PART A

(PART A : TO BE REFFERED BY STUDENTS)

Experiment No. 7 (A)

Aim: To implement the program to remove left recursion from grammar and find first and follow of the given grammar

Objective: Develop a program to find:

a. FIRST set
b. FOLLOW set

for given grammar.

Outcome: Students are able to implement a program to remove left recursion and find FIRST and FOLLOW set required to generate predictive parsing table.

Theory:

Left Recursion:

Left Recursion. The production is **left-recursive** if the leftmost symbol on the right side is the same as the non terminal on the **left** side. For example, expr → expr + term. If one were to code this production in a **recursive**-descent parser, the parser would go in an infinite loop.

IMMEDIATE LEFT RECURSION. A production is immediately left recursive if its left hand side and the head of its right hand side are the same symbol, e.g. B → Bvt

A grammar is called immediately left recursive if it possesses an immediately left recursive production.

INDIRECT LEFT RECURSION. A grammar is said to posess indirect left recursion if it is possible, starting from any symbol of the grammar, to derive a string whose head is that symbol.

Example. A → Br

B → Cs

C → At

Here, starting with A, we can derive Atsr

Left recursion often poses problems for parsers, either because it leads them into infinite recursion (as in the case of most top-down parsers) or because they expect rules in a normal form that forbids it (as in the case of many bottom-up parsers). Therefore, a grammar is often preprocessed to eliminate the left recursion.

Removing direct left recursion

The general algorithm to remove direct left recursion follows. Several improvements to this method have been made.For a left-recursive nonterminal A, discard any rules of the form A -> A and consider those that remain:

A -> $A\alpha_1$ | ….. |$A\alpha_n$ |β_1| ….. |$\beta_m$where:

- each α is a nonempty sequence of nonterminals and terminals, and
- each β is a sequence of nonterminals and terminals that does not start with A.

Replace these with two sets of productions, one set for A:

A -> $\beta_1 A'$ | …….. | $\beta_m A'$and another set for the fresh nonterminal A' (often called the "tail" or the "rest"):

A' -> $\alpha_1 A'$ | ….. |$\alpha_n A'$ | ε

Repeat this process until no direct left recursion remains.

FIRST AND FOLLOW :

The two functions used to make entries in parsing table of predictive parser are FIRST and FOLLOW. The tokens in FOLLOW function can be used as synchronizing tokens during panic-mode error recovery.

If α is a string of grammar symbols, let FIRST(α) be the set of terminals that begin the strings derived from α. If α□ε, then ε is also in FIRST(α).

FOLLOW(A), for non-terminal A, is defined as a set of terminals **a** that can appear immediately to the right of A in some sentential form. That is, the set of terminals a such that there exists a derivation S□αAaβ for some α and β.

If A can be the rightmost symbol in the some sentential form, then $ is in FOLLOW(A).

- **Rules to compute FIRST(X)** for X is any grammar symbol.
  1. If X is terminal, then FIRST(X)={X}.
  2. If X□ε is a production, then add ε to FIRST(X)
  3. If X is a non-terminal and X□$Y_1Y_2Y_3….Y_K$ is a production, then place **a** in FIRST(X) if for some i, **a** is isnFIRST(Y_i), and ε is in all of Y_1, Y_2,…..Y_{i-1}. That is Y_1.....Y_{i-1}□ε.
     If ε is in FIRST(Y_j) for all j = 1,2,3,….k, then add ε to FIRST(X).
     If Y_1 does not derive ε, we cannot add more symbols to FIRST(X), But if Y_1□ε then we add FIRST(Y_2) and so on.
- **Rules to compute FOLLOW(X)** for X is any grammar symbol.
  1. Place $ in FOLLOW(X), where X is a start symbol and $ is end marker of input string.
  2. If there is a production A□αBβ, then then everything in FIRST(β) except for ε is placed in FOLLOW(B).

