program which accept strings that starts and ends with 0 or 1

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

int main() {
    char str[100];

    // Input the string
    printf("Enter the sequence: ");
    scanf("%s", str);

    // Check the first and last characters
    if ((str[0] == '0' || str[0] == '1') && (str[strlen(str) - 1]
== '0' || str[strlen(str) - 1] == '1')) {
        printf("Sequence Accepted\n");
    } else {
        printf("Sequence Rejected\n");
    }
    return 0;
}
```

program that recognizes whether a given
variable name is valid

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

int isValidVariable(char str[]) {
    // Check if the first character is a letter
    if (isalpha(str[0]) == 0) {
        return 0; // Invalid variable if the first character is not a letter
    }

    // Check the rest of the string for valid characters (letters or digits)
    for (int i = 1; i < strlen(str); i++) {
        if (!(isalnum(str[i]))) {
            return 0; // Invalid variable if the character is neither letter nor digit
        }
    }

    return 1; // Valid variable name
}

int main() {
    char str[100];

    // Input the variable name
    printf("Enter a variable name: ");
    scanf("%s", str);

    // Check if the variable is valid
    if (isValidVariable(str)) {
        printf("Valid Variable Name\n");
    } else {
        printf("Invalid Variable Name\n");
    }

    return 0;
}
```

program to implement a arithmetic operations and recognize a valid statement.

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

// Function to check if the expression is valid
int isValidExpression(char* expr) {
    int len = strlen(expr);

    // Check for empty expression
    if (len == 0) {
        return 0;
    }

    // Check if the expression starts or ends with an operator or has invalid characters
    if (expr[0] == '+' || expr[0] == '-' || expr[0] == '*' || expr[0] == '/' || expr[0] == '%' || expr[0] == '(' ||
        expr[len - 1] == '+' || expr[len - 1] == '-' || expr[len - 1] == '*' || expr[len - 1] == '/' || expr[len - 1] == '%') {
        return 0;
    }

    int openParens = 0;
    int closeParens = 0;

    for (int i = 0; i < len; i++) {
        // Check for invalid characters
        if (!(isdigit(expr[i]) || expr[i] == '+' || expr[i] == '-' || expr[i] == '*' || expr[i] == '/' || expr[i] == '%' || expr[i] == '(' || expr[i] == ')')) {
            return 0;
        }
        // Count parentheses
        if (expr[i] == '(') {
            openParens++;
        }
        if (expr[i] == ')') {
            closeParens++;
        }

        // Ensure that operators are not consecutive
        if ((expr[i] == '+' || expr[i] == '-' || expr[i] == '*' || expr[i] == '/' || expr[i] == '%') &&
            (i == 0 || i == len - 1 || !isdigit(expr[i - 1]) || !isdigit(expr[i + 1]))) {
            return 0;
        }
    }

    // Check if parentheses are balanced
    if (openParens != closeParens) {
        return 0;
    }

    return 1; // Expression is valid
}

// Function to evaluate the expression (basic implementation)
int evaluateExpression(char* expr) {
    int result;
    char operator;
    int num1, num2;

    // Use sscanf to extract parts of the expression
    if (sscanf(expr, "%d%c%d", &num1, &operator, &num2) == 3) {
        switch (operator) {
            case '+':
                result = num1 + num2;
                break;
            case '-':
                result = num1 - num2;
                break;
            case '*':
                result = num1 * num2;
                break;
            case '/':
                if (num2 != 0) {
                    result = num1 / num2;
                } else {
                    printf("Error: Division by zero\n");
                    result = 0;
                }
                break;
            case '%':
                result = num1 % num2;
                break;
            default:
                result = 0;
                break;
        }
    } else {
        // If the expression has parentheses, compute it separately
        if (expr[0] == '(') {
            // Remove parentheses and evaluate the inner expression
            char subExpr[100];
            strncpy(subExpr, expr + 1, strlen(expr) - 2); // Extract the content within parentheses
            subExpr[strlen(expr) - 2] = '\0';
            result = evaluateExpression(subExpr);
        } else {
            // Handle more complex expressions (if needed)
            result = 0;
        }
    }

    return result;
}

int main() {
    char expr[100];
    // Input the arithmetic expression
    printf("Enter an arithmetic expression: ");
    scanf("%s", expr);

    // Check if the expression is valid
    if (isValidExpression(expr)) {
        printf("Result=%d\n", evaluateExpression(expr));
        printf("Entered arithmetic expression is Valid\n");
    } else {
        printf("Entered arithmetic expression is Invalid\n");
    }

    return 0;
}
```

convert given infix expression to postfix expression

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

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

// Function to check the precedence of operators
int precedence(char op) {
    if (op == '+' || op == '-') {
        return 1;
    }
    if (op == '*' || op == '/' || op == '%') {
        return 2;
    }
    return 0; // Invalid operator
}

// Function to perform infix to postfix conversion
void infixToPostfix(char* infix, char* postfix) {
    char stack[100]; // Stack for operators
    int top = -1; // Stack pointer
    int j = 0; // Index for postfix
    for (int i = 0; i < strlen(infix); i++) {
        char ch = infix[i];

        // If the character is an operand (letter or digit), add it to the postfix expression
        if (isalnum(ch)) {
            postfix[j++] = ch;
        }
        // If the character is '(', push it onto the stack
        else if (ch == '(') {
            stack[++top] = ch;
        }
        // If the character is ')', pop from stack to postfix until '(' is encountered
        else if (ch == ')') {
            while (top != -1 && stack[top] != '(') {
                postfix[j++] = stack[top--];
            }
            top--; // Pop the '(' from the stack
        }
        // If the character is an operator, pop operators
with higher or equal precedence to postfix
        else if (ch == '+' || ch == '-' || ch == '*' || ch == '/'
|| ch == '%') {
            while (top != -1 && precedence(stack[top]) >=
precedence(ch)) {
                postfix[j++] = stack[top--];
            }
            stack[++top] = ch; // Push the current operator
onto the stack
        }
    }

    // Pop all remaining operators from the stack
    while (top != -1) {
        postfix[j++] = stack[top--];
    }

    postfix[j] = '\0'; // Null-terminate the postfix
expression
}
```

```
int main() {
    char infix[100], postfix[100];

    // Input the infix expression
    printf("Enter an infix expression: ");
    scanf("%s", infix);

    // Convert the infix expression to postfix
    infixToPostfix(infix, postfix);

    // Output the postfix expression
    printf("Postfix expression: %s\n", postfix);

    return 0;
}
```

REGEX

```
#include <iostream>
#include <regex>

using namespace std;

int main() {
    string regexPattern, inputString;

    // Prompt for the regular expression
    cout << "Enter a regular expression: ";
    cin >> regexPattern;

    // Prompt for the string to match
    cout << "Enter the input string: ";
    cin >> inputString;

    try {
        // Compile the regular expression
        regex pattern(regexPattern);

        // Check if the input string matches the pattern
        if (regex_match(inputString, pattern)) {
            cout << "The input string matches the regular expression." << endl;
        } else {
            cout << "The input string does NOT match the regular expression." << endl;
        }
    } catch (const regex_error& e) {
        // Handle errors in the regex pattern
        cout << "Invalid regular expression: " << e.what() << endl;
    }

    return 0;
}
```

THREE ADDRESS CODE

```
#include <iostream>
#include <vector>
#include <string>
#include <sstream>
using namespace std;
struct ThreeAddressCode
{
string result;
string arg1;
string op;
string arg2;
};
vector<string> convertToAssembly(const vector<ThreeAddressCode> &tac)
{
vector<string> assemblyCode;
for (const auto &instr : tac)
{
if (instr.op == "=")
{
```

```
assemblyCode.push_back("MOV " + instr.result + ", " +
instr.arg1);
}
else if (instr.op == "+")
{
assemblyCode.push_back("MOV R0, " + instr.arg1);
assemblyCode.push_back("ADD R0, " + instr.arg2);
assemblyCode.push_back("MOV " + instr.result + ",
R0");
}
else if (instr.op == "-")
{
assemblyCode.push_back("MOV R0, " + instr.arg1);
assemblyCode.push_back("SUB R0, " + instr.arg2);
assemblyCode.push_back("MOV " + instr.result + ",
R0");
}
else if (instr.op == "*")
{
assemblyCode.push_back("MOV R0, " + instr.arg1);
assemblyCode.push_back("MUL R0, " + instr.arg2);
assemblyCode.push_back("MOV " + instr.result + ",
R0");
}
else if (instr.op == "/")
{
assemblyCode.push_back("MOV R0, " + instr.arg1);
assemblyCode.push_back("DIV R0, " + instr.arg2);
assemblyCode.push_back("MOV " + instr.result + ",
R0");
}
}
return assemblyCode;
}
int main()
{
vector<ThreeAddressCode> tac = {
{"t1", "a", "+", "b"},
{"t2", "t1", "/", "d"},
{"t3", "t2", "*", "e"},
{"a", "t3"}};
vector<string> assembly = convertToAssembly(tac);
```

```
cout << "Assembly Code:" << endl;
for (const auto &line : assembly)
{
cout << line << endl;
}
return 0;
}
```

Common subexpression elimination:

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <string>
using namespace std;
struct Expression {
string result;
string operand1;
string op;
string operand2;
// Overloading equality for comparison in unordered_map
```

```
bool operator==(const Expression& other) const {
return operand1 == other.operand1 && op == other.op
&& operand2 == other.operand2;
}
};
// Hash function for Expression struct
struct ExpressionHash {
size_t operator()(const Expression& expr) const {
return hash<string>()(expr.operand1) ^
hash<string>()(expr.op) ^
hash<string>()(expr.operand2);
}
};
// Function to perform common subexpression
elimination
void
eliminateCommonSubexpressions(vector<Expression>
& expressions) {
unordered_map<Expression, string, ExpressionHash>
subexpressionMap;
int tempVarCount = 1;
for (auto& expr : expressions) {
```

```
if (subexpressionMap.find(expr) !=
subexpressionMap.end()) {
// If the subexpression is found, replace result with
existing temp variable
cout << expr.result << " = " << subexpressionMap[expr]
<< " // Reusing " <<
subexpressionMap[expr] << endl;
} else {
// Otherwise, add the subexpression to the map and
generate a new temp variable
string tempVar = "t" + to_string(tempVarCount++);
subexpressionMap[expr] = tempVar;
cout << tempVar << " = " << expr.operand1 << " " <<
expr.op << " " << expr.operand2 << endl;
cout << expr.result << " = " << tempVar << endl;
}
}
}
int main() {
// Example expressions
vector<Expression> expressions = {
```

```
{"a", "x", "+", "y"},
{"b", "x", "+", "y"},
{"c", "x", "+", "y"},
{"d", "x", "+", "y"},
{"e", "c", "+", "b"}
};
eliminateCommonSubexpressions(expressions);
return 0;
}
```

22BCE0706

Chirayu Trivedi

Compiler Design

Digital Assignment – 6

Code Optimization

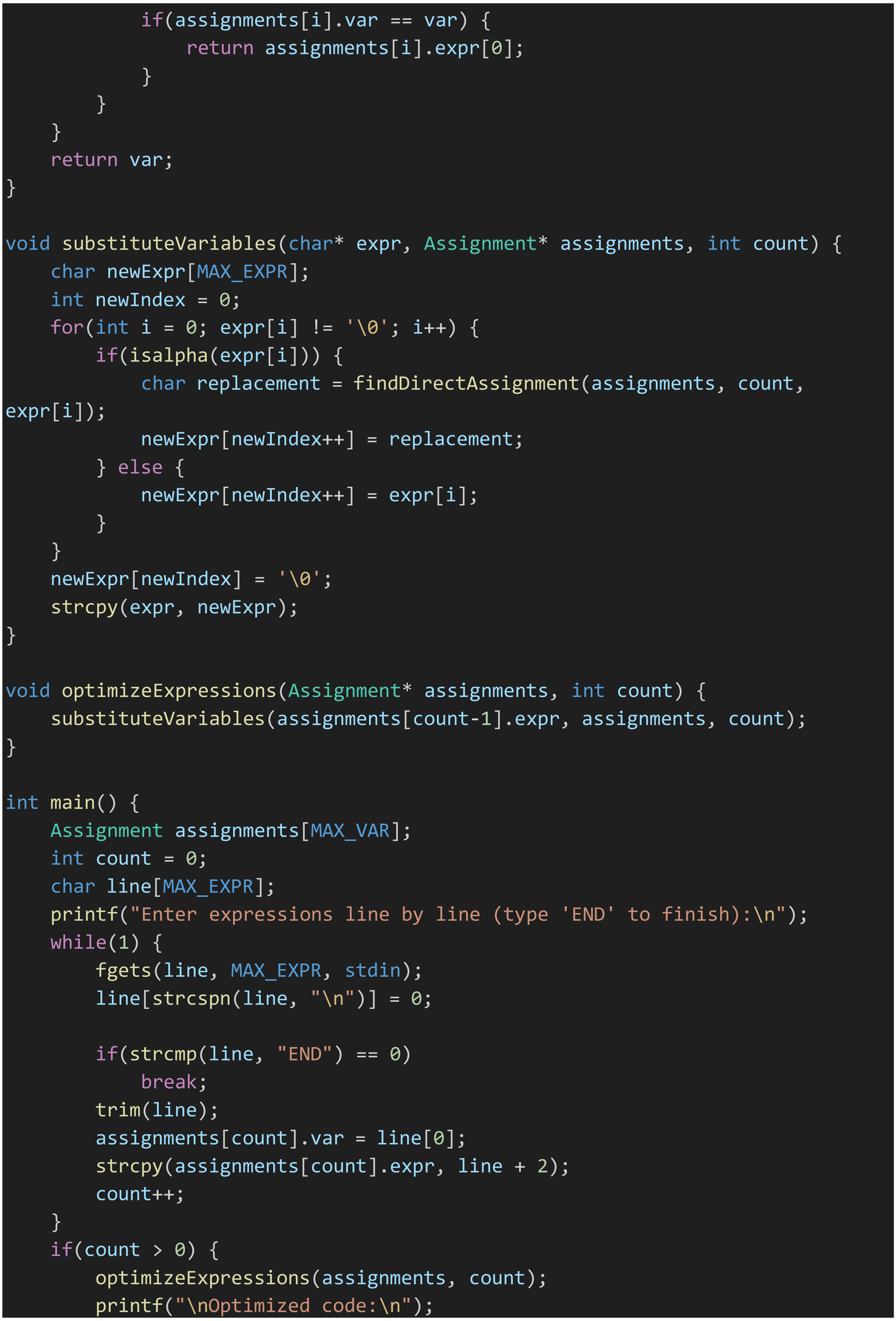
To write a C program for implementation of Code Optimization Technique.

Sample Input

