EXPERIMENT-5

Program:

Write a program to remove left recursion in CFG.

Theory:

Left recursion occurs in a grammar when a non-terminal A directly produces a string that starts with A itself. Left-recursive grammars can be problematic for recursive descent parsers, as they can lead to infinite recursion. To remove left recursion from a grammar, you can use the following algorithm:

Given a grammar with a set of non-terminals N, a set of terminals T, a set of production rules P, and a start symbol S:

1. Identify any left-recursive productions. A production A -> Aα is left-recursive if α can derive the empty string or starts with A.
2. For each left-recursive non-terminal A, create a new non-terminal A' and replace the left-recursive productions with:
    - A -> βA'
    - A' -> αA' | ε

Where β is a sequence of terminals and non-terminals that does not start with A, and ε is the empty string.

3. Replace any occurrences of A in other productions with A'.
4. Update the start symbol if necessary.

Input:

```
#include <stdio.h>

#include <string.h>

#define SIZE 10

int main() {

   char non_terminal;

   char beta, alpha;

   int num;

   char production[10][SIZE];

   int index = 3; /* starting of the string following "->" */

   printf("Enter Number of Production: ");

   scanf("%d", &num);

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

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

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

```
    }
    for (int 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'\n", non_terminal, beta, non_terminal);
                printf("%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;    }
    return 0;
}
```

Errors:

Compilation failed due to following error(s).

```
main.c: In function ‘main’:
main.c:22:9: error: ‘nonterminal’ undeclared (first use in this function); did you mean ‘non_terminal’?
   22 |         nonterminal = production[i][0];
      |         ^~~~~~~~~~~
      |         non_terminal
main.c:22:9: note: each undeclared identifier is reported only once for each function it appears in
```

```
main.c:28:68: error: expected ‘]’ before ‘)’ token
   28 |             while (production[i][index] != 0 && production[i][index) != '|') {
      |                                                                    ^
      |                                                                    ]
main.c:28:70: error: expected expression before ‘!=’ token
   28 |             while (production[i][index] != 0 && production[i][index) != '|') {
      |                                                                     ^~
main.c:28:76: error: expected statement before ‘)’ token
   28 |             while (production[i][index] != 0 && production[i][index) != '|') {
      |                                                                            ^
```

Output:

```
Enter Number of Production: 4
Enter the grammar as E->E-A:
E->EA|A
A->AT|a
T=a
E->i

GRAMMAR : : : E->EA|A is left recursive.
Grammar without left recursion:
E->AE'
E'->AE'|E

GRAMMAR : : : A->AT|a is left recursive.
Grammar without left recursion:
A->aA'
A'->TA'|E

GRAMMAR : : : T=a is not left recursive.

GRAMMAR : : : E->i is not left recursive.
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science and Engineering Amity University, Noida (U.P)			
Programme	B.Tech CSE	Course Name	Complier Construction
Course Code	CSE304	Semester	6
Student Name		Enrollment No.	
Marking Criteria			
Criteria	Total Marks	Marks Obtained	Comments
Concept	2		
Implementation	2		
Performance	2		
Total	6		

EXPERIMENT-6

Program:

Write a program to remove left factoring in the grammar.

Theory:

Left factoring is a technique used in formal language theory and compiler design to transform a grammar in order to remove ambiguity and make parsing easier.

When a grammar has multiple production rules for a non-terminal that all start with the same sequence of symbols (a common prefix), it can lead to ambiguity in parsing. Left factoring is a method to eliminate this ambiguity by factoring out the common prefix into a new production rule.

The general idea of left factoring is to rewrite the grammar rules so that the common prefix is factored out into a new non-terminal symbol. For example, given the grammar:

A -> αβ1 | αβ2

where **α** is a common prefix and **β1** and **β2** are different suffixes, the left-factored grammar would be:

A -> αA' A' -> β1 | β2

Here, **A'** is a new non-terminal symbol representing the common suffixes of the original rules.

Left factoring is often used as a preprocessing step in compiler design, particularly in parser generators, to simplify the grammar and make it more suitable for parsing algorithms.