3. If there is a production A□αB, or a production A□αBβ where FIRST(β) contains ε then everything in FOLLOW(A) is in FOLLOW(B).

- **Example**

Consider the following grammar of example 8.8

E□TE'

E'□+TE' | ε

T□FT'

T'□*FT' | ε

F□(E) | **id**

Thus, the FIRST and FOLLOW set for all the non-terminals are summarized below.

FIRST(E)={(,id}

FIRST(T)={(,id}

FIRST(F)={(,id}

FIRST(E')={+,□}

FIRST(E')={*,□}

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

FOLLOW(T) = { +,), $ }

FOLLOW(T') = { +,), $ }

FOLLOW(F) = {+, *,), $ }

PART B

(PART B : TO BE COMPLETED BY STUDENTS)

(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded at the end of the practical)

Roll. No.B48	Name: Aryan Unhale
Class:TE-B COMPS	Batch:B3
Date of Experiment:29/3/25	Date of Submission: 4/4/25
Grade:	

B.1 Software Code written by student:

(A) Program to remove left recursion from grammar:

```
#include <stdio.h>
#include <string.h>
#define SIZE 10

int main() {
   char non_terminal;
   char beta, alpha;
   int num, i;
   char production[10][SIZE];
   int index = 3;

   printf("Enter Number of Productions: ");
   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;
    }
    return 0;
}
```

(B) Program to find first and follow:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void followfirst(char, int, int);
void follow(char c);
void findfirst(char, int, int);

int count, n = 0;
char calc_first[10][100];
char calc_follow[10][100];
int m = 0;
char production[10][10];
char f[10], first[10];
int k, e;
char ck;

int main(int argc, char **argv) {
    int jm = 0, km = 0;
    int i, choice;
    char c, ch;
    count = 8;

    strcpy(production[0], "E=TR");
    strcpy(production[1], "R=+TR");
    strcpy(production[2], "R=#");
    strcpy(production[3], "T=FY");
    strcpy(production[4], "Y=*FY");
    strcpy(production[5], "Y=#");
    strcpy(production[6], "F=(E)");
    strcpy(production[7], "F=i");

    int kay;
    char done[count];
    int ptr = -1;

    for (k = 0; k < count; k++) {
        for (kay = 0; kay < 100; kay++) {
            calc_first[k][kay] = '!';
```

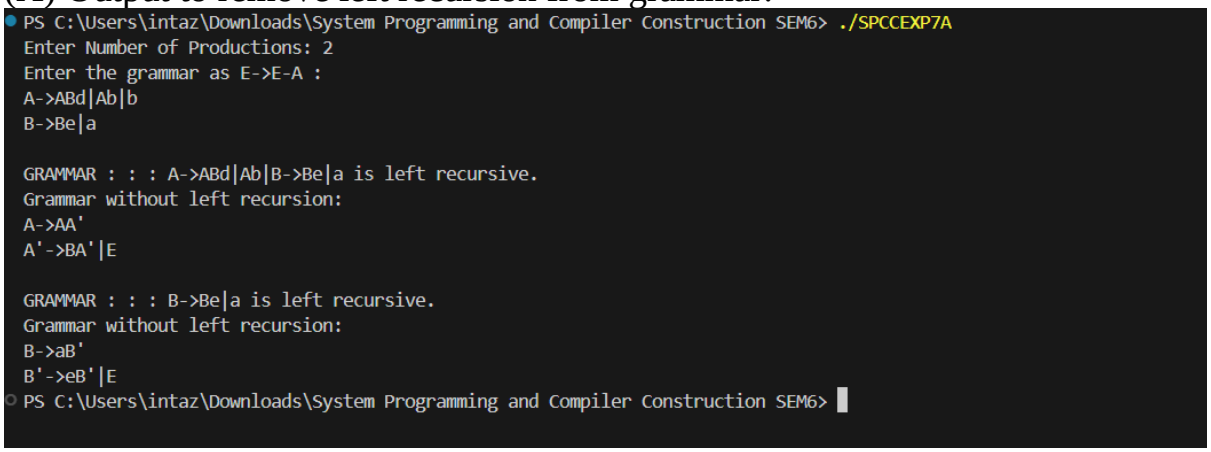
```
    }
}