```
//Before Optimization
     c = a * b
     x = a
     d = x * b + 4


     //After Optimization
     d = a * b + 4
```

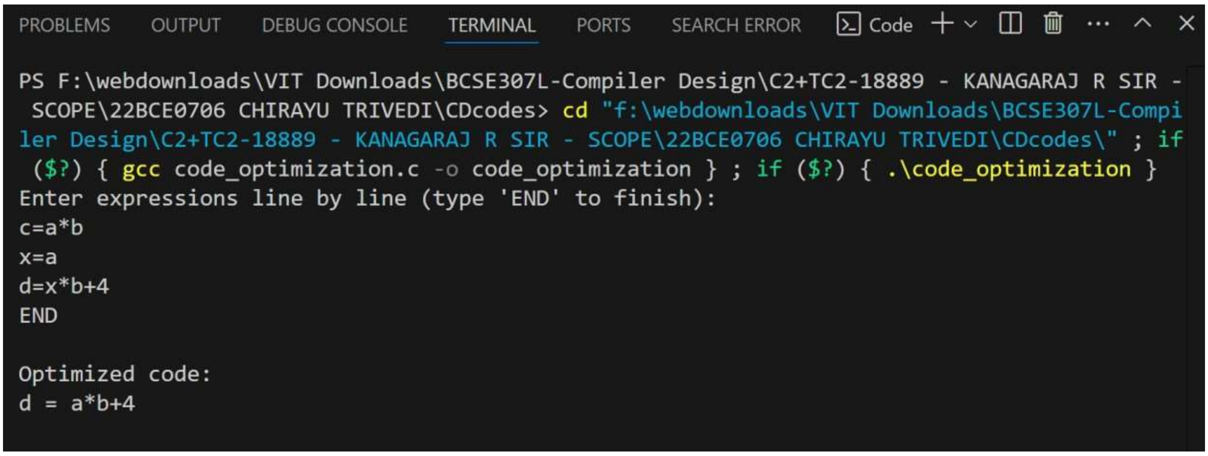
Code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#define MAX_EXPR 100
#define MAX_VAR 26

typedef struct {
    char var;
    char expr[MAX_EXPR];
} Assignment;

void trim(char* str) {
    int i, j = 0;
    for(i = 0; str[i] != '\0'; i++) {
        if(str[i] != ' ' && str[i] != '\t') {
            str[j] = str[i];
            j++;
        }
    }
    str[j] = '\0';
}

char findDirectAssignment(Assignment* assignments, int count, char var) {
    for(int i = 0; i < count; i++) {
        if(strlen(assignments[i].expr) == 1 && assignments[i].expr[0] != var)
{
```



```c
            if(assignments[i].var == var) {
                return assignments[i].expr[0];
            }
        }
    }
    return var;
}

void substituteVariables(char* expr, Assignment* assignments, int count) {
    char newExpr[MAX_EXPR];
    int newIndex = 0;
    for(int i = 0; expr[i] != '\0'; i++) {
        if(isalpha(expr[i])) {
            char replacement = findDirectAssignment(assignments, count,
expr[i]);
            newExpr[newIndex++] = replacement;
        } else {
            newExpr[newIndex++] = expr[i];
        }
    }
    newExpr[newIndex] = '\0';
    strcpy(expr, newExpr);
}

void optimizeExpressions(Assignment* assignments, int count) {
    substituteVariables(assignments[count-1].expr, assignments, count);
}

int main() {
    Assignment assignments[MAX_VAR];
    int count = 0;
    char line[MAX_EXPR];
    printf("Enter expressions line by line (type 'END' to finish):\n");
    while(1) {
        fgets(line, MAX_EXPR, stdin);
        line[strcspn(line, "\n")] = 0;

        if(strcmp(line, "END") == 0)
            break;
        trim(line);
        assignments[count].var = line[0];
        strcpy(assignments[count].expr, line + 2);
        count++;
    }
    if(count > 0) {
        optimizeExpressions(assignments, count);
        printf("\nOptimized code:\n");
        printf("%c = %s\n\n", assignments[count-1].var, assignments[count-
1].expr);
    }
    return 0;
}
```

Output:



```
PS F:\webdownloads\VIT Downloads\BCSE307L-Compiler Design\C2+TC2-18889 - KANAGARAJ R SIR -
 SCOPE\22BCE0706 CHIRAYU TRIVEDI\CDcodes> cd "f:\webdownloads\VIT Downloads\BCSE307L-Compi
ler Design\C2+TC2-18889 - KANAGARAJ R SIR - SCOPE\22BCE0706 CHIRAYU TRIVEDI\CDcodes\" ; if
 ($?) { gcc code_optimization.c -o code_optimization } ; if ($?) { .\code_optimization }
Enter expressions line by line (type 'END' to finish):
c=a*b
x=a
d=x*b+4
END

Optimized code:
d = a*b+4
```

Code Generation

Write a C Program to implement to code generation in Compiler.

for ex, sample input:

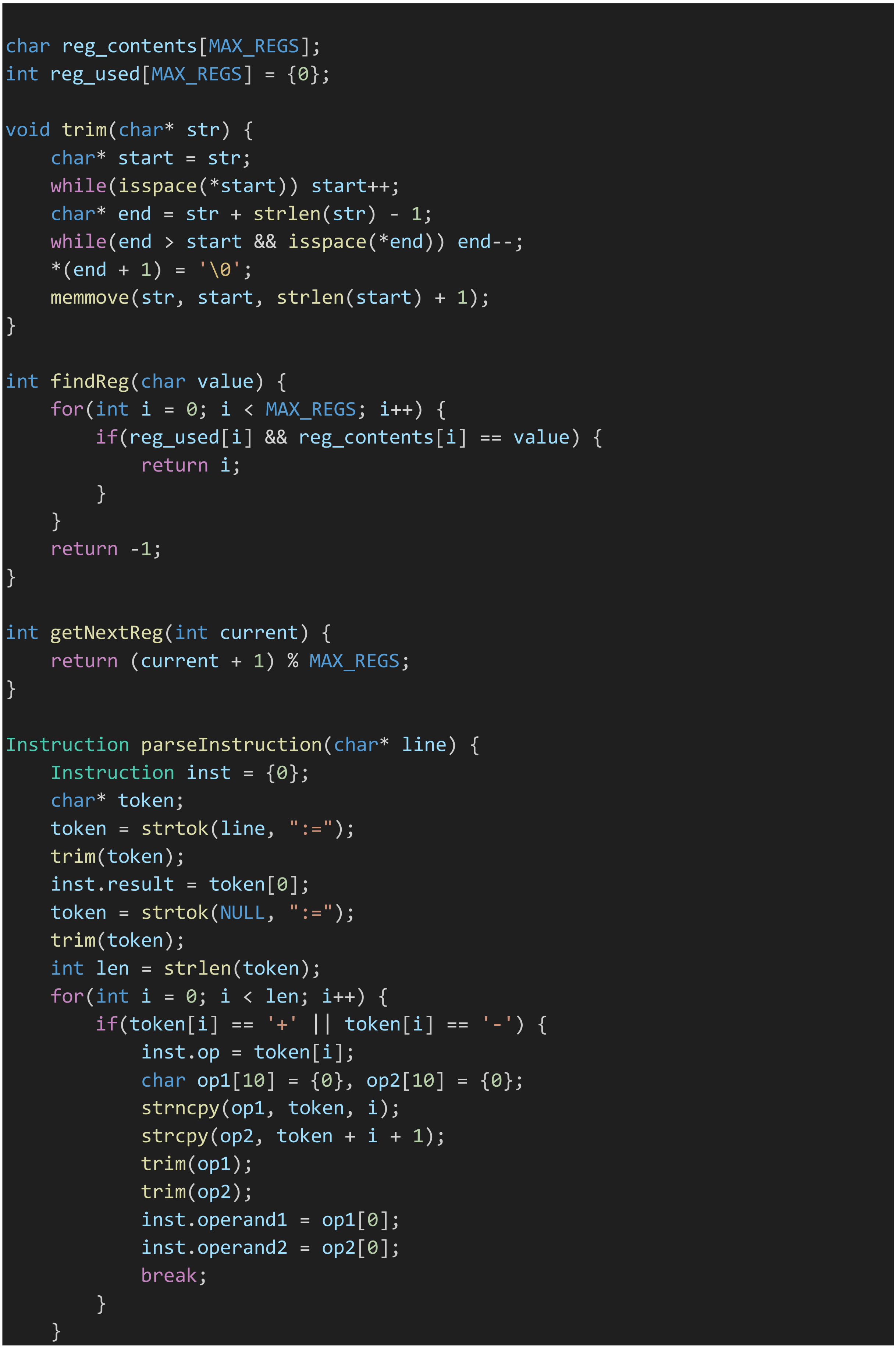
1. t:= a-b
2. u:= a-c
3. v:= t +u
4. d:= v+u

Output:

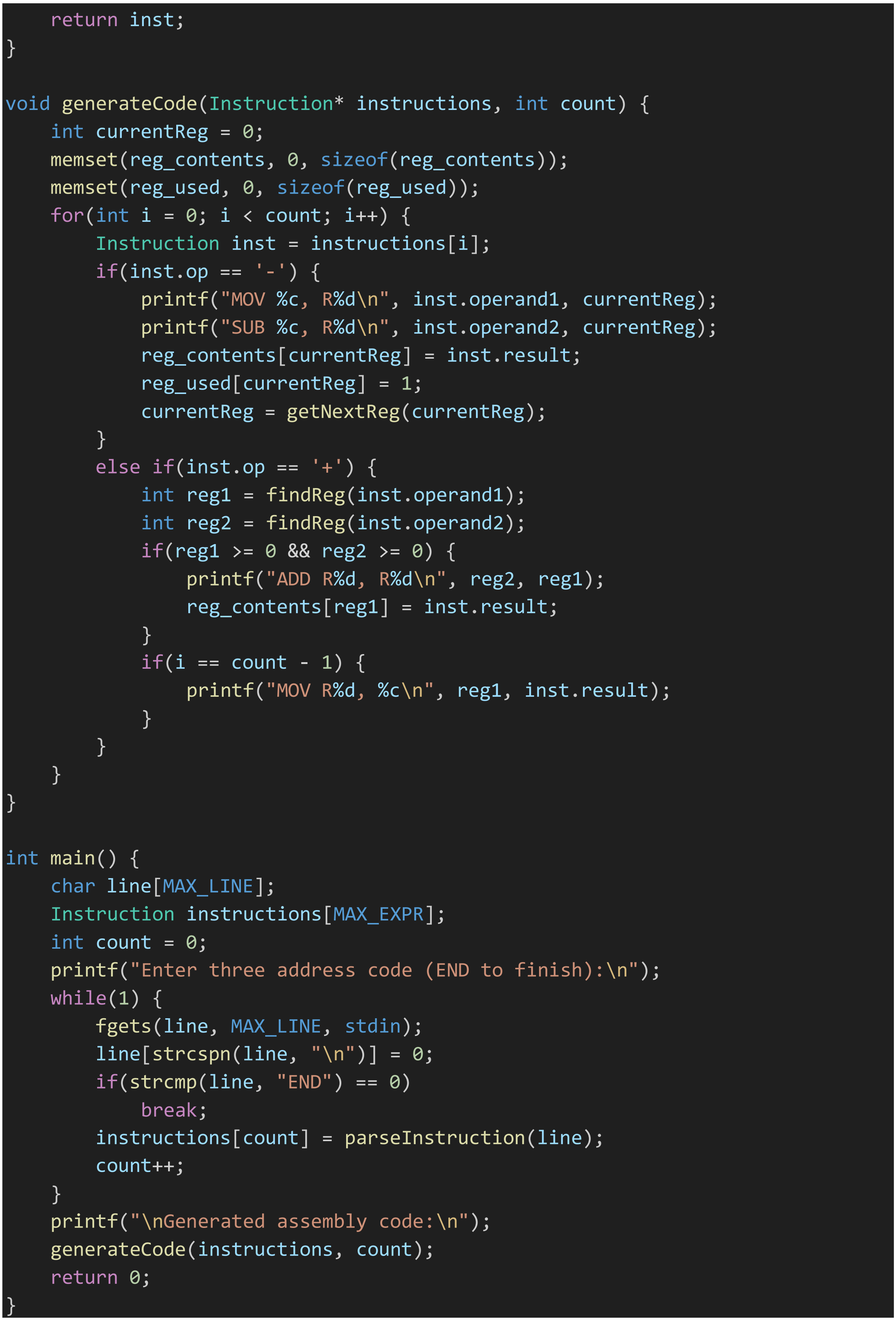
```
MOV a, R0
SUB b, R0

MOV a, R1
SUB c, R1

ADD R1, R0

ADD R1, R0
MOV R0, d
```

Code:

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

#define MAX_LINE 100
#define MAX_EXPR 100
#define MAX_REGS 2

typedef struct {
    char result;
    char op;
    char operand1;
    char operand2;
} Instruction;
```

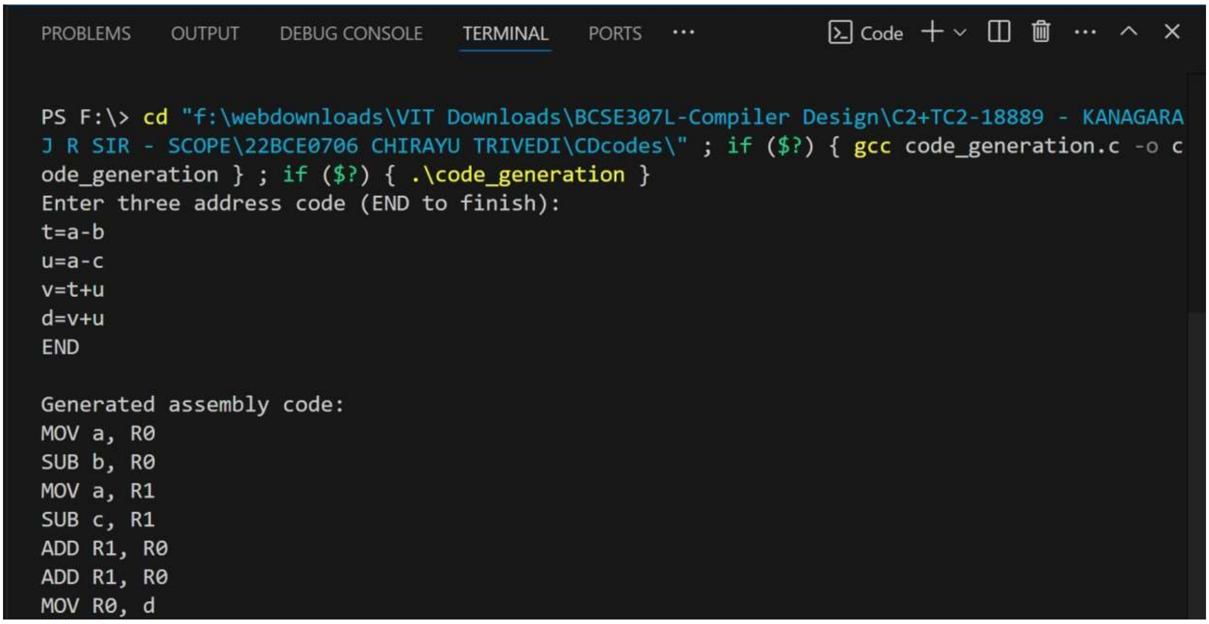


```c
char reg_contents[MAX_REGS];
int reg_used[MAX_REGS] = {0};

void trim(char* str) {
    char* start = str;
    while(isspace(*start)) start++;
    char* end = str + strlen(str) - 1;
    while(end > start && isspace(*end)) end--;
    *(end + 1) = '\0';
    memmove(str, start, strlen(start) + 1);
}

int findReg(char value) {
    for(int i = 0; i < MAX_REGS; i++) {
        if(reg_used[i] && reg_contents[i] == value) {
            return i;
        }
    }
    return -1;
}

int getNextReg(int current) {
    return (current + 1) % MAX_REGS;
}

Instruction parseInstruction(char* line) {
    Instruction inst = {0};
    char* token;
    token = strtok(line, ":=");
    trim(token);
    inst.result = token[0];
    token = strtok(NULL, ":=");
    trim(token);
    int len = strlen(token);
    for(int i = 0; i < len; i++) {
        if(token[i] == '+' || token[i] == '-') {
            inst.op = token[i];
            char op1[10] = {0}, op2[10] = {0};
            strncpy(op1, token, i);
            strcpy(op2, token + i + 1);
            trim(op1);
            trim(op2);
            inst.operand1 = op1[0];
            inst.operand2 = op2[0];
            break;
        }
    }
```



```c
    return inst;
}

void generateCode(Instruction* instructions, int count) {
    int currentReg = 0;
    memset(reg_contents, 0, sizeof(reg_contents));
    memset(reg_used, 0, sizeof(reg_used));
    for(int i = 0; i < count; i++) {
        Instruction inst = instructions[i];
        if(inst.op == '-') {
            printf("MOV %c, R%d\n", inst.operand1, currentReg);
            printf("SUB %c, R%d\n", inst.operand2, currentReg);
            reg_contents[currentReg] = inst.result;
            reg_used[currentReg] = 1;
            currentReg = getNextReg(currentReg);
        }
        else if(inst.op == '+') {
            int reg1 = findReg(inst.operand1);
            int reg2 = findReg(inst.operand2);
            if(reg1 >= 0 && reg2 >= 0) {
                printf("ADD R%d, R%d\n", reg2, reg1);
                reg_contents[reg1] = inst.result;
            }
            if(i == count - 1) {
                printf("MOV R%d, %c\n", reg1, inst.result);
            }
        }
    }
}

int main() {
    char line[MAX_LINE];
    Instruction instructions[MAX_EXPR];
    int count = 0;
    printf("Enter three address code (END to finish):\n");
    while(1) {
        fgets(line, MAX_LINE, stdin);
        line[strcspn(line, "\n")] = 0;
        if(strcmp(line, "END") == 0)
            break;
        instructions[count] = parseInstruction(line);
        count++;
    }
    printf("\nGenerated assembly code:\n");
    generateCode(instructions, count);
    return 0;
}
```

Output:



```
PS F:\> cd "f:\webdownloads\VIT Downloads\BCSE307L-Compiler Design\C2+TC2-18889 - KANAGARA
J R SIR - SCOPE\22BCE0706 CHIRAYU TRIVEDI\CDcodes\" ; if ($?) { gcc code_generation.c -o c
ode_generation } ; if ($?) { .\code_generation }
Enter three address code (END to finish):
t=a-b
u=a-c
v=t+u
d=v+u
END

Generated assembly code:
MOV a, R0
SUB b, R0
MOV a, R1
SUB c, R1
ADD R1, R0
ADD R1, R0
MOV R0, d
```

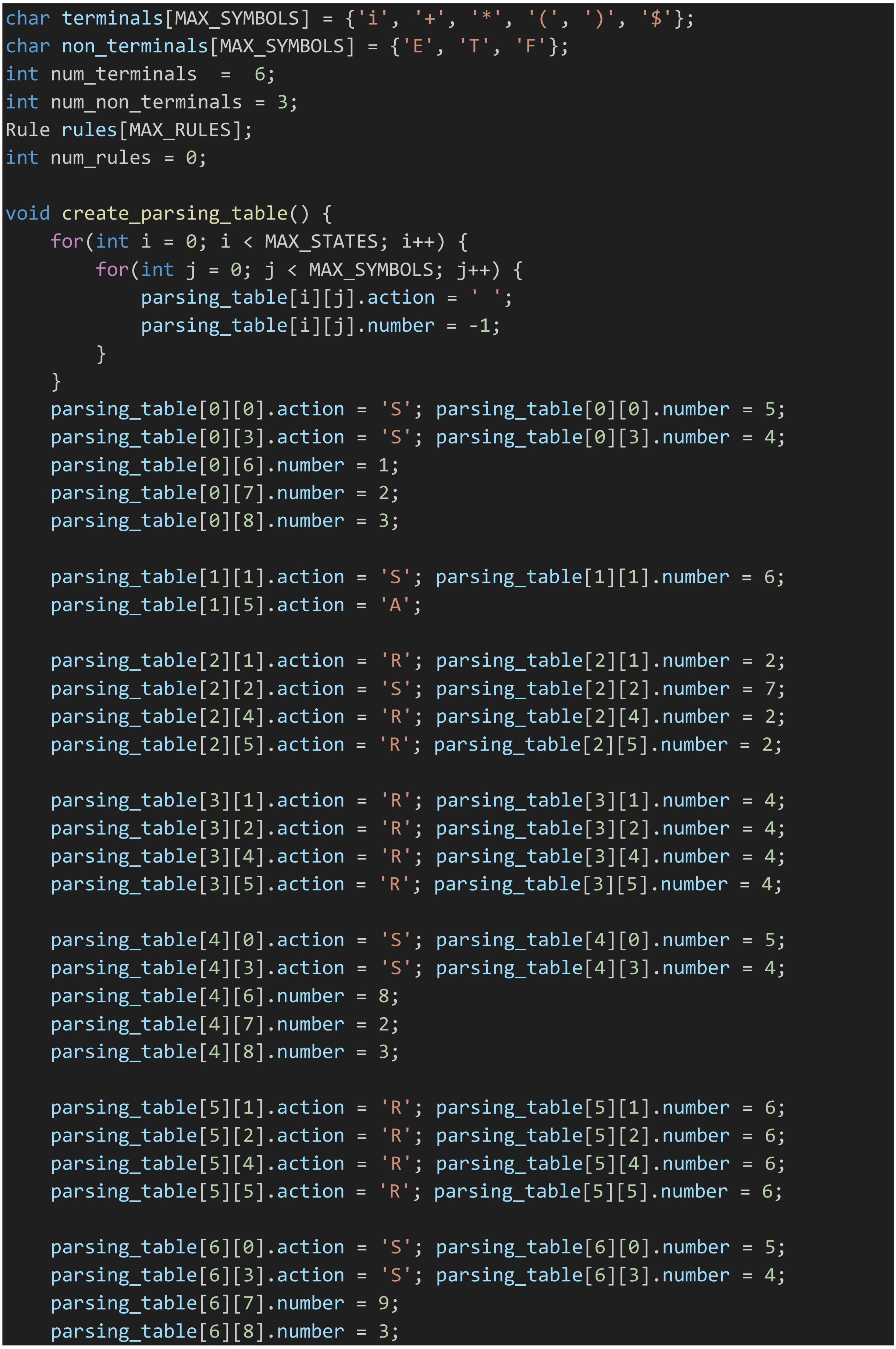
SLR Parser

Write a Program to implement SLR Parser.

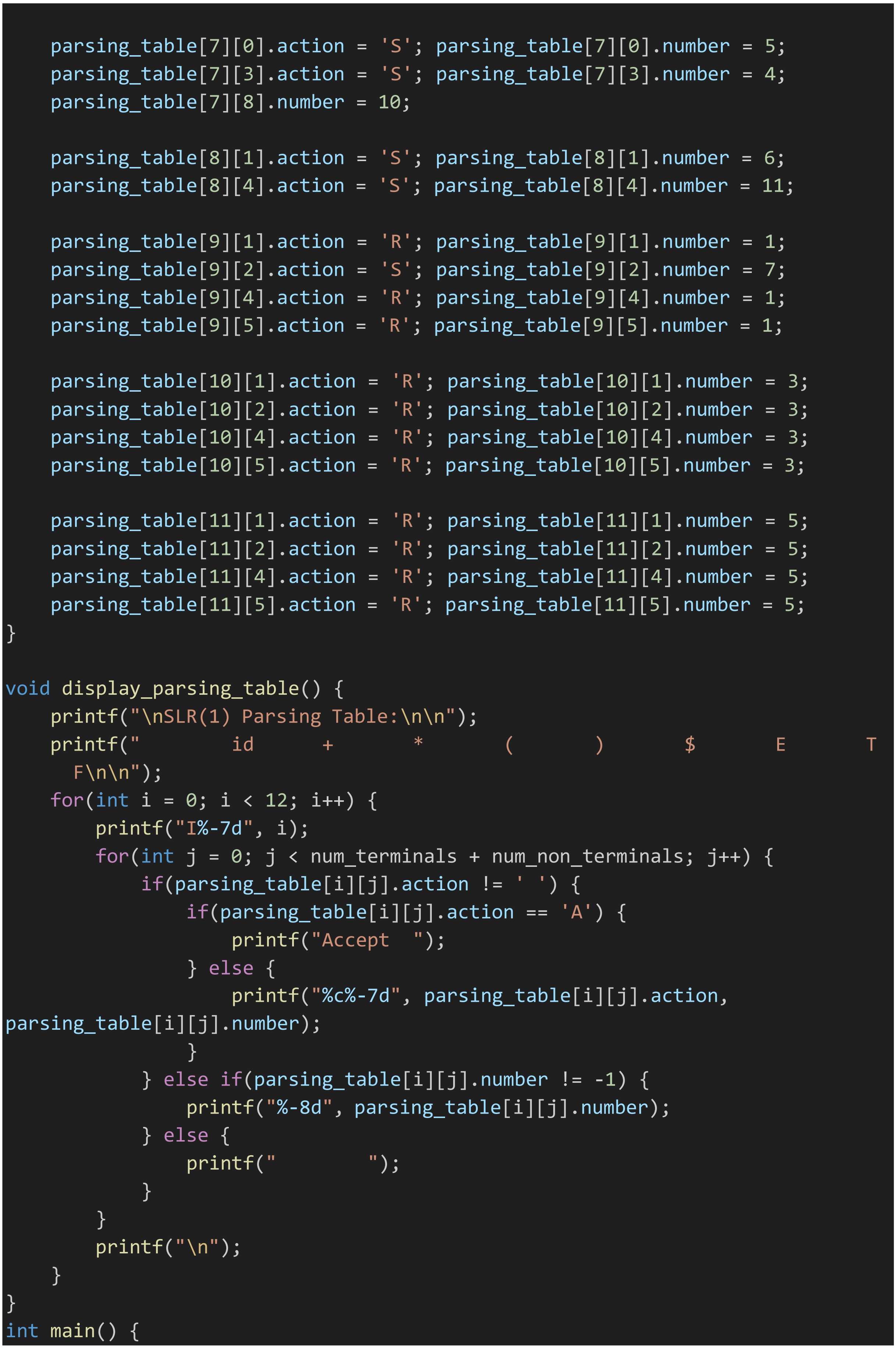
```
Original grammar input:

E -> E + T | T
T -> T * F | F
F -> ( E ) | id

Sample Output

SLR(1) parsing table:

         id       +       *       (       )       $       E       T       F

 I0      S5                      S4                       1       2       3
 I1              S6                          Accept
 I2              R2      S7              R2      R2
 I3              R4      R4              R4      R4
 I4      S5                      S4                       8       2       3
 I5              R6      R6              R6      R6
 I6      S5                      S4                               9       3
 I7      S5                      S4                                      10
 I8              S6                     S11
 I9              R1      S7              R1      R1
I10              R3      R3              R3      R3
I11              R5      R5              R5      R5
```

Code:

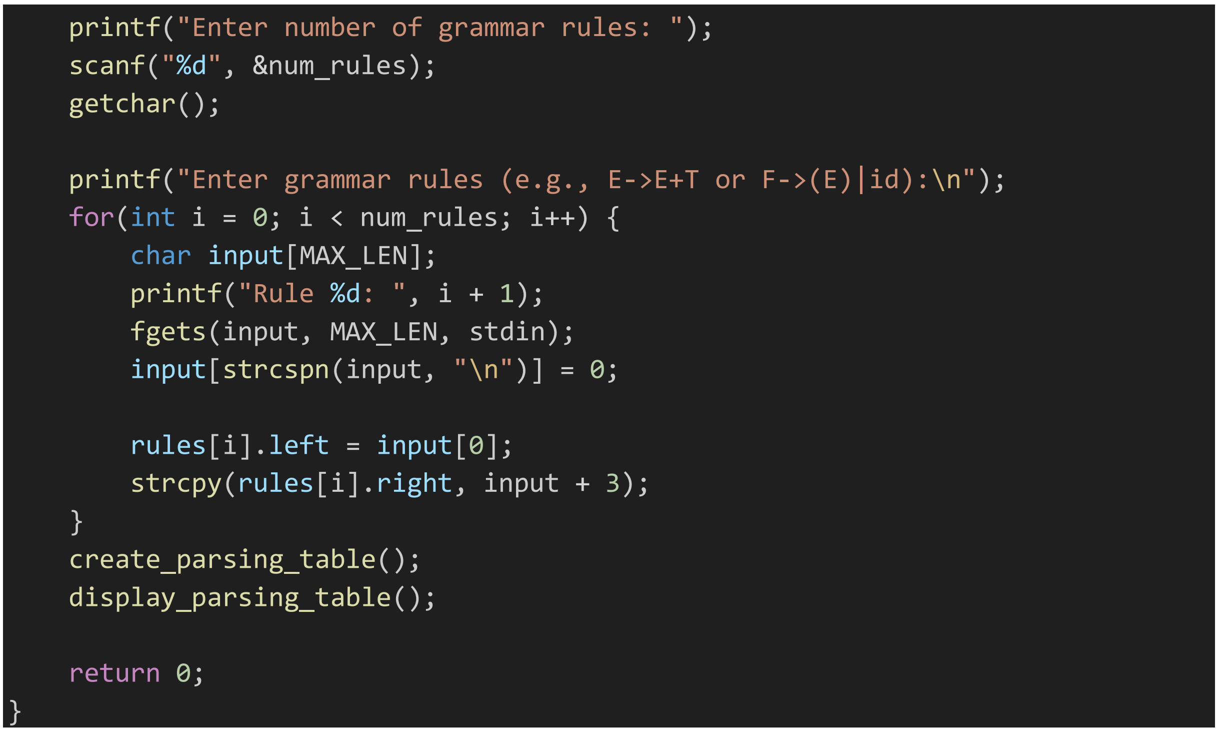
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_RULES 10
#define MAX_STATES 20
#define MAX_SYMBOLS 20
#define MAX_LEN 50

typedef struct {
    char left;
    char right[MAX_LEN];
} Rule;

typedef struct {
    char action;
    int number;
} TableEntry;
TableEntry parsing_table[MAX_STATES][MAX_SYMBOLS];
```



```
char terminals[MAX_SYMBOLS] = {'i', '+', '*', '(', ')', '$'};
char non_terminals[MAX_SYMBOLS] = {'E', 'T', 'F'};
int num_terminals  =  6;
int num_non_terminals = 3;
Rule rules[MAX_RULES];
int num_rules = 0;

void create_parsing_table() {
    for(int i = 0; i < MAX_STATES; i++) {
        for(int j = 0; j < MAX_SYMBOLS; j++) {
            parsing_table[i][j].action = ' ';
            parsing_table[i][j].number = -1;
        }
    }
    parsing_table[0][0].action = 'S'; parsing_table[0][0].number = 5;
    parsing_table[0][3].action = 'S'; parsing_table[0][3].number = 4;
    parsing_table[0][6].number = 1;
    parsing_table[0][7].number = 2;
    parsing_table[0][8].number = 3;

    parsing_table[1][1].action = 'S'; parsing_table[1][1].number = 6;
    parsing_table[1][5].action = 'A';

    parsing_table[2][1].action = 'R'; parsing_table[2][1].number = 2;
    parsing_table[2][2].action = 'S'; parsing_table[2][2].number = 7;
    parsing_table[2][4].action = 'R'; parsing_table[2][4].number = 2;
    parsing_table[2][5].action = 'R'; parsing_table[2][5].number = 2;

    parsing_table[3][1].action = 'R'; parsing_table[3][1].number = 4;
    parsing_table[3][2].action = 'R'; parsing_table[3][2].number = 4;
    parsing_table[3][4].action = 'R'; parsing_table[3][4].number = 4;
    parsing_table[3][5].action = 'R'; parsing_table[3][5].number = 4;

    parsing_table[4][0].action = 'S'; parsing_table[4][0].number = 5;
    parsing_table[4][3].action = 'S'; parsing_table[4][3].number = 4;
    parsing_table[4][6].number = 8;
    parsing_table[4][7].number = 2;
    parsing_table[4][8].number = 3;

    parsing_table[5][1].action = 'R'; parsing_table[5][1].number = 6;
    parsing_table[5][2].action = 'R'; parsing_table[5][2].number = 6;
    parsing_table[5][4].action = 'R'; parsing_table[5][4].number = 6;
    parsing_table[5][5].action = 'R'; parsing_table[5][5].number = 6;

    parsing_table[6][0].action = 'S'; parsing_table[6][0].number = 5;
    parsing_table[6][3].action = 'S'; parsing_table[6][3].number = 4;
    parsing_table[6][7].number = 9;
    parsing_table[6][8].number = 3;
```

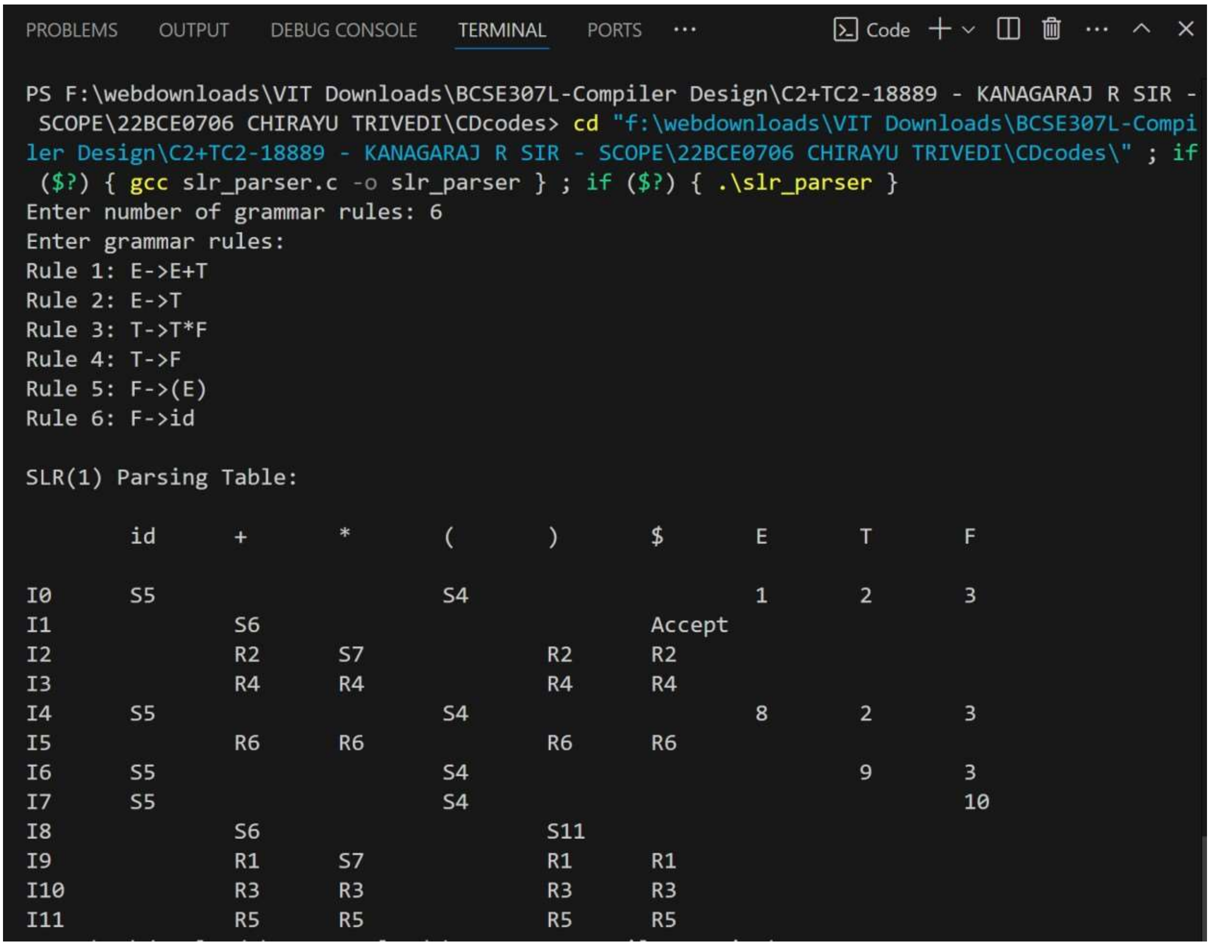


```
    parsing_table[7][0].action = 'S'; parsing_table[7][0].number = 5;
    parsing_table[7][3].action = 'S'; parsing_table[7][3].number = 4;
    parsing_table[7][8].number = 10;

    parsing_table[8][1].action = 'S'; parsing_table[8][1].number = 6;
    parsing_table[8][4].action = 'S'; parsing_table[8][4].number = 11;

    parsing_table[9][1].action = 'R'; parsing_table[9][1].number = 1;
    parsing_table[9][2].action = 'S'; parsing_table[9][2].number = 7;
    parsing_table[9][4].action = 'R'; parsing_table[9][4].number = 1;
    parsing_table[9][5].action = 'R'; parsing_table[9][5].number = 1;

    parsing_table[10][1].action = 'R'; parsing_table[10][1].number = 3;
    parsing_table[10][2].action = 'R'; parsing_table[10][2].number = 3;
    parsing_table[10][4].action = 'R'; parsing_table[10][4].number = 3;
    parsing_table[10][5].action = 'R'; parsing_table[10][5].number = 3;

    parsing_table[11][1].action = 'R'; parsing_table[11][1].number = 5;
    parsing_table[11][2].action = 'R'; parsing_table[11][2].number = 5;
    parsing_table[11][4].action = 'R'; parsing_table[11][4].number = 5;
    parsing_table[11][5].action = 'R'; parsing_table[11][5].number = 5;
}

void display_parsing_table() {
    printf("\nSLR(1) Parsing Table:\n\n");
    printf("        id      +       *       (       )       $       E       T
      F\n\n");
    for(int i = 0; i < 12; i++) {
        printf("I%-7d", i);
        for(int j = 0; j < num_terminals + num_non_terminals; j++) {
            if(parsing_table[i][j].action != ' ') {
                if(parsing_table[i][j].action == 'A') {
                    printf("Accept  ");
                } else {
                    printf("%c%-7d", parsing_table[i][j].action,
parsing_table[i][j].number);
                }
            } else if(parsing_table[i][j].number != -1) {
                printf("%-8d", parsing_table[i][j].number);
            } else {
                printf("        ");
            }
        }
        printf("\n");
    }
}
int main() {
```



```
    printf("Enter number of grammar rules: ");
    scanf("%d", &num_rules);
    getchar();

    printf("Enter grammar rules (e.g., E->E+T or F->(E)|id):\n");
    for(int i = 0; i < num_rules; i++) {
        char input[MAX_LEN];
        printf("Rule %d: ", i + 1);
        fgets(input, MAX_LEN, stdin);
        input[strcspn(input, "\n")] = 0;

        rules[i].left = input[0];
        strcpy(rules[i].right, input + 3);
    }
    create_parsing_table();
    display_parsing_table();

    return 0;
}
```

Output:



```
PS F:\webdownloads\VIT Downloads\BCSE307L-Compiler Design\C2+TC2-18889 - KANAGARAJ R SIR -
 SCOPE\22BCE0706 CHIRAYU TRIVEDI\CDcodes> cd "f:\webdownloads\VIT Downloads\BCSE307L-Compi
ler Design\C2+TC2-18889 - KANAGARAJ R SIR - SCOPE\22BCE0706 CHIRAYU TRIVEDI\CDcodes\" ; if
 ($?) { gcc slr_parser.c -o slr_parser } ; if ($?) { .\slr_parser }
Enter number of grammar rules: 6
Enter grammar rules:
Rule 1: E->E+T
Rule 2: E->T
Rule 3: T->T*F
Rule 4: T->F
Rule 5: F->(E)
Rule 6: F->id

SLR(1) Parsing Table:

        id      +       *       (       )       $       E       T       F

I0      S5                      S4                      1       2       3
I1              S6                              Accept
I2              R2      S7              R2      R2
I3              R4      R4              R4      R4
I4      S5                      S4                      8       2       3
I5              R6      R6              R6      R6
I6      S5                      S4                              9       3
I7      S5                      S4                                      10
I8              S6                      S11
I9              R1      S7              R1      R1
I10             R3      R3              R3      R3
I11             R5      R5              R5      R5
```

