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 recursiveDescent(struct Grammar* g, char input[],int inputStart, char expanded[], int
expandedStart){
    if (expanded[expandedStart]=='\0'){
        return strcmp(input,expanded)==0;
    }

    if (input[inputStart]==expanded[expandedStart]){
        return recursiveDescent(g, input, inputStart + 1, expanded, expandedStart + 1);
    }

    char current = expanded[expandedStart];
    for (int i=0;i<g->production_num;++i){
        if (current==g->rules[i].symbol){
```

```
            char expanded_copy[100];
            strcpy(expanded_copy,expanded);

            expanded[expandedStart] = '\0';
            if (g->rules[i].expression[0]!='e' || g->rules[i].expression[1]!='\0'){
                strcat(expanded,g->rules[i].expression);
            }
            strcat(expanded,expanded_copy+expandedStart+1);
            push_derivation(g->rules[i]);

            if (recursiveDescent(g,input,inputStart,expanded,expandedStart)) {
                return true;
            }

            pop_derivation();
            strcpy(expanded,expanded_copy);
        }
    }
    return false;
}

bool parse(struct Grammar* g,char input[]){
    char expanded[100];
    expanded[0] = g->startState;
    expanded[1] = '\0';
    return recursiveDescent(g,input,0,expanded,0);
}

int main(){
    struct Grammar* g = read_grammar();
    char input[20];
```

```
    scanf("%s",input);
    if (parse(g,input)){
        printf("String accepted\n");
    } else {
        printf("String rejected\n");
    }
    free(g);
    print_delete_derivation();
}
```

OUTPUT:

input.txt:

2 2 3

E

EZ

+i

E->iZ

Z->+iZ

Z->e

i+i+i

Output:

String accepted

E->iZ

Z->+iZ

Z->+iZ

Z->e