int point1 = 0, point2, xxx;
for (k = 0; k < count; k++) {
    c = production[k][0];
    point2 = 0;
    xxx = 0;
    for (kay = 0; kay <= ptr; kay++)
        if (c == done[kay]) xxx = 1;
    if (xxx == 1) continue;

    findfirst(c, 0, 0);
    ptr += 1;
    done[ptr] = c;
    printf("\nFirst(%c) = { ", c);
    calc_first[point1][point2++] = c;

    for (i = 0 + jm; i < n; i++) {
        int lark = 0, chk = 0;
        for (lark = 0; lark < point2; lark++) {
            if (first[i] == calc_first[point1][lark]) {
                chk = 1;
                break;
            }
        }
        if (chk == 0) {
            printf("%c, ", first[i]);
            calc_first[point1][point2++] = first[i];
        }
    }
    printf("}\n");
    jm = n;
    point1++;
}

printf("\n.....\n\n");

char donee[count];
ptr = -1;
for (k = 0; k < count; k++) {
    for (kay = 0; kay < 100; kay++) {
        calc_follow[k][kay] = '!';
    }
}

point1 = 0;
int land = 0;

for (e = 0; e < count; e++) {
```

```
        ck = production[e][0];
        point2 = 0;
        xxx = 0;
        for (kay = 0; kay <= ptr; kay++)
            if (ck == donee[kay]) xxx = 1;
        if (xxx == 1) continue;

        land += 1;
        follow(ck);
        ptr += 1;
        donee[ptr] = ck;
        printf("Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;

        for (i = 0 + km; i < m; i++) {
            int lark = 0, chk = 0;
            for (lark = 0; lark < point2; lark++) {
                if (f[i] == calc_follow[point1][lark]) {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0) {
                printf("%c, ", f[i]);
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }
}

void follow(char c) {
    int i, j;
    if (production[0][0] == c) {
        f[m++] = '$';
    }
    for (i = 0; i < 10; i++) {
        for (j = 2; j < 10; j++) {
            if (production[i][j] == c) {
                if (production[i][j + 1] != '\0') {
                    followfirst(production[i][j + 1], i, (j + 2));
                }
                if (production[i][j + 1] == '\0' && c != production[i][0]) {
                    follow(production[i][0]);
                }
            }
        }
    }
```

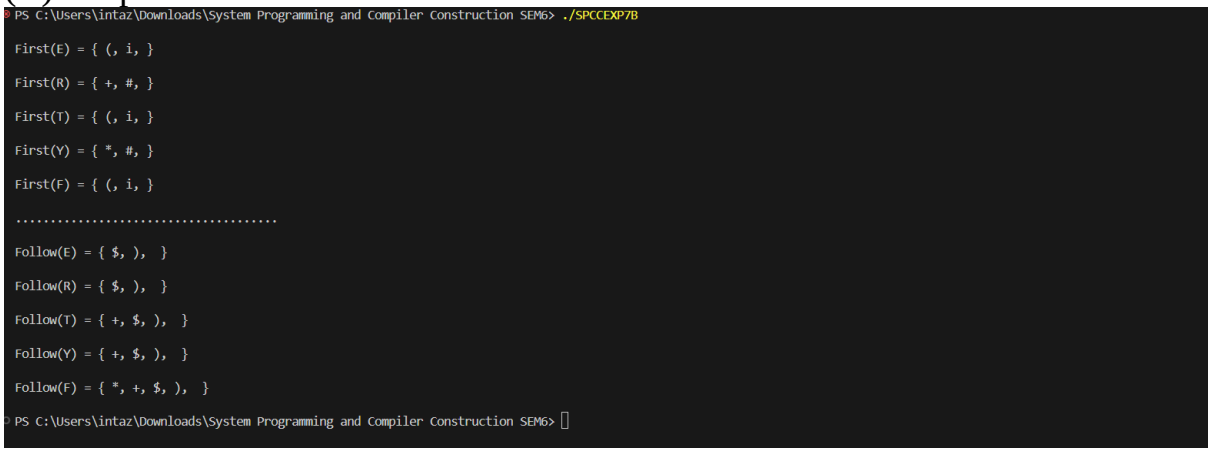
```
}