Left Recursion

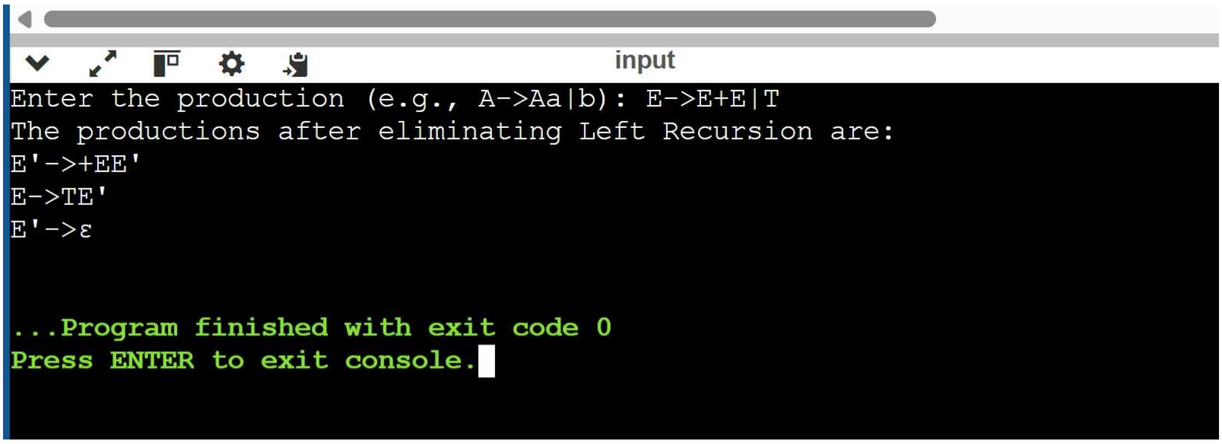
Write a C program to implement Left recursion.

```
Enter the productions: E->E+E|T
The productions after eliminating Left Recursion are:
E->+EE'
E'->TE'
E->ε
```

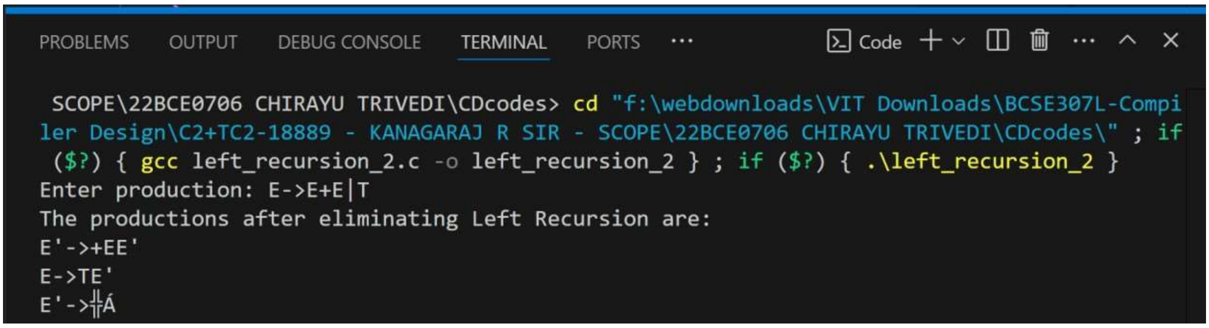
Code:

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

void main() {
    char input[100], lhs[50], rhs[50], temp[50], newProduction[25][50],
newSymbol[55];
    int i = 0, j, flag = 0;

    printf("Enter production: ");
    scanf("%s", input);

    sscanf(input, "%[^->]->%s", lhs, rhs);

    snprintf(newSymbol, sizeof(newSymbol), "%s'", lhs);

    char *token = strtok(rhs, "|");

    while (token != NULL) {
        if (token[0] == lhs[0]) {
            flag = 1;
            sprintf(newProduction[i++], "%s->%s%s", newSymbol, token + 1,
newSymbol);
        } else {
            sprintf(newProduction[i++], "%s->%s%s", lhs, token, newSymbol);
        }
        token = strtok(NULL, "|");
    }

    if (flag) {
        sprintf(newProduction[i++], "%s->ε", newSymbol);
        printf("The productions after eliminating Left Recursion are:\n");
        for (j = 0; j < i; j++) {
            printf("%s\n", newProduction[j]);
        }
    } else {
        printf("The given grammar has no Left Recursion.\n");
    }
}
```

Output:



```
Enter the production (e.g., A->Aa|b): E->E+E|T
The productions after eliminating Left Recursion are:
E'->+EE'
E->TE'
E'->ε


...Program finished with exit code 0
Press ENTER to exit console.
```



```
 SCOPE\22BCE0706 CHIRAYU TRIVEDI\CDcodes> cd "f:\webdownloads\VIT Downloads\BCSE307L-Compi
ler Design\C2+TC2-18889 - KANAGARAJ R SIR - SCOPE\22BCE0706 CHIRAYU TRIVEDI\CDcodes\" ; if
 ($?) { gcc left_recursion_2.c -o left_recursion_2 } ; if ($?) { .\left_recursion_2 }
Enter production: E->E+E|T
The productions after eliminating Left Recursion are:
E'->+EE'
E->TE'
E'->╬Á
```

Left Factoring

Write a C program to remove the left factoring from the following Grammer.

Input:

S → iEtS / iEtSeS / a

E → b

Output:

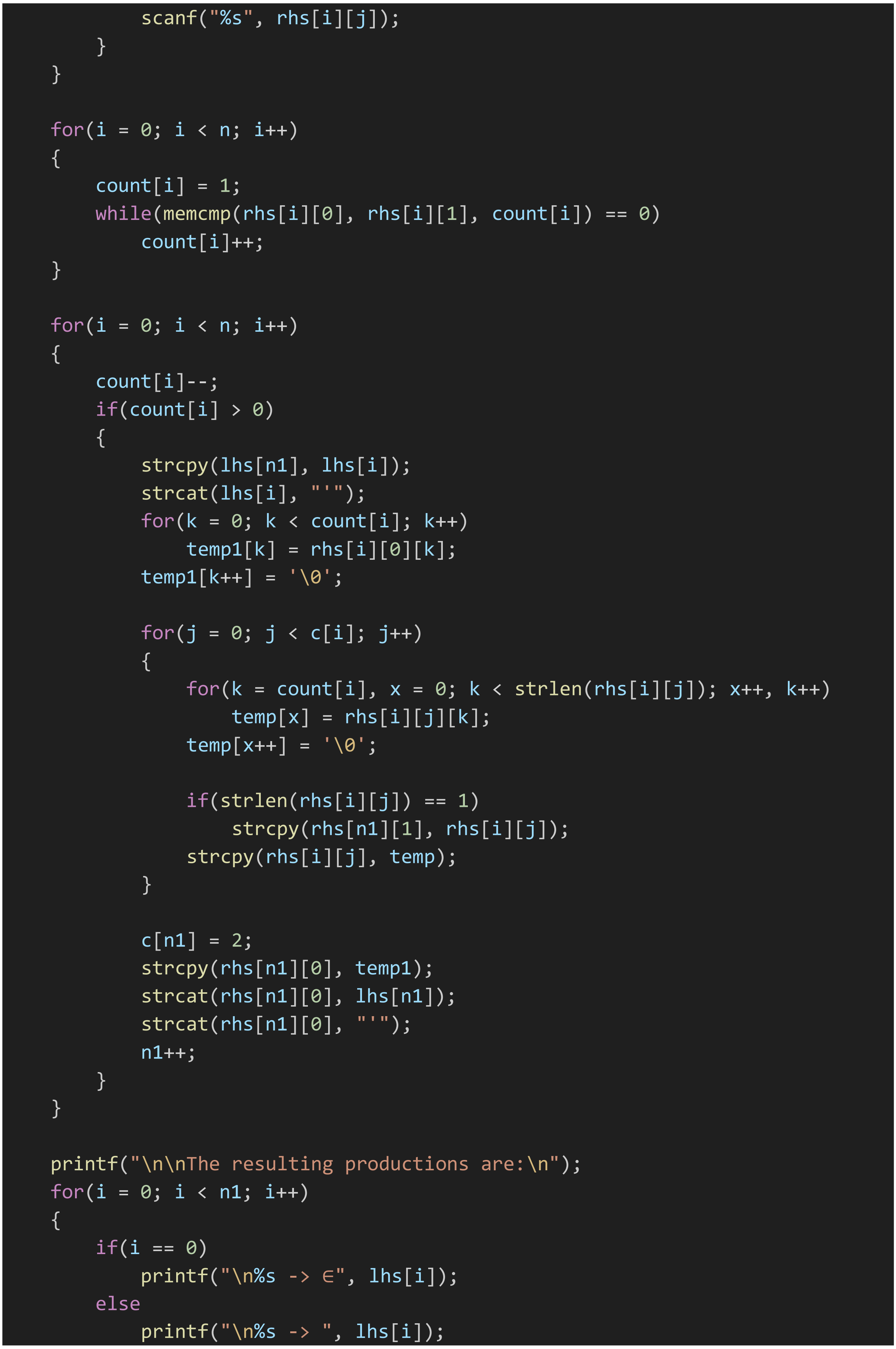
S → iEtSS′ / a

S′ → eS / ∈

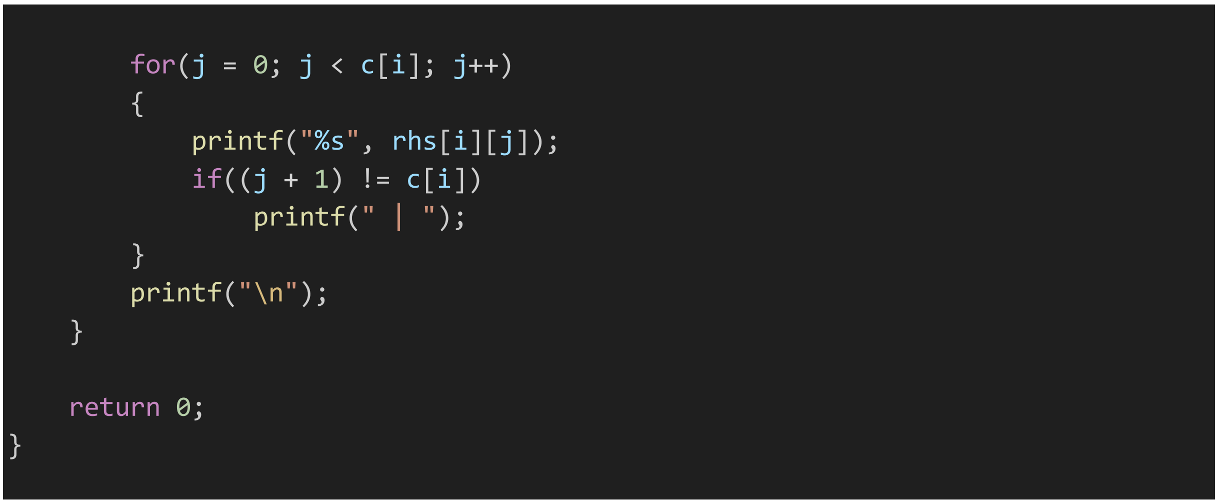
E → b

Code:

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

int main()
{
    char ch, lhs[20][20], rhs[20][20][20], temp[20], temp1[20];
    int n, n1, count[20], x, y, i, j, k, c[20];

    printf("\nEnter the no. of nonterminals: ");
    scanf("%d", &n);
    n1 = n;
    for(i = 0; i < n; i++)
    {
        printf("\nNonterminal %d\nEnter the no. of productions: ", i + 1);
        scanf("%d", &c[i]);

        printf("\nEnter LHS: ");
        scanf("%s", lhs[i]);

        for(j = 0; j < c[i]; j++)
        {
            printf("%s->", lhs[i]);
```

