Name	Roll No.	Section	DoP
Vidhan Dahatkar	166	B	

Practical - 5

Aim- Develop a program to find FIRST and FOLLOW of all variables. Write a suitable data structure to store a context fee grammar. Prerequisite is to eliminate left recursion from the grammar before storing .

Theory:

Introduction:

In compiler design, parsing a context-free grammar is a fundamental step in syntax analysis. However, to parse a grammar correctly, especially using top-down parsers, it is essential to first remove left recursion and compute the FIRST and FOLLOW sets for non-terminals. These sets are crucial in building predictive parsers such as LL(1) parsers.

Left Recursion: A grammar is left-recursive if a non-terminal A has a production A -> Aα | β, which can cause infinite recursion in top-down parsers. To eliminate left recursion, we transform productions of the form: A -> Aα | β into: A -> βA' A' -> αA' | ε where A' is a new non-terminal and ε denotes the empty string.

FIRST Set: The FIRST set of a symbol is the set of terminals that begin the strings derivable from the symbol. It helps in determining which production to use in predictive parsing.

Rules:

1. If X is a terminal, FIRST(X) = {X}
2. If X -> ε is a production, then ε is in FIRST(X)
3. If X -> Y1Y2...Yn, then add FIRST(Y1) to FIRST(X). If Y1 can derive ε, add FIRST(Y2), and so on.

FOLLOW Set: The FOLLOW set of a non-terminal A is the set of terminals that can appear immediately to the right of A in some sentential form. It is used to determine the valid symbols that can follow a non-terminal in a parse.

Rules:

1. Place $ (end of input) in FOLLOW(S), where S is the start symbol.
2. If there is a production A -> αBβ, add FIRST(β) - {ε} to FOLLOW(B).
3. If β can derive ε or is empty, add FOLLOW(A) to FOLLOW(B).

Data Structures: To represent the grammar and compute the sets efficiently, the following data structures can be used:

1. **Map<string,vector>** productions;
   - o Stores the productions with non-terminals as keys and their corresponding productions as values.
2. **Set firstSet[variable**];
   - o Stores the FIRST set for each non-terminal.
3. **Set followSet[variable];**
   - o Stores the FOLLOW set for each non-terminal.
4. **List of Non-terminals and Terminals:**
   - o Helps in differentiating symbols and applying rules accordingly.

Algorithm Overview:

1. Input the number of productions and read each production.
2. Store the grammar in a map or appropriate data structure.
3. Eliminate left recursion for each non-terminal.
4. Compute FIRST sets using recursive or iterative approach.
5. Compute FOLLOW sets using the computed FIRST sets and applying the rules.
6. Display the FIRST and FOLLOW sets for each non-terminal.

Program:

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

#define MAX 10

// Global variables
char production[MAX][10];
char first[MAX][MAX], follow[MAX][MAX];
int numProductions;
char nonTerminals[MAX];
int firstCount[MAX], followCount[MAX];
int ntCount = 0;

// Function to check if a character is a non-terminal
int isNonTerminal(char c) {
   return isupper(c);
}

// Function to add a character to a set if it's not already present
void addToSet(char set[], int *count, char c) {
```

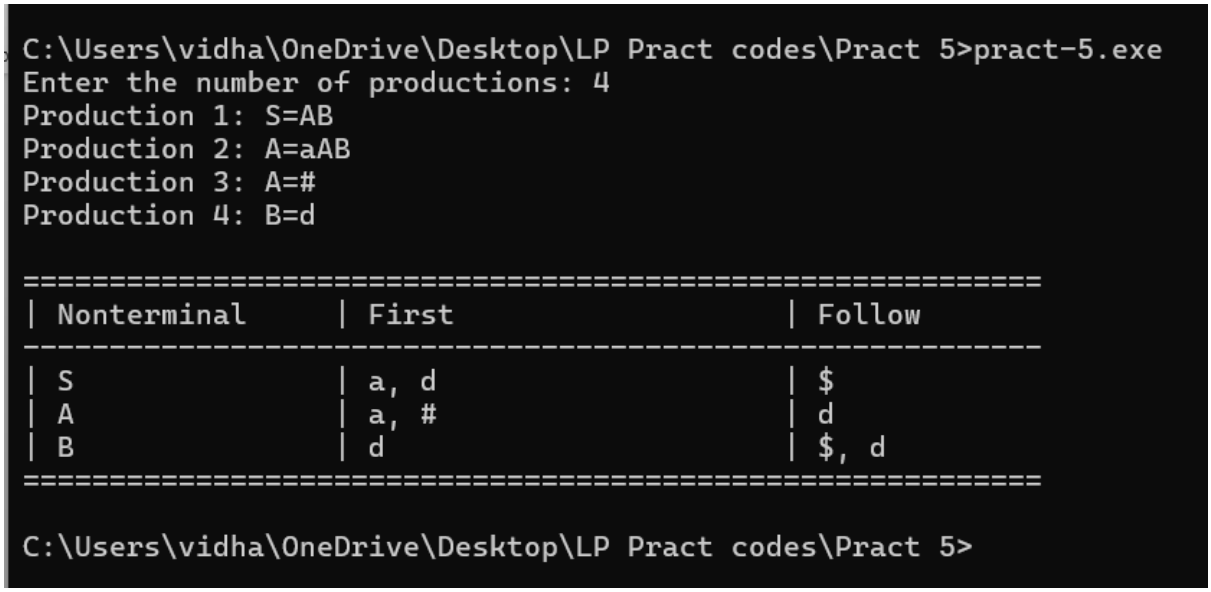
```
    for (int i = 0; i < *count; i++) {
        if (set[i] == c) return;
    }
    set[(*count)++] = c;
}