void findfirst(char c, int q1, int q2) {
   int j;
   if (!(isupper(c))) {
      first[n++] = c;
      return;
   }
   for (j = 0; j < count; j++) {
      if (production[j][0] == c) {
         if (production[j][2] == '#') {
            if (production[q1][q2] == '\0')
               first[n++] = '#';
            else if (production[q1][q2] != '\0' && (q1 != 0 || q2 != 0))
               findfirst(production[q1][q2], q1, (q2 + 1));
            else
               first[n++] = '#';
         } else if (!isupper(production[j][2])) {
            first[n++] = production[j][2];
         } else {
            findfirst(production[j][2], j, 3);
         }
      }
   }
}

void followfirst(char c, int c1, int c2) {
   int k;
   if (!(isupper(c))) {
      f[m++] = c;
   } else {
      int i = 0, j = 1;
      for (i = 0; i < count; i++) {
         if (calc_first[i][0] == c)
            break;
      }
      while (calc_first[i][j] != '!') {
         if (calc_first[i][j] != '#') {
            f[m++] = calc_first[i][j];
         } else {
            if (production[c1][c2] == '\0') {
               follow(production[c1][0]);
            } else {
               followfirst(production[c1][c2], c1, c2 + 1);
            }
         }
         j++;
      }
   }
}
```

B.2 Input and Output:

(A) Output to remove left recursion from grammar:



(B) Output to find first and follow:



B.3 Observations and learning:

1) A Grammar G (V, T, P, S) is left recursive if it has a production in the form.
$A \rightarrow A\alpha | \beta$.
The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with
$A \rightarrow \beta A'$
$A \rightarrow \alpha A' | \epsilon$
2) **FIRST ()** - It is a function that gives the set of terminals that begin the strings derived from the production rule.
A symbol c is in FIRST (α) if and only if α cβ for some sequence β of grammar symbols.
3) A terminal symbol a is in **FOLLOW (N)** if and only if there is a derivation from the start symbol S of the grammar such that
$S \rightarrow \alpha N\alpha\beta$, where α and β are a (possible empty) sequence of grammar symbols. In other words, a terminal c is in FOLLOW (N) if c can follow N at some point in a derivation.

B.4 Conclusion:

In this experiment, we have successfully implemented the program to remove left recursion from grammar and found out first and follow of the given grammar

B.5 Question of Curiosity

1. What is the need of Predictive parser?

A Predictive Parser is a type of top-down parser used for parsing context-free grammars without backtracking. It is essential because:

1. Eliminates Backtracking
    - Unlike recursive descent parsers, which may need to backtrack (try different rules when an error occurs), predictive parsing eliminates backtracking by using lookahead symbols.
    - This makes parsing faster and more efficient.
2. Efficient Parsing using LL(1) Grammars
    - Predictive parsers work best with LL(1) grammars (Left-to-right scanning, Leftmost derivation, and 1-token lookahead).
    - The parsing decision is made using a single lookahead symbol, reducing complexity.
3. Table-Driven Parsing (Non-Recursive)
    - It can be implemented using a parsing table and a stack, making it iterative instead of recursive.
    - This avoids excessive recursion, reducing memory usage and improving performance.
4. Useful for Compiler Design
    - Predictive parsing is widely used in compilers for syntax analysis of programming languages.
    - It helps in checking whether the syntax of a program follows the grammar of the language.
5. Easy to Implement for Well-Defined Grammars
    - If a grammar is free from left recursion and ambiguity, predictive parsing becomes simple to implement.
    - It is deterministic, meaning each input symbol leads to only one possible action.