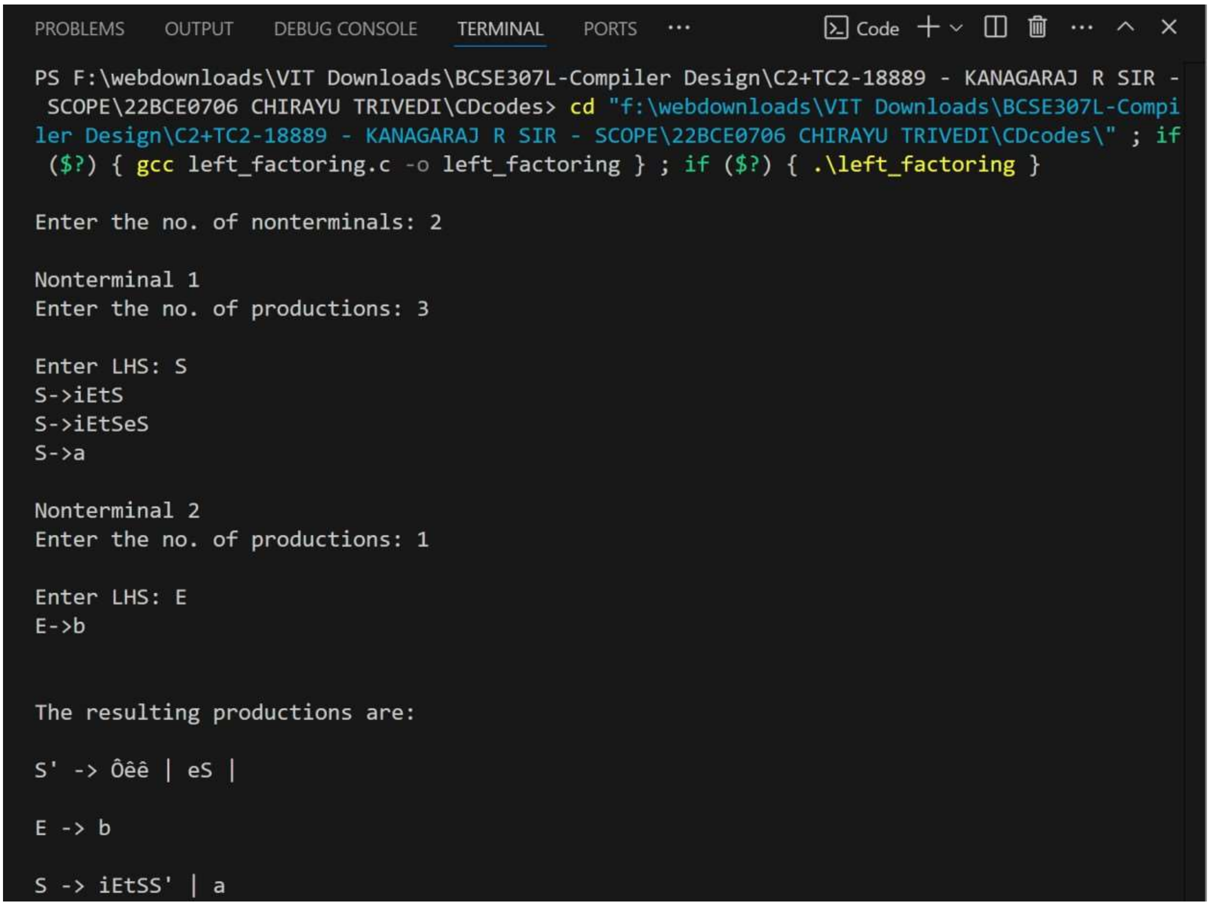


```
            scanf("%s", rhs[i][j]);
        }
    }

    for(i = 0; i < n; i++)
    {
        count[i] = 1;
        while(memcmp(rhs[i][0], rhs[i][1], count[i]) == 0)
            count[i]++;
    }

    for(i = 0; i < n; i++)
    {
        count[i]--;
        if(count[i] > 0)
        {
            strcpy(lhs[n1], lhs[i]);
            strcat(lhs[i], "'");
            for(k = 0; k < count[i]; k++)
                temp1[k] = rhs[i][0][k];
            temp1[k++] = '\0';

            for(j = 0; j < c[i]; j++)
            {
                for(k = count[i], x = 0; k < strlen(rhs[i][j]); x++, k++)
                    temp[x] = rhs[i][j][k];
                temp[x++] = '\0';

                if(strlen(rhs[i][j]) == 1)
                    strcpy(rhs[n1][1], rhs[i][j]);
                strcpy(rhs[i][j], temp);
            }

            c[n1] = 2;
            strcpy(rhs[n1][0], temp1);
            strcat(rhs[n1][0], lhs[n1]);
            strcat(rhs[n1][0], "'");
            n1++;
        }
    }

    printf("\n\nThe resulting productions are:\n");
    for(i = 0; i < n1; i++)
    {
        if(i == 0)
            printf("\n%s -> ∈", lhs[i]);
        else
            printf("\n%s -> ", lhs[i]);
```



```
        for(j = 0; j < c[i]; j++)
        {
            printf("%s", rhs[i][j]);
            if((j + 1) != c[i])
                printf(" | ");
        }
        printf("\n");
    }

    return 0;
}
```

Output:



```
PS F:\webdownloads\VIT Downloads\BCSE307L-Compiler Design\C2+TC2-18889 - KANAGARAJ R SIR -
 SCOPE\22BCE0706 CHIRAYU TRIVEDI\CDcodes> cd "f:\webdownloads\VIT Downloads\BCSE307L-Compi
ler Design\C2+TC2-18889 - KANAGARAJ R SIR - SCOPE\22BCE0706 CHIRAYU TRIVEDI\CDcodes\" ; if
 ($?) { gcc left_factoring.c -o left_factoring } ; if ($?) { .\left_factoring }

Enter the no. of nonterminals: 2

Nonterminal 1
Enter the no. of productions: 3

Enter LHS: S
S->iEtS
S->iEtSeS
S->a

Nonterminal 2
Enter the no. of productions: 1

Enter LHS: E
E->b


The resulting productions are:

S' -> Ôêê | eS |

E -> b

S -> iEtSS' | a
```

Complier Design Digital Assignment

Arvind Balaji A

22BCE0666

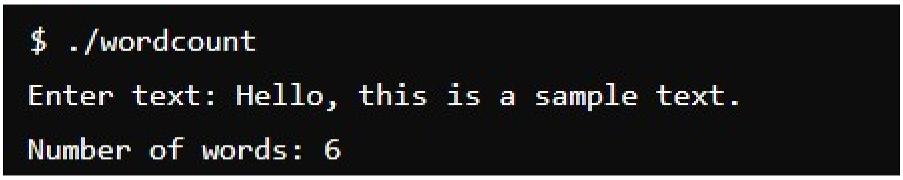
2)

Code:

```
%{
#include <stdio.h>
int word_count = 0;
%}

%%
[ \t\n]+       /* Ignore whitespace (space, tab, newline) */;
[a-zA-Z]+      { word_count++; }  /* Increment word count for each word */

%%
```

```
int main()
{
    printf("Enter text: ");
    yylex();
    printf("Number of words: %d\n", word_count);
    return 0;
}

int yywrap() {
    return 1;
}
```

Output:



```
$ ./wordcount
Enter text: Hello, this is a sample text.
Number of words: 6
```

Complier Design Digital Assignment

Arvind Balaji A

22BCE0666

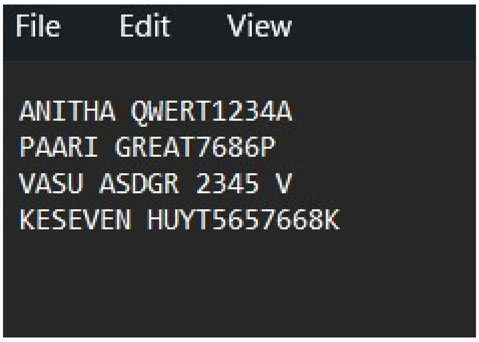
3)

Code:

```
%{
#include <stdio.h>
#include <string.h>
%}

%%

^[A-Z]{5}[0-9]{4}[A-Z]$ {
    // Check for a match with the required pattern
    int valid = 1;
    // Extract the 10th character and first character of
PAN
    char tenth_char = yytext[9];
    char first_char = yytext[0];

    // Ensure 10th character matches the first alphabet
of the PAN card holder name
    if (tenth_char != first_char) {
        valid = 0;
    }

    if (valid) {
        printf("VALID\n");
    } else {
        printf("INVALID\n");
    }
}

.* {
    // For any input that doesn't match the expected
pattern
```

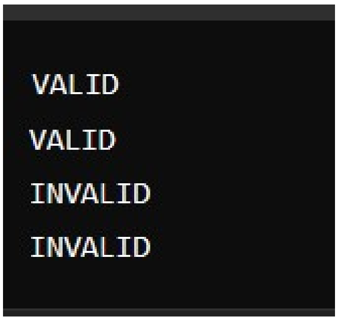
```
    printf("INVALID\n");
}

%%

int main(int argc, char *argv[]) {
    // Check if input file is provided
    if (argc > 1) {
        // Open the file
        FILE *file = fopen(argv[1], "r");
        if (file) {
            // Set input to file
            yyin = file;
            // Run the lexer
            yylex();
            // Close the file
            fclose(file);
        } else {
            fprintf(stderr, "Error opening file.\n");
            return 1;
        }
    } else {
        fprintf(stderr, "No input file provided.\n");
        return 1;
    }

    return 0;
}
```

Input file :



```
ANITHA QWERT1234A
PAARI GREAT7686P
VASU ASDGR 2345 V
KESEVEN HUYT5657668K
```

Output:



```
VALID
VALID
INVALID
INVALID
```

Complier Design Digital Assignment

Arvind Balaji A

22BCE0666

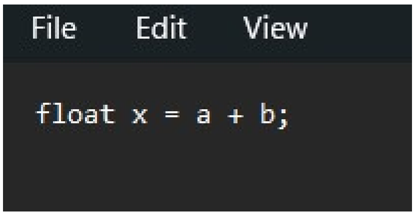
4)

Code:

```
%{
#include <stdio.h>
#include <ctype.h>

// Define tokens
#define KEYWORD 1
#define IDENTIFIER 2
#define OPERATOR 3
#define NUMBER 4
```

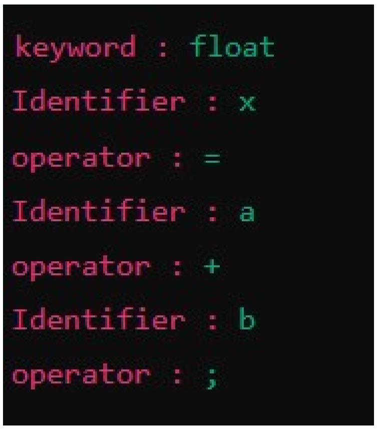
```
void print_token(int token_type, char *token);
%}

%option noyywrap

%%

"float" | "int" | "char"              {
print_token(KEYWORD, yytext); }
"+" | "-" | "*" | "/" | "=" | ";"      {
print_token(OPERATOR, yytext); }
[a-zA-Z_][a-zA-Z0-9_]*               {
print_token(IDENTIFIER, yytext); }
[ \t\n]                          { /* Ignore whitespace */ }
.                              { /* Ignore any unrecognized
character */ }

%%

void print_token(int token_type, char *token) {
    switch(token_type) {
            case KEYWORD:
                printf("keyword : %s\n", token);
                break;
            case IDENTIFIER:
                printf("Identifier : %s\n", token);
                break;
            case OPERATOR:
                printf("operator : %s\n", token);
                break;
            default:
                printf("Unknown token : %s\n", token);
        }
    }

    int main(int argc, char **argv) {
        if (argc > 1) {
            FILE *file = fopen(argv[1], "r");
            if (!file) {
                printf("Could not open file %s\n", argv[1]);
                return 1;
            }
            yyin = file;
        }
        yylex();
        return 0;
    }
```

Input File:



```
float x = a + b;
```

Output:



```
keyword : float
Identifier : x
operator : =
Identifier : a
operator : +
Identifier : b
operator : ;
```

Complier Design Digital Assignment

Arvind Balaji A

22BCE0666

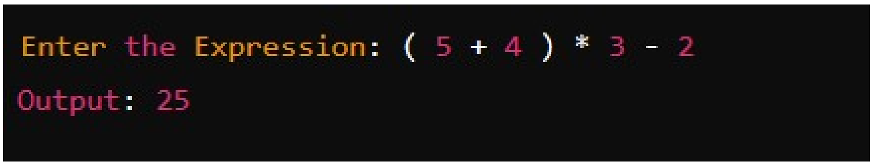
5)

Code:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int evaluate(const char *expression) {
   char command[100];
   snprintf(command, sizeof(command), "echo '%s' | bc", expression);
   FILE *fp = popen(command, "r");
   if (fp == NULL) {
      perror("Error evaluating expression");
      exit(1);
   }

   char result[100];
   fgets(result, sizeof(result), fp);
   pclose(fp);
   return atoi(result);
}
%}

%%
[0-9()+\-*/ \t\n]+ {
    printf("Result: %d\n", evaluate(yytext));
}
. {
    printf("Invalid character: %s\n", yytext);
}
%%

int main() {
    printf("Enter the expression: ");
    yylex();
    return 0;
}
```

OUTPUT

:



```
Enter the Expression: ( 5 + 4 ) * 3 - 2
Output: 25
```

Complier Design Digital Assignment

Arvind Balaji A

22BCE0666

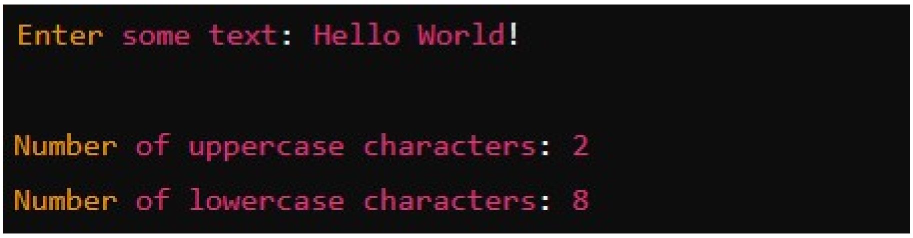
1)

Code:

```
%{
#include <stdio.h>
int uppercase_count = 0;
int lowercase_count = 0;
%}

%%

[A-Z] { uppercase_count++; }
[a-z] { lowercase_count++; }
.     { /* Ignore other characters */ }
```

```
%%

int main() {
    printf("Enter some text: ");
    yylex();  // Start the lexer

    printf("\nNumber of uppercase characters: %d\n", uppercase_count);
    printf("Number of lowercase characters: %d\n", lowercase_count);

    return 0;
}

int yywrap() {
    return 1;
}
```

Output:



```
Enter some text: Hello World!


