.

Examples

Consider the following grammar and apply LMD and RMD

S →aAB

A →bBb

B →A/ε

Using Leftmost derivation-

S →a***A***B

S →ab***B***bB

S →abb***B***

S →abb

Using Rightmost derivation-

S →aA***B***

S →a***A***

S →ab***B***b

S →abb

**The Language of the Grammar**

If G(V, T, P, S) is a CFG, then the language of G is **L(G) = {w in T** ∗ **| S** ∗ ⇒ **G w }** i.e., the set of strings over T derivable from the start symbol.

If G is a CFG, we call L(G) a context-free language. Example: L(Gpal) is a context-free language.

**Example**

Recall G1:

E → I | E + E | E ∗ E | (E)

I → a | b | Ia | Ib | I0 | I1

1− Then E ∗ (I + E) is a sentential form since

E ⇒ E ∗ E ⇒ E ∗ (E) ⇒ E ∗ (E + E) ⇒ E(I + E) This derivation is neither leftmost, nor right-most.

2− a ∗ E left-sentential form, since

E ⇒ E ∗ E ⇒ I ∗ E ⇒ a ∗ E

3− E ∗ (E + E) is a right-sentential form since

E ⇒ E ∗ E ⇒ E ∗ (E) ⇒ E ∗ (E + E)

**Ambiguity**

For some language *w*, there may exist more than one parse trees that means there may exist more than one way of deriving, *w* from *S*, using the productions of the grammar. Or A CFG is said to be ambiguous if there exists a string which has more than one leftmost derivation.

**Left recursion**

The grammar for assignment statements is left-recursive; that is, in the process of expanding the first nonterminal, that nonterminal is generated again:

**Expression → Expression + Term**

**Common prefix**

S**→** a α

S**→** a β

Consider the above two productions, those are having ‘a’ as common prefix

**Program:**

#include<stdio.h>

#include<string.h>

#include<conio.h>

#define SIZE 10

int main ()

{

char non\_terminal;

char beta,alpha;

int num,i;

char production[10][SIZE];

int index=3;

/\* starting of the string following "->" \*/

clrscr();

printf("Enter Number of Production : ");

scanf("%d",&num);

printf("Enter the grammar as E->E-A :\n");

for(i=0;i<num;i++)

scanf("%s",production[i]);

for(i=0;i<num;i++){

printf("\nGRAMMAR : : : %s",production[i]);

non\_terminal=production[i][0];

if(non\_terminal==production[i][index])

{

alpha=production[i][index+1];

printf(" is left recursive.\n");

while(production[i][index]!=0 && production[i][index]!='|')

index++;

if(production[i][index]!=0)

{

beta=production[i][index+1];

printf("Grammar without left recursion:\n");

printf("%c->%c%c\'",non\_terminal,beta,non\_terminal);

printf("\n%c\'->%c%c\'|E\n",non\_terminal,alpha,non\_terminal);

}

else

printf(" can't be reduced\n");

}

else

printf(" is not left recursive.\n");

index=3;

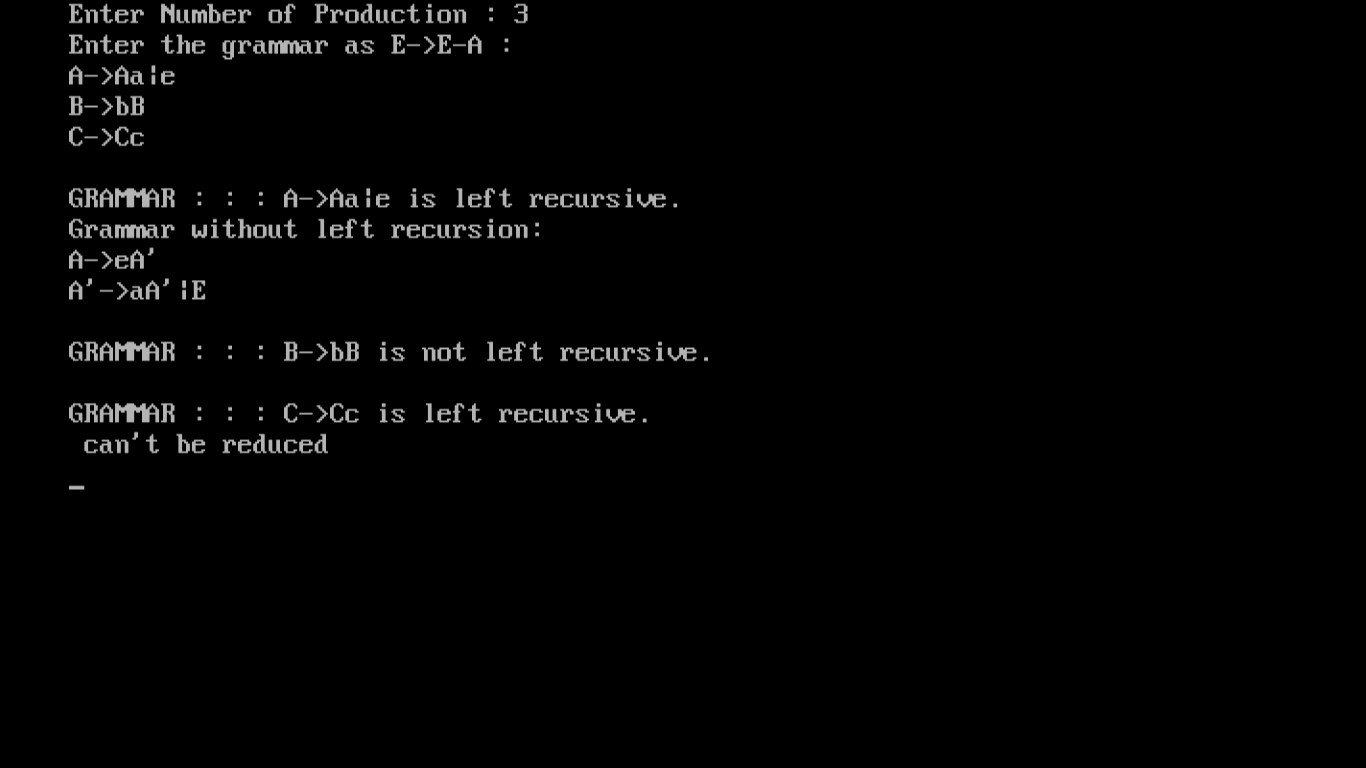
}

getch();

return 0;

}

**Output:**



**Conclusion:** A simple program to remove left recursion from a Context free grammar has been made and left recursion has been removed from the given CFG.