Example of Predictive Parsing

Consider the grammar:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

A predictive parser constructs a parse table based on FIRST and FOLLOW sets, allowing direct parsing decisions without backtracking.

2. Difference between top-down and bottom-up parser?

Feature	Top-Down Parser	Bottom-Up Parser
Definition	Starts from the **start symbol** and derives the input string using **leftmost derivation**.	Starts from the **input string** and reduces it to the **start symbol** using **rightmost derivation in reverse**.
Parsing Approach	Expands non-terminals to match the input (derivation-driven).	Reduces input into non-terminals until reaching the start symbol (shift-reduce approach).
Example Parsers	- Recursive Descent Parser - Predictive Parser (LL(1))	- Shift-Reduce Parser - LR Parsers (SLR, CLR, LALR)
Backtracking	May require backtracking unless it's a **predictive parser**.	No backtracking needed, as it efficiently reduces input.
Parsing Order	Uses **leftmost derivation**.	Uses **rightmost derivation (in reverse)**.
Grammar Used	Works with **LL(k) grammars** (non-left-recursive, non-ambiguous).	Works with **LR(k) grammars** (more general, can handle left recursion).
Efficiency	Less powerful, struggles with ambiguous and left-recursive grammars.	More powerful, handles a broader class of grammars.
Stack Usage	Uses an **explicit or implicit stack** (via recursion).	Uses an **explicit stack** for shift-reduce operations.
Predictability	Must decide expansion based on lookahead tokens (LL(1) requires one-token lookahead).	More deterministic, constructs parse tree without guessing.
Usage	Used in **simple parsers and compilers** (e.g., hand-written parsers).	Used in **complex compilers** (e.g., Yacc, Bison, modern programming languages).

3. Why there is a necessity of removing a left recursion?

Necessity of Removing Left Recursion

Left recursion in a grammar occurs when a non-terminal symbol refers to itself as the first symbol in one of its production rules. This causes issues in certain types of parsers, especially top-down parsers like recursive descent parsers, which can enter an infinite loop while trying to expand the left-recursive production.

Problems Caused by Left Recursion:

1. Infinite Recursion in Top-Down Parsing:
    - A recursive descent parser expands the leftmost non-terminal first.
    - If the grammar has left recursion, the parser keeps calling itself indefinitely, leading to non-termination.
    - Example:
    - $A \rightarrow A\,\alpha \mid \beta$

      The parser trying to expand A will keep calling A infinitely.

2. Parsing Failure in LL(1) Parsers:
    - LL(1) parsers (which predict the next rule based on one lookahead token) cannot handle left recursion because they do not backtrack.
    - Example:
    - E → E + T | T

      When parsing E, the parser gets stuck trying to expand E before moving to + T.

3. Inefficiency in Recursive Descent Parsing:
    - Even if backtracking is allowed, left-recursive grammars make parsing inefficient and increase time complexity.

Solution: Removing Left Recursion

To convert left-recursive grammar into right-recursive (or non-recursive) form, we use the following transformation:

General Form of Left Recursion:

A → Aα | β

where:

- A is a non-terminal.
- α is a sequence of symbols (can be terminals or non-terminals).
- β is an alternative rule that does not start with A.

Converted to Right Recursion:

A → β A'
A' → α A' | ε

where:

- A' is a new non-terminal that removes the left recursion.
- ε represents an empty production.

Example:

Given Grammar (Left Recursion):

E → E + T | T

Converted Grammar (Without Left Recursion):

E → T E'
E' → + T E' | ε

This transformation allows the parser to process input without infinite recursion.