Number of uppercase characters: 2
Number of lowercase characters: 8
```

File handling code c++

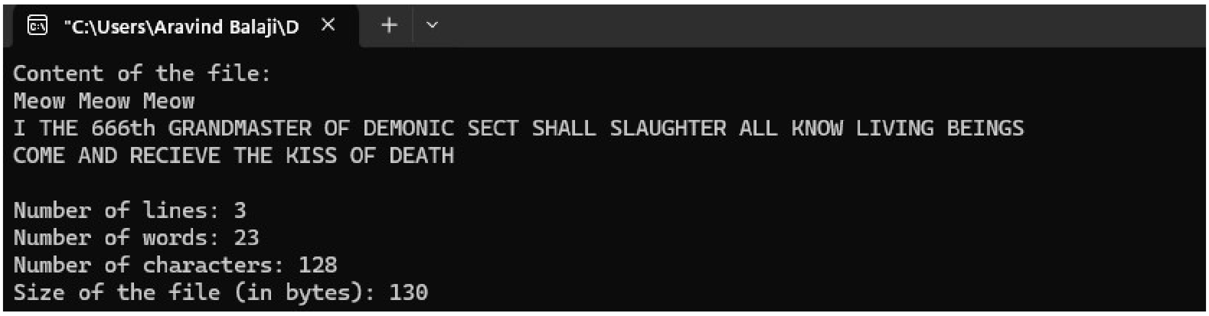
NAME : Arvind Balaji A

Reg no : 22BCE0666

Code :

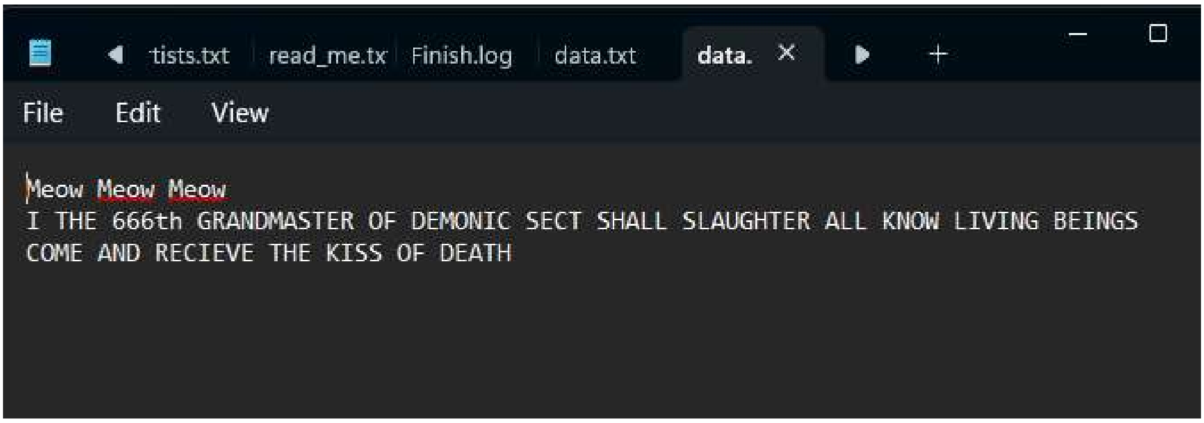
```
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
using namespace std;
void analyzeFile(const string &filename)
{
    ifstream file(filename, ios::in | ios::binary);
    if (!file.is_open())
    {
        cout << "Unable to open file: " << filename << endl;
        return;
    }
    string content, line;
    int lineCount = 0;
    int wordCount = 0;
    int charCount = 0;
    streampos fileSize = 0;
    while (getline(file, line))
    {
        content += line + '\n';
        lineCount++;
        bool inWord = false;
        for (char c : line)
        {
            charCount++;
            if (isspace(c))
            {
                inWord = false;
            }
            else if (!inWord && isalnum(c))
            {
                inWord = true;
                wordCount++;
            }
        }
    }
    file.clear();
    file.seekg(0, ios::end);
    fileSize = file.tellg();
    file.close();
    if (fileSize == -1)
    {
        cout << "Error getting file size." << endl;
    }
    else
    {
        cout << "Content of the file:\n"
             << content << endl;
        cout << "Number of lines: " << lineCount << endl;
        cout << "Number of words: " << wordCount << endl;
        cout << "Number of characters: " << charCount << endl;
        cout << "Size of the file (in bytes): " << fileSize << endl;
    }
}
int main()
{
    string filename = "data.txt"; // Replace with your file name
    analyzeFile(filename);

    return 0;

}
```

Output :



```
Content of the file:
Meow Meow Meow
I THE 666th GRANDMASTER OF DEMONIC SECT SHALL SLAUGHTER ALL KNOW LIVING BEINGS
COME AND RECIEVE THE KISS OF DEATH

Number of lines: 3
Number of words: 23
Number of characters: 128
Size of the file (in bytes): 130
```

Data.txt file :



```
Meow Meow Meow
I THE 666th GRANDMASTER OF DEMONIC SECT SHALL SLAUGHTER ALL KNOW LIVING BEINGS
COME AND RECIEVE THE KISS OF DEATH
```

Lexical Analysis in c++

Name: Arvind Balaji A

Registration number:22BCE0666

Code:

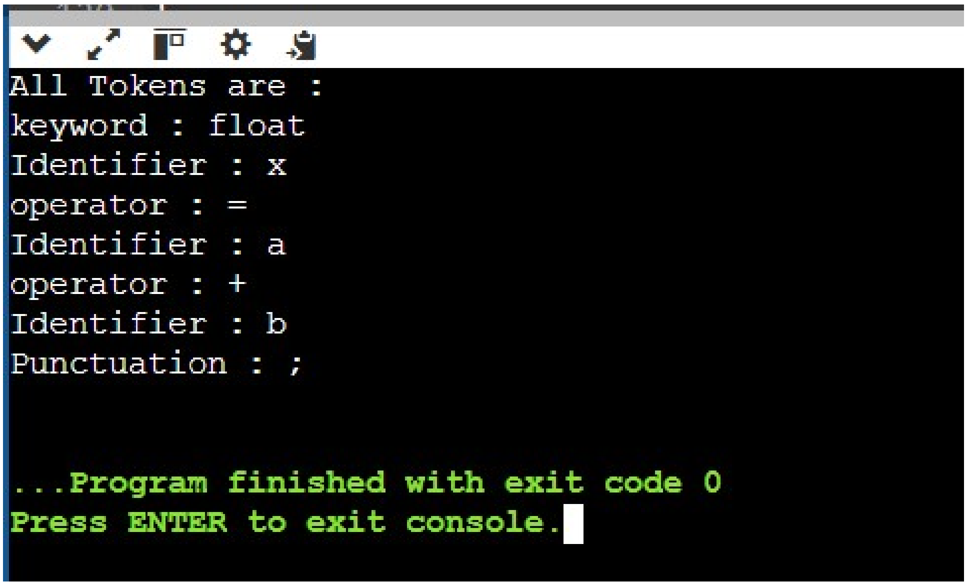
```
#include <iostream>
#include <vector>
#include <string>
#include <cctype>
#include <unordered_set>
#include <unordered_map>
using namespace std;
enum TokenType
{
    KEYWORD,
    IDENTIFIER,
    OPERATOR,
    NUMBER,
    PUNCTUATION,
    UNKNOWN
};
struct Token
{
    TokenType type;
    string value;
};
const unordered_set<string> keywords = {
    "int", "float", "return", "if", "else", "for", "while", "do",
    "break", "continue", "double", "char", "void", "const", "bool",
    "true", "false", "class", "struct", "public", "private", "protected",
    "namespace", "using", "new", "delete", "sizeof", "this", "throw",
    "try", "catch", "include", "define"};
const unordered_set<char> operators = {
    '+', '-', '*', '/', '%', '=', '<', '>', '&', '|', '!', '^', '~'};
const unordered_set<char> punctuation = {
    '(', ')', '{', '}', '[', ']', ';', ',', '.', ':', '?', '#'};
TokenType identifyToken(const string &str)
{
    if (keywords.find(str) != keywords.end())
    {
        return KEYWORD;
    }
    if (isalpha(str[0]) || str[0] == '_')
    {
        return IDENTIFIER;
    }
    if (isdigit(str[0]) || (str[0] == '.' && str.size() > 1 && isdigit(str[1])))
    {
        return NUMBER;
    }
    if (str.length() == 1 && operators.find(str[0]) != operators.end())
    {
        return OPERATOR;
    }
    if (str.length() == 1 && punctuation.find(str[0]) != punctuation.end())
    {
        return PUNCTUATION;
    }
    return UNKNOWN;
}
vector<Token> lexicalAnalysis(const string &input)
{
```

```
vector<Token> tokens;
string current;
for (size_t i = 0; i < input.size(); ++i)
{
    char ch = input[i];
    if (isspace(ch))
    {
        if (!current.empty())
        {
            tokens.push_back({identifyToken(current), current});
            current.clear();
        }
    }
    else if (operators.find(ch) != operators.end() || punctuation.find(ch) != punctuation.end())
    {
        if (!current.empty())
        {
            tokens.push_back({identifyToken(current), current});
            current.clear();
        }
        tokens.push_back({identifyToken(string(1, ch)), string(1, ch)});
    }
    else
    {
        current += ch;
    }
}
if (!current.empty())
{
    tokens.push_back({identifyToken(current), current});
}
```

```
    return tokens;
}
void printTokens(const vector<Token> &tokens)
{
    cout << "All Tokens are :" << endl;
    for (const Token &token : tokens)
    {
        string tokenType;
        switch (token.type)
        {
        case KEYWORD:
            tokenType = "keyword";
            break;
        case IDENTIFIER:
            tokenType = "Identifier";
            break;
        case OPERATOR:
            tokenType = "operator";
            break;
        case NUMBER:
            tokenType = "Number";
            break;
        case PUNCTUATION:
            tokenType = "Punctuation";
            break;
        default:
            tokenType = "Unknown";
            break;
        }
        cout << tokenType << " : " << token.value << endl;
    }
```

```
}
int main()
{
    string input = "float x = a + b;";
    auto tokens = lexicalAnalysis(input);
    printTokens(tokens);
    return 0;
}
```

Output:



```
All Tokens are :
keyword : float
Identifier : x
operator : =
Identifier : a
operator : +
Identifier : b
Punctuation : ;


...Program finished with exit code 0
Press ENTER to exit console.
```

Symbol Table code c++

NAME : Arvind balaji A

Reg no: 22BCE0666

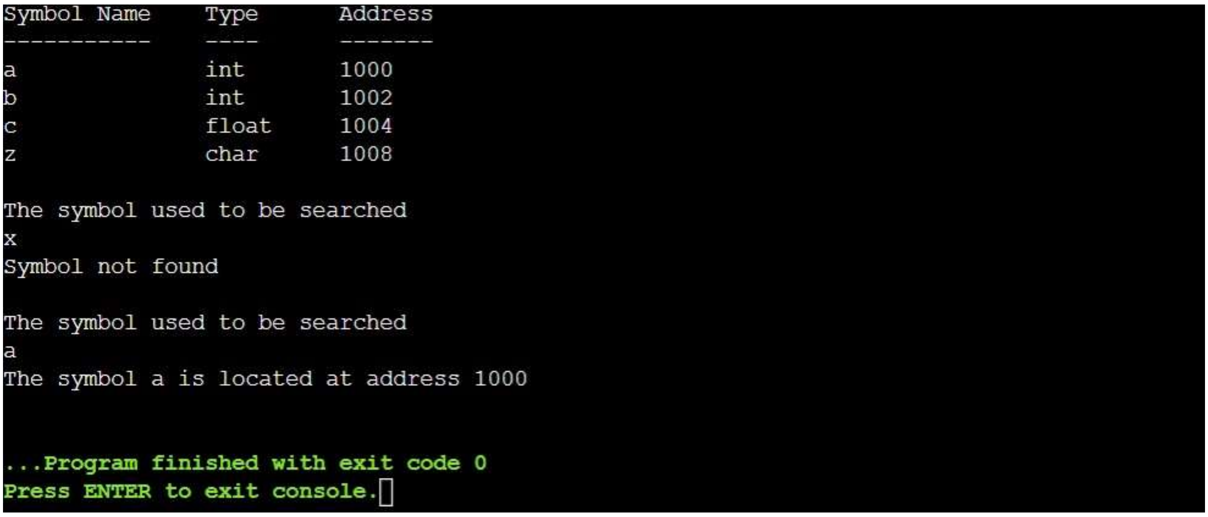
Code :

```
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
#include <iomanip>
#include <sstream>

using namespace std;

struct Symbol {
  string  name;
  string type;
  int address;
};

class SymbolTable {
private:
  vector<Symbol> symbols;
  unordered_map<string, int> symbolMap;
  int nextAddress;

public:
```

```
SymbolTable() : nextAddress(1000) {}

void addSymbol(const string& name, const string& type) {
    if (symbolMap.find(name) != symbolMap.end()) {
        cout << "Symbol " << name << " already exists in the table." << endl;
        return;
    }
    Symbol sym = { name, type, nextAddress };
    symbols.push_back(sym);
    symbolMap[name] = symbols.size() - 1;
    nextAddress += (type == "float") ? 4 : 2;
}

Symbol* searchSymbol(const string& name) {
    if (symbolMap.find(name) != symbolMap.end()) {
        return &symbols[symbolMap[name]];
    }
    return nullptr;
}

void display() {
    cout << left << setw(15) << "Symbol Name"
         << setw(10) << "Type"
         << setw(10) << "Address" << endl;

    cout << left << setw(15) << " --------- "
         << setw(10) << " -- "
         << setw(10) << " ----- " << endl;

    for (const Symbol& sym : symbols) {
        cout << left << setw(15) << sym.name
```

