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;
```

Date: 07/08/2024

Experiment 3.1

AIM

To simulate FIRST and FOLLOW of a grammar

ALGORITHM

1. Start
2. Read grammar G with n non terminals and m terminals.
3. Each non terminal is represented as a number between 0 to n-1 and each terminal as a number between 0 to m-1.
4. Initialize a n*(m+1) 2D array of boolean values to store FIRST and FOLLOW of each non-terminal where the element at index (i,j) represents whether the terminal j is part of FIRST or FOLLOW of i. The index m+1 for terminals represents special characters 'ε' and '$' in FIRST and FOLLOW respectively.
5. Find FIRST of all non terminals by repeating the following till no changes occur to the FIRST set in a given iteration:
    1. For each production rule of form $X \rightarrow Y1Y2...Yk$:
        1. If the production of the form $X \rightarrow \varepsilon$:
            1. Add ε to FIRST(X).
            2. Continue to next iteration.
        2. Set nullable = true.
        3. For i from 1 to k:
            1. If Y_i is a terminal:
                1. Add Y_i to FIRST(X).
                2. Set nullable = false.
                3. Break.
            2. Add all terminals in FIRST(Y_i) to FIRST(X) except 'ε'.
            3. If $\varepsilon \notin$ FIRST(Y_i):
                1. Set nullable = false.
                2. Break.

```
    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';
}
```

4. If nullable, add 'ε' to FIRST(X).

6. Find FOLLOW of all non terminals by repeating the following till no changes occur to the FOLLOW set in a given iteration:
    1. Add '$' to FOLLOW(S) where S is the starting symbol.
    2. For each production rule of form $X \rightarrow Y1Y2...Yk$:
        1. For i from 1 to k:
            1. Set nullable = true.
            2. For j from i+1 to k:
                1. If Y_j is a terminal:
                    1. Add Y_j to FOLLOW(Y_i).
                    2. Set nullable = false.
                    3. Break.
                2. Add all terminals in FIRST(Y_j) to FOLLOW(Y_i) except 'ε'.
                3. If $\varepsilon \notin$ FIRST(Y_j):
                    1. Set nullable = false.
                    2. Break.
            3. If nullable, add all terminals in FOLLOW(X) to FOLLOW(Y_i) including '$'.
7. Display FIRST and FOLLOW of each non-terminal.
8. Stop

```
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}

RESULT

Successfully found FIRST and FOLLOW of all non terminals in a given grammar

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;
```

Date: 25/09/2025

Experiment 3.2

AIM

To develop shift reduce parser for a given grammar

ALGORITHM

1. Start
2. Initialize input and output stack which return '$' as default value if stack is empty.
3. Read Grammar.
4. Read Input.
5. Traverse input string in reverser and add to stack.
6. While input isn't marked as valid or invalid, do the following:
    1. Try to reduce by doing the following:
        1. For each production of the form $X \rightarrow Y_1Y_2..Y_k$:
            1. Check if top k elements of the stack are $Y_k, ,Y_{k-1},\ldots,Y_1$.
            2. If the stack matches the expression, pop k elements on the stack.
            3. Add X to the stack.
            4. Add the production to the Right Most Derivation (in reverse).
            5. Reduction is successful. Therefore, exit for loop.
    2. If reduction is successful, go to next iteration of while loop.
    3. If input stack is not empty, Shift by popping and element from input stack and pushing it into the output stack.
    4. If shifting fails:
        1. Check if output stack has only a single symbol corresponding to the start symbol. If yes, string is marked as accepted.
        2. Otherwise, string is marked as rejected.
7. Display sequence of steps taken and the RMD generated.
8. Stop

```
    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();
    }
}
```

shift_reduce_parse.c:

```
#include "shift_reduce_common.c"
```

```
void parse(struct Grammar* g,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);
    }
    int max_iterations = 1000;
    while (max_iterations-->0){
        //Try reduce
        if (reduce(outputStack,inputStack,g)){
            continue;
        }

        //Try shift
        if (shift(inputStack,outputStack)){
            continue;
        }

        //Accept or reject if neither works
        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();
    int n = strlen(g->terminals)+1;
    char input[20];
    scanf("%19s",input);
    if (validInput(g,input)){
        parse(g,input);
    } else {
        printf("Invalid input\n");
    }
    return 0;
}
```

OUTPUT:

input.txt:

1 3 3

E

E

i+*

E->E+E

E->E*E

E->i

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: Reduce Input: *i$ Output: E$

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

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

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

Action: Reduce Input: $ Output: E$

String Accepted

The RMD is as follows:

E->E*E

E->i

E->E+E

E->i

E->i

RESULT

Successfully implemented shift reduce parser for the given grammar.

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;
```

