Name: Pradyumn R Pai
Roll No: 50
Class: CS7A

PROGRAM CODE

grammar.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

struct ProductionRule{
    char symbol;
    char expression[20];
};

struct Grammar{
    char startState;
    char* non_terminals;
    char* terminals;
    struct ProductionRule* rules;
    int production_num;
};

struct LMDStackNode {
    struct ProductionRule rule;
    struct LMDStackNode* next;
};

struct LMDStackNode* head = NULL;

void free_grammar(struct Grammar* g){
    if (!g) return;
```

```
    if (g->non_terminals) free(g->non_terminals);
    if (g->terminals) free(g->terminals);
    if (g->rules) free(g->rules);
    free(g);
}

int find_index(char s[], char c){
    int n = strlen(s);
    for (int i=0;i<n;++i){
        if (s[i]==c){
            return i;
        }
    }
    return -1;
}

bool str_contains(char str[],char c){
    return find_index(str,c)!=-1;
}



void add_str(char str[], char c){
    int n = strlen(str);
    for (int i=0;i<n;++i){
        if (str[i]==c){
            return ;
        }
    }
    str[n] = c;
    str[n+1] = '\0';
}
```

```
bool validTerminal(struct Grammar* g, char c){
    return str_contains(g->terminals,c);
}

bool validNonTerminal(struct Grammar* g, char c){
    return str_contains(g->non_terminals,c);
}

bool validInput(struct Grammar* g, char input[]){
    int n = strlen(input);
    for (int i=0;i<n;++i){
        if (!validTerminal(g,input[i])){
            return true;
        }
    }
    return true;
}

bool validExpansion(struct Grammar* g, char input[]){
    int n = strlen(input);
    if (n==1 && input[0]=='e') return true;
    for (int i=0;i<n;++i){
        if (!validTerminal(g,input[i]) && !validNonTerminal(g,input[i])){
            return true;
        }
    }
    return true;
}

struct Grammar* read_grammar() {
```

```
int num_non_terminal, num_terminal, num_production_rule;
scanf("%d %d %d",&num_non_terminal,&num_terminal,&num_production_rule);

struct Grammar* g = malloc(sizeof(struct Grammar));
if (!g){
    printf("Coulnd't create grammar\n");
    return NULL;
}
scanf(" %c",&g->startState);
if (g->startState==EOF){
    printf("Reached EOF when reading start state\n");
    free_grammar(g);
    return NULL;
}
g->production_num = num_production_rule;

//Read non terminals
g->non_terminals = malloc(sizeof(char)*num_non_terminal);
if (!g->non_terminals){
    printf("Couldnt' allocate non terminals\n");
    free_grammar(g);
    return NULL;
}
for (int i=0;i<num_non_terminal;++i){
    char c;
    scanf(" %c",&c);
    if (c==EOF){
        printf("Reached EOF when reading non terminals\n");
        free_grammar(g);
        return NULL;
    }
```

```
        g->non_terminals[i] = c;
    }
    g->non_terminals[num_non_terminal] = '\0';

    //Read terminals
    g->terminals = malloc(sizeof(char)*num_terminal);
    for (int i=0;i<num_terminal;++i){
        char c;
        scanf(" %c",&c);
        if (c==EOF){
            printf("Reached EOF when reading terminals\n");
            free_grammar(g);
            return NULL;
        }
        g->terminals[i] = c;
    }
    g->terminals[num_terminal] = '\0';

    //Read Production Rules
    g->rules = malloc(sizeof(struct ProductionRule)*num_production_rule);
    if (!g){
        printf("Error reading production rules\n");
        free_grammar(g);
        return NULL;
    }
    for (int i=0;i<num_production_rule;++i){
        char rule[20];
        scanf("%s",rule);
        sscanf(rule,"%c->%s",&(g->rules[i].symbol),&g->rules[i].expression);
        if (!validNonTerminal(g,g->rules[i].symbol) || !validExpansion(g,g->rules[i].expression))
{
```

```
            printf("Production rule %s invalid\n",rule);
            if (!validNonTerminal(g,g->rules[i].symbol)){
                printf("Invalid symbol on LHS\n");
            }
            if (!validExpansion(g,g->rules[i].expression)){
                printf("Invalid expression on RHS");
            }
            free_grammar(g);
            return NULL;
        }
    }

    return g;
}

void push_derivation(struct ProductionRule r){
    struct LMDStackNode* n = malloc(sizeof(struct LMDStackNode));
    n->next = head;
    n->rule = r;
    head = n;
}

bool empty_derivation(){
    if (head) return false;
    return true;
}

void pop_derivation(){
    if (!head) return;
    struct LMDStackNode* n = head->next;
    free(head);
```

```
    head = n;
}

struct ProductionRule top_derivation(){
    return head->rule;
}



void print_delete_derivation(){
    if (empty_derivation()) return;
    struct ProductionRule p = top_derivation();
    pop_derivation();
    print_delete_derivation();
    printf("%c->%s\n",p.symbol,p.expression);
}
```

first_follow.c:

```
#include "grammar.c"
int n,m;
bool changed;

void update(int** matrix, int i, int j, bool new){
    if (!new || matrix[i][j]) return;
    matrix[i][j] = true;
    changed = true;
}

void update_set(int* st1, int* st2, int size){
    for (int i=0;i<size;++i){
        if (!st1[i] && st2[i]){
            st1[i] = true;
            changed = true;
```

```
        }
    }
}

void find_first(struct Grammar* g, int** first){
    changed = true;
    int production_num = g->production_num;
    while (changed){
        changed = false;
        for (int p=0;p<production_num;++p){
            int X = find_index(g->non_terminals,g->rules[p].symbol);
            char* expression = g->rules[p].expression;
            if (X<0){
                continue;
            }
            int k = strlen(expression);
            if (k==1 && expression[0]=='e'){
                update(first,X,m,true);
                continue;
            }

            bool nullable = true;
            for (int i=0;i<k;++i){
                int Y = find_index(g->non_terminals,expression[i]);
                if (Y<0){
                    // Terminal encountered
                    int t = find_index(g->terminals,expression[i]);
                    update(first,X,t,true);
                    nullable = false;
                    break;
                }
```

```
            // Add all symbols from FIRST(Y) to FIRST(X) except epsilon
            update_set(first[X], first[Y], m);

            // If Y doesn't have epsilon, stop
            if (!first[Y][m]){
                nullable = false;
                break;
            }
        }

        if (nullable){
            update(first,X,m,true);
        }
    }
  }
}

void find_follow(struct Grammar* g, int** first,int** follow){
  changed = true;
  int production_num = g->production_num;
  // Add $ to follow set of start symbol
  int start_idx = find_index(g->non_terminals, g->startState);
  if (start_idx >= 0) {
    follow[start_idx][m] = true;
  }
  while (changed){
    changed = false;
    for (int p=0;p<production_num;++p){
      int X = find_index(g->non_terminals,g->rules[p].symbol);
      char* expression = g->rules[p].expression;
```

```
            if (X<0){
                continue;
            }
            int k = strlen(expression);
            for (int i=0;i<k;++i){
                int Y = find_index(g->non_terminals,expression[i]);
                if (Y<0){
                    continue;
                }
                bool nullable = true;
                for (int j=i+1;j<k && nullable;++j){
                    int Z = find_index(g->non_terminals,expression[j]);
                    if (Z<0){
                        int t = find_index(g->terminals,expression[j]);
                        update(follow,Y,t,true);
                        nullable = false;
                        break;
                    }
                    if (!first[Z][m]){
                        nullable = false;
                    }
                    update_set(follow[Y],first[Z],m);
                }
                if (nullable){
                    update_set(follow[Y],follow[X],m+1); //Include m representing $
                }
            }
        }
    }
}
```

```
void find_first_follow(struct Grammar* g){
    n = strlen(g->non_terminals);
    m = strlen(g->terminals);
    g->terminals[m] = 'e';
    g->terminals[m+1] = '\0';

    int** first = malloc(sizeof(int*)*n);
    int** follow = malloc(sizeof(int*)*n);
    for (int i=0;i<n;++i){
        first[i] = malloc(sizeof(int)*(m+1));
        follow[i] = malloc(sizeof(int)*(m+1));
        for (int j=0;j<=m;++j){
            first[i][j] = 0;
            follow[i][j] = 0;
        }
    }

    find_first(g,first);
    find_follow(g,first,follow);

    for (int i=0;i<n;++i){
        printf("First(%c) = {",g->non_terminals[i]);
        bool flag = false;
        for (int j=0;j<=m;++j){
            if (!first[i][j]) continue;
            if (flag){
                printf(",");
            }
            flag = true;
            char c = 'e';
            if (j!=m){
```

```
            c = g->terminals[j];
        }
        printf("%c",c);
    }
    printf("}\n");

    printf("Follow(%c) = {",g->non_terminals[i]);
    flag = false;
    for (int j=0;j<=m;++j){
        if (!follow[i][j]) continue;
        if (flag){
            printf(",");
        }
        flag = true;
        char c = '$';
        if (j!=m){
            c = g->terminals[j];
        }
        printf("%c",c);
    }
    printf("}\n");
  }

  for (int i=0;i<n;++i){
    free(first[i]);
    free(follow[i]);
  }
  free(first);
  free(follow);
}
```

```
int main(){
    struct Grammar* g = read_grammar();
    find_first_follow(g);
    free(g);
    return 0;
}
```

OUTPUT:

input.txt:

4 3 6

T

TQRS

xyz

T->Qx

Q->RS

R->y

R->e

S->z

S->e

xyxyz

Output:

First(T) = {x,y,z}

Follow(T) = {$}

First(Q) = {y,z,e}

Follow(Q) = {x}

First(R) = {y,e}

Follow(R) = {x,z}

First(S) = {z,e}

Follow(S) = {x}