// Function to get the index of a non-terminal in the nonTerminals array
int getIndex(char c) {
    for (int i = 0; i < ntCount; i++) {
        if (nonTerminals[i] == c) return i;
    }
    return -1;
}

// Function to compute FIRST set for a non-terminal
void computeFirst(char c) {
    int index = getIndex(c);
    for (int i = 0; i < numProductions; i++) {
        if (production[i][0] == c) {
            int j = 2; // Skip the non-terminal and ->
            while (production[i][j] != '\0') {
                char symbol = production[i][j];

                if (!isNonTerminal(symbol)) {
                    // If terminal, add to FIRST set
                    addToSet(first[index], &firstCount[index], symbol);
                    break;
                } else {
                    // If non-terminal, compute its FIRST if not already done
                    int nextIndex = getIndex(symbol);
                    if (firstCount[nextIndex] == 0)
                        computeFirst(symbol);

                    // Add all non-epsilon symbols from the FIRST set
                    int epsilonFound = 0;
                    for (int k = 0; k < firstCount[nextIndex]; k++) {
                        if (first[nextIndex][k] == '#') {
                            epsilonFound = 1;
                        } else {
                            addToSet(first[index], &firstCount[index], first[nextIndex][k]);
                        }
                    }

                    // If epsilon not found, stop processing this production
                    if (!epsilonFound) break;

                    j++;
```

```
                // If we reached end of production, add epsilon
                if (production[i][j] == '\0') {
                    addToSet(first[index], &firstCount[index], '#');
                }
            }
        }
    }
}

// Function to compute FOLLOW set for a non-terminal
void computeFollow(char c) {
    int index = getIndex(c);

    // Rule 1: $ is in FOLLOW(S) where S is the start symbol
    if (production[0][0] == c)
        addToSet(follow[index], &followCount[index], '$');

    for (int i = 0; i < numProductions; i++) {
        char *rhs = production[i];
        for (int j = 2; rhs[j] != '\0'; j++) {
            if (rhs[j] == c) {
                // Case 1: A → αBβ
                if (rhs[j + 1] != '\0') {
                    char next = rhs[j + 1];

                    // Subcase 1: β is a terminal
                    if (!isNonTerminal(next)) {
                        addToSet(follow[index], &followCount[index], next);
                    }
                    // Subcase 2: β is a non-terminal
                    else {
                        int nextIndex = getIndex(next);

                        // Add FIRST(β) except ε to FOLLOW(B)
                        for (int k = 0; k < firstCount[nextIndex]; k++) {
                            if (first[nextIndex][k] != '#') {
                                addToSet(follow[index], &followCount[index], first[nextIndex][k]);
                            }
                        }

                        // If FIRST(β) contains ε, add FOLLOW(A) to FOLLOW(B)
                        if (strchr(first[nextIndex], '#') != NULL) {
                            int lhsIndex = getIndex(production[i][0]);
                            if (followCount[lhsIndex] == 0)
                                computeFollow(production[i][0]);
                            for (int k = 0; k < followCount[lhsIndex]; k++) {
```

```
                    addToSet(follow[index], &followCount[index], follow[lhsIndex][k]);
                }
            }
        }
    }
    // Case 2: A → αB (or A → αBβ where β →* ε)
    else {
        int lhsIndex = getIndex(production[i][0]);
        if (lhsIndex != index) {
            if (followCount[lhsIndex] == 0)
                computeFollow(production[i][0]);
            for (int k = 0; k < followCount[lhsIndex]; k++) {
                addToSet(follow[index], &followCount[index], follow[lhsIndex][k]);
            }
        }
    }
  }
}

// Function to print the FIRST and FOLLOW sets in a table format
void printTable() {
  printf("\n============================================================\n");
  printf("| %-15s | %-23s | %-25s |\n", "Nonterminal", "First", "Follow");
  printf("------------------------------------------------------------\n");

  for (int i = 0; i < ntCount; i++) {
    char firstSet[30] = "";
    char followSet[30] = "";

    // Format FIRST set
    for (int j = 0; j < firstCount[i]; j++) {
      strncat(firstSet, &first[i][j], 1);
      if (j < firstCount[i] - 1) {
        strcat(firstSet, ", ");
      }
    }

    // Format FOLLOW set
    for (int j = 0; j < followCount[i]; j++) {
      strncat(followSet, &follow[i][j], 1);
      if (j < followCount[i] - 1) {
        strcat(followSet, ", ");
      }
    }
```

```
        // Print row with formatted columns
        printf("| %-15c | %-23s | %-25s |\n", nonTerminals[i], firstSet, followSet);
    }

    printf("========================================================\n");
}

int main() {
    printf("Enter the number of productions: ");
    scanf("%d", &numProductions);
    getchar(); // Consume the newline character

    // Input productions
    for (int i = 0; i < numProductions; i++) {
        printf("Production %d: ", i + 1);
        fgets(production[i], sizeof(production[i]), stdin);
        production[i][strcspn(production[i], "\n")] = '\0'; // Remove newline

        // Add non-terminal to list if not already present
        char nt = production[i][0];
        if (getIndex(nt) == -1) {
            nonTerminals[ntCount++] = nt;
        }
    }

    // Compute FIRST sets for all non-terminals
    for (int i = 0; i < ntCount; i++) {
        computeFirst(nonTerminals[i]);
    }

    // Compute FOLLOW sets for all non-terminals
    for (int i = 0; i < ntCount; i++) {
        computeFollow(nonTerminals[i]);
    }

    // Print the results
    printTable();

    return 0;
}
```

Output:



Conclusion:

This program integrates the theory of compiler design and practical implementation using suitable data structures and parsing techniques. Eliminating left recursion and computing FIRST and FOLLOW sets are foundational steps for developing efficient parsers and syntax analysers.