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);
}
```

stack.c:

```
#include "grammar.c"

struct StackNode{
    char symbol;
    char firstTerminal;
    struct StackNode* next;
};



bool emptyStack(struct StackNode** indirect){
    if (*indirect) return false;
    return true;
}

void stackPush(struct StackNode** indirect,char symbol, bool terminal){
```

```
    char firstTerminal = '$';
    if (terminal){
        firstTerminal = symbol;
    } else if (*indirect){
        firstTerminal = (*indirect)->firstTerminal;
    }
    struct StackNode* st = malloc(sizeof(struct StackNode));
    st->symbol = symbol;
    st->firstTerminal = firstTerminal;
    st->next = *indirect;
    *indirect = st;
}

void popStack(struct StackNode** indirect){
    if (*indirect){
        struct StackNode* st = *indirect;
        *indirect = st->next;
        free(st);
    }
}

char stackTopValue(struct StackNode** indirect){
    if (*indirect){
        return (*indirect)->symbol;
    }
    return '$';
}

char stackTerminal(struct StackNode** indirect){
    if (!emptyStack(indirect)){
        return (*indirect)->firstTerminal;
```

```
    }
    return '$';
}
void freeStack(struct StackNode** indirect){
    while (!emptyStack(indirect)){
        popStack(indirect);
    }
}

void printState(struct StackNode** indirect){
    while (*indirect){
        printf("%c",(*indirect)->symbol);
        // printf("(%c,%c)",(*indirect)->symbol,(*indirect)->firstTerminal);
        indirect = &((*indirect)->next);
    }
    printf("$");
}
```

shift_reduce_common.c:

```
#include "stack.c"

bool match(struct StackNode** indirect, char s[]){
    int n = strlen(s);
    struct StackNode* current = *indirect;

    for (int i=n-1;i>=0;--i){
        if (!current){
            return false;
        }
        char lhs = current->symbol;
        char rhs = s[i];
        if (lhs!=rhs){
```

```
            return false;
        }

        current = current->next;
    }

    // Don't remove matched nodes here, that happens in reduce()

    return true;
}

bool shift(struct StackNode** inputStack,struct StackNode** outputStack){
    if (!emptyStack(inputStack)){
        stackPush(outputStack,stackTopValue(inputStack),true);
        popStack(inputStack);
        printf("Action: Shift Input: ");
        printState(inputStack);
        printf(" Output: ");
        printState(outputStack);
        printf("\n");
        return true;
    }
    return false;
}

bool reduce(struct StackNode** outputStack,struct StackNode** inputStack,struct Grammar*
g){
    int np = g->production_num;
    bool res = false;
    for (int p=0;p<np;++p){
        char* expression = g->rules[p].expression;
```

```
        char symbol = g->rules[p].symbol;
        if (match(outputStack,expression)){
            int n = strlen(expression);
            while (n-->0){
                popStack(outputStack);
            }
            stackPush(outputStack,symbol,false);
            push_derivation(g->rules[p]);
            res = true;
            printf("Action: Reduce Input: ");
            printState(inputStack);
            printf(" Output: ");
            printState(outputStack);
            printf("\n");
        }
    }
    return res;
}

void derivation_parse(){
    printf("The RMD is as follows:\n");
    while (!empty_derivation()){
        struct ProductionRule r = top_derivation();
        printf("%c->%s\n",r.symbol,r.expression);
        pop_derivation();
    }
}
```

operator.c:

```
#include "shift_reduce_common.c"

char ** read_precedence(struct Grammar* g){
```

```
    int n = strlen(g->terminals)+1;
    char ** table = malloc(sizeof(char*)*n);
    for (int i=0;i<n;++i){
        table[i] = malloc(sizeof(char)*n);
        scanf("%s",table[i]);
    }
    return table;
}

void free_precedence(char** table, int n){
    for (int i=0;i<n;++i){
        free(table[i]);
    }
    free(table);
}

char findAction(struct Grammar* g,char**table, struct StackNode** outputStack,struct
StackNode** inputStack){
    char lhs = stackTerminal(outputStack);
    char rhs = stackTerminal(inputStack);
    int l = find_index(g->terminals,lhs);
    int r = find_index(g->terminals,rhs);
    if (l<0) l = strlen(g->terminals);
    if (r<0) r = strlen(g->terminals);
    return table[l][r];
}



void parse(struct Grammar* g, char** table,char input[20]){
    struct StackNode * inputHead = NULL;
```

```
struct StackNode * stackHead = NULL;

struct StackNode** inputStack = &inputHead;
struct StackNode** outputStack = &stackHead;
int n = strlen(input);
for (int i=n-1;i>=0;--i){
    stackPush(inputStack,input[i],true);
}
char action = '=';
int max_iterations = 1000;
while (max_iterations-->0 && action!='A'){
    action = findAction(g,table,outputStack,inputStack);


    switch(action){
        case '>':
        case '=':
            //Same for left associative grammar
            if (!reduce(outputStack,inputStack,g)){
                max_iterations = 0;
            }
            break;
        case '<':
            if (!shift(inputStack,outputStack)){
                max_iterations = 0;
            }
            break;
        case 'A': //Accept
            if (emptyStack(inputStack) && !emptyStack(outputStack)
            && !stackHead->next && stackTopValue(outputStack)==g->startState){
                //Valid input
```

```
                printf("String Accepted\n");
            } else {
                printf("String rejected\n");
            }
            break;
        }
    }
    derivation_parse();
    freeStack(inputStack);
    freeStack(outputStack);
}

int main(){
    struct Grammar* g = read_grammar();
    char ** table = read_precedence(g);
    int n = strlen(g->terminals)+1;
    char input[20];
    scanf("%19s",input);
    if (validInput(g,input)){
        parse(g,table,input);
    } else {
        printf("Invalid input\n");
    }
    free_precedence(table,n);
    free_grammar(g);
    return 0;
}
```

OUTPUT:

input.txt:

1 3 3

E

E

i+*

E->E+E

E->E*E

E->i

=>>>

<><>

<><>

<<<A

i+i*i

Output:

Action: Shift Input: +i*i$ Output: i$

Action: Reduce Input: +i*i$ Output: E$

Action: Shift Input: i*i$ Output: +E$

Action: Shift Input: *i$ Output: i+E$

Action: Reduce Input: *i$ Output: E+E$

Action: Shift Input: i$ Output: *E+E$

Action: Shift Input: $ Output: i*E+E$

Action: Reduce Input: $ Output: E*E+E$

Action: Reduce Input: $ Output: E+E$

Action: Reduce Input: $ Output: E$

String Accepted

The RMD is as follows:

E->E+E

E->E*E

E->i

E->i

E->i