Input:

```
#include <stdio.h>
#include <string.h>
int main()
{
   char gram[50], part1[50], part2[50], modifiedGram[50], newGram[50];
   int i, j = 0, k = 0, pos = 0;
   printf("Enter Production : A->");
   fgets(gram, sizeof(gram), stdin);
   gram[strcspn(gram, "\n")] = '\0';
   for (i = 0; gram[i] != '|'; i++)
      part1[j++] = gram[i];
   part1[j] = '\0';
   for (j = 0, i = i + 1; gram[i] != '\0'; i++)
      part2[j++] = gram[i];
   part2[j] = '\0';
   for (i = 0; i < strlen(part1) && i < strlen(part2); i++)
```

```
    {
        if (part1[i] == part2[i])
        {
            modifiedGram[k++] = part1[i];
            pos = i + 1;
        }
        else
        {
            break;
        }
    }
    modifiedGram[k] = '\0';
    strcpy(newGram, "X");
    strcat(newGram, &part1[pos]);
    strcat(newGram, "|");
    strcat(newGram, &part2[pos]);
    printf("\nA->%s", modifiedGram);
    printf("\nX->%s\n", newGram);

    return 0;
}
```

Errors:

Compilation failed due to following error(s).

```
main.c: In function ‘main’:
main.c:10:5: warning: ‘gets’ is deprecated [-Wdeprecated-declarations]
   10 |     gets(gram, sizeof(gram), stdin);
      |     ^~~~
In file included from main.c:1:
/usr/include/stdio.h:605:14: note: declared here
  605 | extern char *gets (char *__s) __wur __attribute_deprecated__;
      |             ^~~~
```

```
main.c:10:5: error: too many arguments to function ‘gets’
   10 |     gets(gram, sizeof(gram), stdin);
      |     ^~~~
In file included from main.c:1:
/usr/include/stdio.h:605:14: note: declared here
  605 | extern char *gets (char *__s) __wur __attribute_deprecated__;
      |             ^~~~
```

```
main.c:31:42: warning: implicit declaration of function ‘len’ [-Wimplicit-function-declaration]
   31 |     for (i = 0; i < strlen(part1) && i < len(part2); i++)
      |                                          ^~~
/usr/bin/ld: /tmp/ccnm4J7A.o: in function `main':
main.c:(.text+0x213): undefined reference to `len'
collect2: error: ld returned 1 exit status
```

Output:

```
Enter Production : A->aE|bcD|aE+eIT

A->
X->XaE|bcD|aE+eIT
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science and Engineering Amity University, Noida (U.P)			
Programme	B.Tech CSE	Course Name	Complier Construction
Course Code	CSE304	Semester	6
Student Name		Enrollment No.	
Marking Criteria			
Criteria	Total Marks	Marks Obtained	Comments
Concept	2		
Implementation	2		
Performance	2		
Total	6		

EXPERIMENT-7

Program:

Write a program to find FIRST and FOLLOW of non-terminals in a grammar.

Theory:

In the context of formal language theory and grammar, "first" and "follow" sets are used in predictive parsing and LL parsing algorithms to determine which production to use when parsing a string. These sets are used for non-terminal symbols in the grammar.

1. **First Set**: The first set of a non-terminal symbol in a grammar is the set of terminals that can appear as the first symbol of a string derived from that non-terminal. It is used to predict which production rule to apply when parsing a string.
    - For a terminal symbol **X**, **First(X)** is simply **{X}**.
    - For a non-terminal symbol **A**, **First(A)** is the union of the first sets of all symbols that **A** can derive, considering epsilon (ε) productions. This means **First(A)** includes the first sets of the symbols in the right-hand side of each production rule for **A**, as well as **ε** if **A** can derive the empty string.
2. **Follow Set**: The follow set of a non-terminal symbol in a grammar is the set of terminals that can appear immediately to the right of that non-terminal in a derivation. It is used to decide when to expand a non-terminal during parsing.
    - The start symbol has **$** (end of input) in its follow set.
    - For a non-terminal symbol **A**, **Follow(A)** is the set of terminals that can follow **A** in some sentential form. It includes the first sets of the symbols that can follow **A**, as well as **Follow(B)** for any non-terminal **B** if **A** can derive ε.

The first and follow sets are computed iteratively until there are no changes in the sets. These sets play a crucial role in LL parsing table construction and help in determining the correct production to use during parsing.

Input:

```
#include<stdio.h>

#include<math.h>

#include<string.h>

#include<ctype.h>

#include<stdlib.h>

int n,m=0,p,i=0,j=0;

char a[10][10],f[10];

void follow(char c);

void first(char c);

int main()

