**Name: Sparsh Karna**

**Reg Number: 23BDS1172**

**Lab Excercise: 7**

**Aim:**

To develop a C program that accepts a context-free grammar (CFG) as input and computes the **FIRST** and **FOLLOW** sets for all non-terminals in the grammar. This is essential for constructing predictive parsers and understanding the syntax of programming languages.

**Procedure:**

1. Define a grammar using productions with non-terminals (uppercase letters) and terminals (lowercase letters or symbols).
2. Represent epsilon productions using the symbol #.
3. Implement two recursive functions:
   1. **FIRST(X):** Finds the set of terminals that begin the strings derivable from symbol X.
   2. **FOLLOW(X):** Finds the set of terminals that can appear immediately to the right of X in some sentential form.
4. For computing FIRST:
   1. If X is a terminal, then FIRST(X) = {X}.
   2. If X → ε is a production, add ε to FIRST(X).
   3. If X → Y1Y2...Yn, then add FIRST(Y1) (excluding ε). If ε ∈ FIRST(Y1), then also include FIRST(Y2), and so on.
5. For computing FOLLOW:
   1. Place $ (end marker) in FOLLOW of the start symbol.
   2. If there is a production A → αBβ, then everything in FIRST(β) except ε is in FOLLOW(B).
   3. If A → αB or A → αBβ where ε ∈ FIRST(β), then everything in FOLLOW(A) is in FOLLOW(B).
6. Store results in arrays to avoid duplicates.
7. Display the computed FIRST and FOLLOW sets for each non-terminal.

**Algorithm:**

1. Read the number of productions and store them.
2. For each non-terminal on the left-hand side (LHS):
   1. Compute FIRST set recursively:
      1. If the first symbol on RHS is a terminal → add to FIRST.
      2. If it is a non-terminal → recursively compute its FIRST.
      3. If ε-production exists → include ε.
   2. Compute FOLLOW set recursively:
      1. If the non-terminal appears before another symbol, add FIRST of that symbol (excluding ε).
      2. If it appears at the end or the next symbol derives ε → add FOLLOW of the LHS.
      3. Add $ to FOLLOW of the start symbol.
3. Ensure sets do not contain duplicate elements.
4. Print FIRST and FOLLOW sets for each non-terminal.

**Program:**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAX 20

char production[MAX][MAX];

char firstSets[MAX][MAX];

char followSets[MAX][MAX];

int n;

void computeFirst(char \*result, char c);

void computeFollow(char \*result, char c);

int isNonTerminal(char c);

void addToSet(char \*set, char c);

int contains(char \*set, char c) {

for (int i = 0; set[i] != '\0'; i++) {

if (set[i] == c) return 1;

}

return 0;

}

void addToSet(char \*set, char c) {

if (!contains(set, c)) {

int len = strlen(set);

set[len] = c;

set[len + 1] = '\0';

}

}

void computeFirst(char \*result, char c) {

if (!isNonTerminal(c)) {

addToSet(result, c);

return;

}

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

if (production[i][0] == c) {

if (production[i][2] == '#') {

addToSet(result, '#');

} else {

int j = 2;

while (production[i][j] != '\0') {

char symbol = production[i][j];

char temp[MAX] = "";

computeFirst(temp, symbol);

for (int k = 0; temp[k] != '\0'; k++) {

if (temp[k] != '#') addToSet(result, temp[k]);

}

if (contains(temp, '#')) {

j++;

if (production[i][j] == '\0') addToSet(result, '#');

} else break;

}

}

}

}

}

void computeFollow(char \*result, char c) {

if (production[0][0] == c) addToSet(result, '$');

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

for (int j = 2; production[i][j] != '\0'; j++) {

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

char next = production[i][j + 1];

if (next != '\0') {

char temp[MAX] = "";

computeFirst(temp, next);

for (int k = 0; temp[k] != '\0'; k++) {

if (temp[k] != '#') addToSet(result, temp[k]);