Date: 25/09/2025

Experiment 3.3

AIM

To develop operator precedence parser for a given grammar

ALGORITHM

1. Start
2. Initialize input and output stack which return '$' as default value if stack is empty.
3. Read Grammar.
4. Read precedence table.
5. Read Input.
6. Traverse input string in reverser and add to stack.
7. While input isn't marked as valid or invalid, do the following:
    1. Let l and r be the top non terminals on output and input stacks respectively. Find action a to take based on precedence table for l and r.
    2. If a='>' or a='=', Try to reduce by doing the following:
        1. For each production of the form $X \rightarrow Y_1Y_2..Y_k$:
            1. Check if top k elements of the stack are $Y_k, ,Y_{k-1},\ldots,Y_1$.
            2. If the stack matches the expression, pop k elements on the stack.
            3. Add X to the stack.
            4. Add the production to the Right Most Derivation (in reverse).
            5. Reduction is successful. Therefore, exit for loop.
        2. If reduction is unsuccessful, mark string as rejected.
    3. Otherwise, if a='<':
        1. If input stack is not empty, Shift by popping and element from input stack and pushing it into the output stack.
        2. If shifting fails, mark string as rejected.
    4. Otherwise, if a='A':
        1. Check if input stack is empty and output stack has only a single symbol corresponding to the start symbol. If yes, string is marked as accepted.

```
    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';
}
```

    2. Otherwise, string is marked as rejected.
8. Display sequence of steps taken and the RMD generated.
9. Stop

```
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

RESULT

Successfully implemented operator precedence parser for the given grammar.

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;
```

Date: 25/09/2025

Experiment 3.4

AIM

To implement recursive descent parsing

ALGORITHM

1. Start
2. Define recursiveDescent function that takes as input a grammar G pointers to strings containing the input string and expanded string and does the following:
    1. If pointer to expanded string is at the end of the string, return whether the pointer to input string is at the end of the string.
    2. If both pointers point to same character, increment both pointers and call and return recursiceDescent recursively.
    3. Let c be the current pointer.
    4. For each production rule in G of form $X \rightarrow Y_1 Y_2 .. Y_k$ where X is the pointed symbol in the expanded string:
        1. Copy the string expanded to another string.
        2. Truncate expanded at the current pointer.
        3. Append $Y_1 Y_2 .. Y_k$ to expanded.
        4. Append the suffix of the pointer from the copy to expanded.
        5. Call recursiveDescent function:
            1. If it returns true, return true.
        6. Restore expanded from the copy.
    5. Return false.
3. Read grammar.
4. Read input.
5. Initialize string expanded with only the start symbol of the grammar.
6. Call recursiveDescent for the grammar with string pointers to beginning of input and expanded.
7. Print LMD generated if string is accepted.
8. Stop

```
    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

RESULT

Successfully implemented recursive descent parsing.

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

PROGRAM CODE

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>


char tempVar = 'A';


bool resolveOperator(char op, char input[]){
    int n = strlen(input);
    for (int i=0;i<n;++i){
        if (input[i]==op){
            if (i==0 || i==(n-1)){
                printf("Error\n");
                return false;
            }
            printf("%c\t%c\t%c\t%c\n",op,tempVar,input[i-1],input[i+1]);
            char temp[100];
            strcpy(temp,input+i+2);
            input[i-1] = tempVar++;
            strcpy(input+i,temp);
        }
    }
    return true;
}


void parse(char input[]){
    printf("Operator\tDestination\tOperand 1\tOperator 2\n");
    if (!resolveOperator('/',input)) return;
```

Date: 29/09/2025

Experiment 4.1

AIM

To implement intermediate code generation

ALGORITHM

1. Start
2. Read arithmetic expression as a string
3. For each operator in order of precedence (‘/’>’*’>’+’>’-’):
    1. Parse input from left to right
    2. If the i^{th} character is an operator:
        1. The $(i-1)^{th}$ character and the $(i+1)^{th}$ character are considered operands.
        2. A new temporary variable is initialized and considered the destination.
        3. Three address code based on operator, operands, and the destination is represented as quadruple and printed.
        4. The ith character and the two characters surrounding it are replaced with the temporary variable.
4. Handle assignment by representing it as as quadruple where the LHS is the operand and the RHS is the destination. The operator is ‘=’.
5. Stop

```
    if (!resolveOperator('*',input)) return;
    if (!resolveOperator('+',input)) return;
    if (!resolveOperator('-',input)) return;
    // Resolve = symbol
    if (strlen(input)==3 && input[1]=='='){
        printf("=\t%c\t%c\n",input[0],input[2]);
        return;
    }
    if (strlen(input)>1){
        printf("Error\n");
    }
}

int main(){
    char input[100];
    scanf("%s",input);
    parse(input);
    return 0;
}
```

OUTPUT:

input.txt:

q=a+b-c/d*2

Output:

Operator	Destination	Operand 1	Operator 2
/	A	c	d
*	B	A	2
+	C	a	b
-	D	C	B
=	q	D	

RESULT

Successfully implemented intermediate code generation