{ int i,z;
```

```
    char c,ch;
    printf("Enter the no of productions:\n");
    scanf("%d",&n);
    printf("Enter the productions:\n");
    for(i=0;i<n;i++)
    scanf("%s%c",a[i],&ch);
    do
    {   m=0;
        printf("Enter the elements whose first & follow is to be found:");
        scanf("%c",&c);
        first(c);
        printf("First(%c)={",c);
        for(i=0;i<m;i++)
        printf("%c",f[i]);
        printf("}\n");
    strcpy(f," ");
    m=0;
    follow(c);
    printf("Follow(%c)={",c);
    for(i=0;i<m;i++)
    printf("%c",f[i]);
    printf("}\n");
    printf("Continue(0/1)?");
    scanf("%d%c",&z,&ch);    }
while(z==1);
return(0);}
void first(char c)
{    int k;
    if(!isupper(c))
f[m++]=c;
for(k=0;k<n;k++) {
if(a[k][0]==c)
```

{ if(a[k][2]=='$')

follow(a[k][0]);

else if(islower(a[k][2]))

f[m++]=a[k][2];

else first(a[k][2]); }

} } void follow(char c)

{ if(a[0][0]==c)

f[m++]='$';

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

{ for(j=2;j<strlen(a[i]);j++)

{ if(a[i][j]==c)

{ if(a[i][j+1]!='\0')

first(a[i][j+1]);

if(a[i][j+1]=='\0' && c!=a[i][0])

follow(a[i][0]); } } } }

Errors:

Compilation failed due to following error(s).

```
main.c: In function ‘main’:
main.c:59:45: error: ‘first’ undeclared (first use in this function)
   59 |                 if (calc_first[jm][lark] == first[j]) {
      |                                             ^~~~~
main.c:59:45: note: each undeclared identifier is reported only once for each function it appears in
main.c:101:46: error: ‘f’ undeclared (first use in this function)
  101 |                 if (calc_follow[km][lark] == f[j]) {
      |                                              ^
main.c: In function ‘follow’:
main.c:118:9: error: ‘production’ undeclared (first use in this function)
  118 |     if (production[0][0] == c) {
      |         ^~~~~~~~~~
main.c:119:9: error: ‘f’ undeclared (first use in this function)
  119 |         f[m++] = '$';
      |         ^
main.c: In function ‘findfirst’:
main.c:140:9: error: ‘first’ undeclared (first use in this function)
  140 |         first[n++] = c;
      |         ^~~~~
main.c:143:13: error: ‘production’ undeclared (first use in this function)
  143 |         if (production[j][0] == c) {
      |             ^~~~~~~~~~
main.c:147:17: error: expected expression at end of input
  147 |                 else if (production[q1][q2] != '\0' && (
      |                 ^~~~
main.c:147:17: error: expected declaration or statement at end of input
main.c:147:17: error: expected declaration or statement at end of input
main.c:147:17: error: expected declaration or statement at end of input
main.c:147:17: error: expected declaration or statement at end of input
```

```
main.c:149:43: warning: missing terminating ' character
  149 |                 if (production[q1][q2] == '\0
      |                                           ^
main.c:149:43: error: missing terminating ' character
  149 |                 if (production[q1][q2] == '\0
      |                                           ^~~
main.c:149:17: error: expected expression at end of input
  149 |                 if (production[q1][q2] == '\0
      |                 ^~
main.c:149:17: error: expected declaration or statement at end of input
main.c:149:17: error: expected declaration or statement at end of input
main.c:149:17: error: expected declaration or statement at end of input
main.c:149:17: error: expected declaration or statement at end of input
```

Output:

```
Enter the no of productions:
5
Enter the productions:
S=AB
A=a
B=bc
D=cx
C=z
Enter the elements whose first & follow is to be found:S
First(S)={a}
Follow(S)={$}
Continue(0/1)?1
Enter the elements whose first & follow is to be found:A
First(A)={a}
Follow(A)={b}
Continue(0/1)?1
Enter the elements whose first & follow is to be found:B
First(B)={b}
Follow(B)={$}
Continue(0/1)?1
Enter the elements whose first & follow is to be found:D
First(D)={c}
Follow(D)={}
Continue(0/1)?1
Enter the elements whose first & follow is to be found:C
First(C)={z}
Follow(C)={}
Continue(0/1)?0