}

if (contains(temp, '#')) {

char tempFollow[MAX] = "";

computeFollow(tempFollow, production[i][0]);

for (int k = 0; tempFollow[k] != '\0'; k++) {

addToSet(result, tempFollow[k]);

}

}

} else {

if (c != production[i][0]) {

char tempFollow[MAX] = "";

computeFollow(tempFollow, production[i][0]);

for (int k = 0; tempFollow[k] != '\0'; k++) {

addToSet(result, tempFollow[k]);

}

}

}

}

}

}

}

int isNonTerminal(char c) {

return isupper(c);

}

int main() {

printf("Enter number of productions: ");

scanf("%d", &n);

printf("Enter productions (use # for epsilon):\n");

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

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

}

printf("\nFIRST and FOLLOW sets:\n");

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

char nonTerminal = production[i][0];

char first[MAX] = "";

char follow[MAX] = "";

computeFirst(first, nonTerminal);

computeFollow(follow, nonTerminal);

printf("FIRST(%c) = { ", nonTerminal);

for (int j = 0; first[j] != '\0'; j++) {

printf("%c ", first[j]);

}

printf("}\n");

printf("FOLLOW(%c) = { ", nonTerminal);

for (int j = 0; follow[j] != '\0'; j++) {

printf("%c ", follow[j]);

}

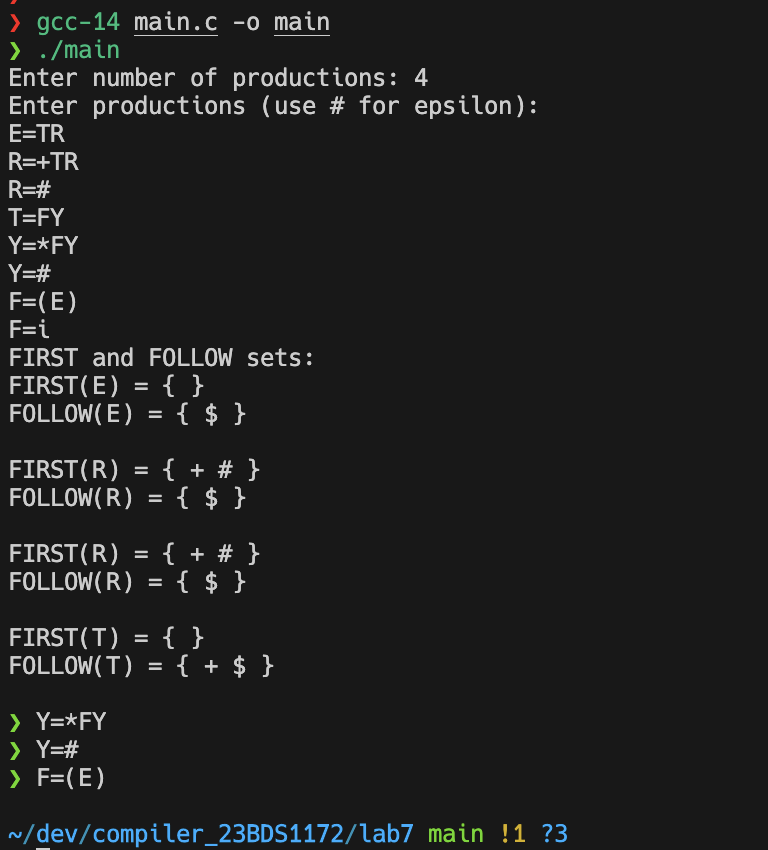
printf("}\n\n");

}

return 0;

}

**Output:**



**Result:**

The program successfully computes the **FIRST** and **FOLLOW** sets of a given context-free grammar. It correctly handles epsilon productions, recursive non-terminals, and ensures that sets contain unique symbols. This implementation can be extended for LL(1) parsing table construction.