...Program finished with exit code 0
Press ENTER to exit console.
```

Internal Assessment (Mandatory Experiment) Sheet for Lab Experiment Department of Computer Science and Engineering Amity University, Noida (U.P)			
Programme	B.Tech CSE	Course Name	Complier Construction
Course Code	CSE304	Semester	6
Student Name		Enrollment No.	
Marking Criteria			
Criteria	Total Marks	Marks Obtained	Comments
Concept	2		
Implementation	2		
Performance	2		
Total	6		

OPEN ENDED EXPERIMENT

Program:

Write a program to construct a complier of any language.

Theory:

Constructing a compiler for any language involves several key steps, typically following a well-defined process. Here's a high-level overview of the theory behind compiler construction:

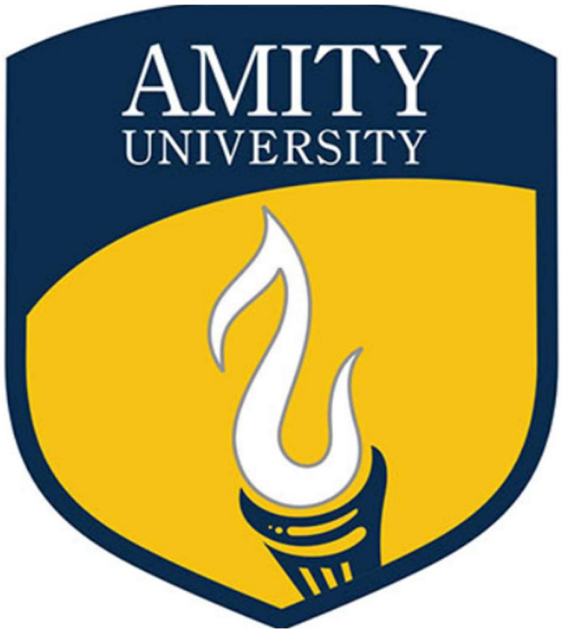
1. **Lexical Analysis (Scanning):** The first step is to break the input source code into tokens. This is done by the lexical analyzer, which reads the input characters and groups them into tokens like identifiers, keywords, literals, and operators.
2. **Syntax Analysis (Parsing):** The next step is to analyze the structure of the tokens to ensure they conform to the grammar rules of the language. This is done by the parser, which typically generates a parse tree or an abstract syntax tree (AST) representing the syntactic structure of the program.
3. **Semantic Analysis:** Once the syntax is verified, the compiler performs semantic analysis to check for semantic errors like type mismatches, undeclared variables, etc. This step often involves building a symbol table to keep track of identifiers and their properties.
4. **Intermediate Code Generation:** After the source code is analyzed and validated, the compiler generates an intermediate representation (IR) of the code. This IR is often a lower-level, platform-independent representation of the code that makes further optimization and code generation easier.
5. **Optimization:** The compiler may perform various optimizations on the IR to improve the efficiency of the generated code. This can include optimizations like constant folding, dead code elimination, and loop optimization.
6. **Code Generation:** Finally, the compiler translates the optimized IR into machine code or another target language. This involves mapping the high-level constructs of the source language to the corresponding machine instructions or code in the target language.
7. **Linking and Loading:** For languages that support modular programming, the compiler may also be responsible for linking together separate modules and resolving external references. The resulting executable or library can then be loaded and executed by the target platform.

Throughout these steps, the compiler needs to handle error reporting and recovery to gracefully handle syntax and semantic errors in the source code.

It's important to note that compiler construction is a complex and challenging task, often requiring a deep understanding of language theory, formal grammars, and programming languages. Modern compilers also incorporate advanced techniques like just-in-time compilation (JIT) and support for multiple target platforms.

AMITY SCHOOL OF ENGINEERING & TECHNOLOGY

AMITY UNIVERSITY CAMPUS, SECTOR-125, NOIDA-201303



Compiler Construction Lab

PRACTICAL FILE

COURSE CODE: CSE304

Submitted to:
Dr. Misha Kakkar

Submitted by:
Boddu Asmitha Bhavya
A2305221386
6CSE-6X

INDEX

SL.NO	Experiments	Date of assigned	Date of submission	Signature