```
            << setw(10) << sym.type
            << setw(10) << sym.address << endl;
    }
}

void parseAndAddSymbols(const string& declaration) {
    istringstream iss(declaration);
    string token;
    string type;

    while (getline(iss, token, ';')) {
        token.erase(0, token.find_first_not_of(" \t"));
        token.erase(token.find_last_not_of(" \t") + 1);

        size_t pos = token.find(' ');
        if (pos != string::npos) {
            type = token.substr(0, pos);
            token = token.substr(pos + 1);
        }

        istringstream varsStream(token);
        string var;
        while (getline(varsStream, var, ',')) {
            var.erase(0, var.find_first_not_of(" \t"));
            var.erase(var.find_last_not_of(" \t") + 1);

            if (!var.empty()) {
                addSymbol(var, type);
            }
        }
    }
    }
};

int main() {
    SymbolTable table;

    string input = "int a, b; float c; char z;";

    table.parseAndAddSymbols(input);

    table.display();

    vector<string> symbolsToSearch = { "x", "a" };
    for (const string& symbol : symbolsToSearch) {
        cout << "\nThe symbol used to be searched\n" << symbol << endl;
        Symbol* sym = table.searchSymbol(symbol);
        if (sym) {
            cout << "The symbol " << sym->name << " is located at address " << sym->address << endl;
        } else {
            cout << "Symbol not found" << endl;
        }
    }

    return 0;
}
```

Output :



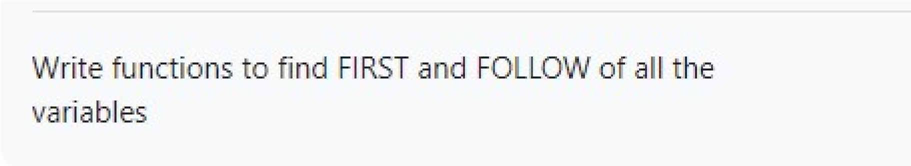
```
Symbol Name     Type      Address
-----------     ----      -------
a               int       1000
b               int       1002
c               float     1004
z               char      1008

The symbol used to be searched
x
Symbol not found

The symbol used to be searched
a
The symbol a is located at address 1000


...Program finished with exit code 0
Press ENTER to exit console.
```

Predictive Parsing CODE

Name : Arvind Balaji A

Regno: 22BCE0666

Question :



Write functions to find FIRST and FOLLOW of all the variables

CODE :

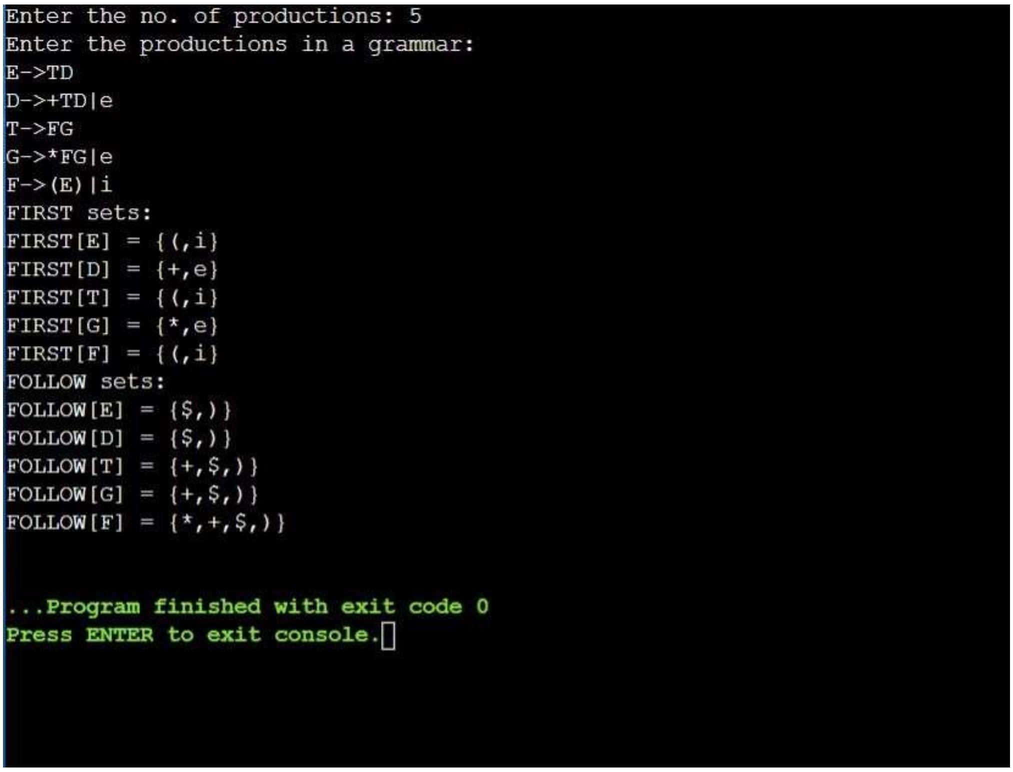
```
#include<s

tdio.h>

#include<s

tring.h>


char fin[10][20], st[10][20], ft[20][20], fol[20][20];
 int a = 0, e, i, t, b, c, n, k, l = 0, j, s, m, p;
```

```
void first(){
   for(i = 0; i < n; i++)
    fol[i][0] = '\0';  // Initialize FOLLOW sets


  for(s = 0; s <

    n; s++) {

    for(i = 0; i <

    n; i++) {

      j = 3; l = 0; a = 0;
      l1: if(!((st[i][j] > 64) && (st[i][j] <

        91))) { // Terminal for(m = 0; m <

        l; m++) {

          if(ft[i][m]

            == st[i][j])

            goto s1;

        }
  ft[i][l] = st[i][j]; // Add

  terminal to FIRST setl++;

  s1: j++;

} else { // Non-

  terminal if(s >

  0) {

    while(st[i][j]

      != st[a][0])

      a++;

    b = 0;

    while(ft[a][b] != '\0') {

      for(m = 0; m <

        l; m++) {

        if(ft[i][m] ==

        ft[a][b])

          goto s2;

      }

      ft[i][l] = ft[a][b];  // Add FIRST

      set of non-terminal l++;

      s2: b++;

    }

  }
```

```
      }
      while(st[i][j] != '\0') {
        if(st[i][j]

          == '|') {

          j++;
          goto l1;
        }

        j+

        +;

      }
      ft[i][l] = '\0'; // Terminate FIRST set

    }

  }

printf("FIRST sets:\n");
```

```
  for(i = 0; i < n; i++) {
    printf("FIRST[%c]

    = ", st[i][0]);

    printf("{");

    for(j = 0; ft[i][j]

      != '\0'; j++) {if

      (j > 0)

      printf(",");
      printf("%c", ft[i][j]);
    }
    printf("}");
    printf("\n");
  }


}
void follow(){
   fol[0][0] = '$';  // Start symbol has '$' in FOLLOW set


  for(i = 0; i < n; i++) {
    k = 0; j = 3; l = (i == 0) ? 1 : 0;
    k1: while((st[i][0] != st[k][j])

      && (k < n)) {if(st[k][j] ==
  '\0') {
    k+
    +;
    j =
    2;
  }
  j+
  +;
}
j+
+;
if(st[i][0] == st[k][j-1]) {
  if((st[k][j] != '|') &&
    (st[k][j] != '\0')) {a =
    0;
    if(!((st[k][j] > 64) &&
      (st[k][j] < 91))) {for(m =
      0; m < l; m++) {
        if(fol[i][m] == st[k][j])
      goto q3;

  }
  fol[i][l] =

  st[k][j];

  l++;

  q3: p++;
} else {
  while(st[k][j]

    != st[a][0])

    a++;
  p = 0;
  while(ft[a][p] != '\0') {
    if(ft[a][p] != 'e') {
      for(m = 0; m <

        l; m++) {

        if(fol[i][m] ==

        ft[a][p])

          goto q2;
      }
      fol[i][l] =

      ft[a][p];

      l++;
      } else
        e =
      1;
      q2:
      p++;
    }
    if(e ==
      1) {e
      = 0;
      goto a1;
    }
  }
} else {
  a1: c = 0; a = 0;
  while(st[k][0]
    != st[a][0])
    a++;
  while((fol[a][c] != '\0') && (st[a][0] != st[i][0])) {
          for(m = 0; m < l; m++) {
            if(fol[i][m] ==

              fol[a][c])

              goto q1;

          }

          fol[i][l] =

          fol[a][c];

          l++;

          q1: c++;

        }

      }

      goto k1;

    }

    fol[i][l] = '\0'; // Terminate FOLLOW set

  }

  printf("FOLLO

   W sets:\n");

   for(i = 0; i < n;

   i++) {
    printf("FOLLOW[%c

    ] = ", st[i][0]);

    printf("{");

    for(j = 0; fol[i][j]

      != '\0'; j++) {if (j

      > 0) printf(",");

      printf("%c", fol[i][j]);

    }

    printf("}");

    printf("\n");

  }

}

int main() {




  printf("Enter the no. of

  productions: ");

  scanf("%d", &n);
  printf("Enter the

  productions in a grammar:

  \n"); for(i = 0; i < n; i++)
    scanf("%s", st[i]);


  first();
  follow();


  return 0;

}
```

Output :



```
Enter the no. of productions: 5
Enter the productions in a grammar:
E->TD
D->+TD|e
T->FG
G->*FG|e
F->(E)|i
FIRST sets:
FIRST[E] = {(,i}
FIRST[D] = {+,e}
FIRST[T] = {(,i}
FIRST[G] = {*,e}
FIRST[F] = {(,i}
FOLLOW sets:
FOLLOW[E] = {$,)}
FOLLOW[D] = {$,)}
FOLLOW[T] = {+,$,)}
FOLLOW[G] = {+,$,)}
FOLLOW[F] = {*,+,$,)}


...Program finished with exit code 0
Press ENTER to exit console.
```

COMPILER DESIGN

NAME: ARVIND BALAJI A

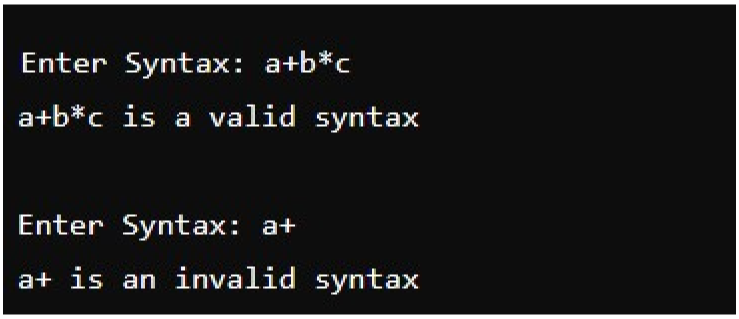
REG. NO. : 22BCE0666

Q) Write a C or C++ program to design syntax analyzer for sample language Sample input: Enter Syntax: a+b*c a+b*c is a valid syntax Enter Syntax: a+ a+ is a invalid syntax

Code:

```
#include <iostream>

#include <cctype>


using namespace std;


bool isValidSyntax(const string &expr) {
    int n = expr.length();


    if (!isalpha(expr[0])) {
        return false;
    }


    bool expectingOperand = false;


    for (int i = 1; i < n; i++) {
        char currentChar = expr[i];


        if (expectingOperand) {
            if (!isalpha(currentChar)) {
                return false;
            }
            expectingOperand = false;
        } else {
            if (!(currentChar == '+' || currentChar == '-' || currentChar == '*' || currentChar == '/')) {
                return false;
            }
            expectingOperand = true;
        }
    }

    return !expectingOperand;
}

int main() {
    string expr;

    while (true) {
        cout << "Enter Syntax: ";
        cin >> expr;

        if (isValidSyntax(expr)) {
            cout << expr << " is a valid syntax" << endl;
        } else {
            cout << expr << " is an invalid syntax" << endl;
        }
    }

    return 0;
}
```

Output:



```
Enter Syntax: a+b*c
a+b*c is a valid syntax

Enter Syntax: a+
a+ is an invalid syntax
```

QUADRUPLE TRIPLE CODE

```
#include<bits/stdc++.h>
using namespace std;

struct Quadruple {
    string op, arg1, arg2, result;
};

struct Triple {
    string op, arg1, arg2;
};

struct IndirectTriple {
    int index;
    Triple triple;
};

void printQuadruples(const vector<Quadruple>& quadruples) {
    cout << "\nQuadruples:" << endl;
    cout << "Op\tArg1\tArg2\tResult" << endl;
    for (const auto& q : quadruples) {
        cout << q.op << "\t" << q.arg1 << "\t" << q.arg2 <<
"\t" << q.result << endl;
    }
}

void printTriples(const vector<Triple>& triples) {
    cout << "\nTriples:" << endl;
    cout << "Index\tOp\tArg1\tArg2" << endl;
    for (size_t i = 0; i < triples.size(); ++i) {
        cout << i << "\t" << triples[i].op << "\t" <<
triples[i].arg1 << "\t" << triples[i].arg2 << endl;
    }
}

void printIndirectTriples(const vector<IndirectTriple>&
indirectTriples) {
    cout << "\nIndirect Triples:" << endl;
    cout << "Index\tOp\tArg1\tArg2" << endl;
    for (const auto& it : indirectTriples) {
        cout << it.index+10 << "\t" << it.triple.op << "\t" <<
it.triple.arg1 << "\t" << it.triple.arg2 << endl;
    }
}

int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}

void generateThreeAddressCode(const string&
expression, vector<Quadruple>& quadruples,
vector<Triple>& triples, string& lhsVar) {
    stack<string> operands;
    stack<char> operators;
    int tempCount = 1;
    lhsVar = "";

    auto handleOperation = [&](char op) {
        if (operands.size() < 2) {
            cerr << "Error: Not enough operands for the
operation " << op << endl;
            return;
        }
        string arg2 = operands.top(); operands.pop();
        string arg1 = operands.top(); operands.pop();
        string temp = "t" + to_string(tempCount++);

        quadruples.push_back({string(1, op), arg1, arg2,
temp});
        triples.push_back({string(1, op), arg1, arg2});
        operands.push(temp);
    };

    for (size_t i = 0; i < expression.length(); ++i) {
        if (expression[i] == ' ') continue;
        if (isalnum(expression[i])) {
            string operand;
            while (i < expression.length() &&
isalnum(expression[i])) {
```

```
                operand += expression[i++];
            }
            --i;
            operands.push(operand);
        }
        else if (expression[i] == '-' && (i == 0 || expression[i - 1] == '(' || expression[i - 1] == '*' || expression[i - 1] == '=')) {
            i++;
            string operand;
            while (i < expression.length() && isalnum(expression[i])) {
                operand += expression[i++];
            }
            --i;
            string temp = "t" + to_string(tempCount++);
            quadruples.push_back({"-", operand, "", temp});
            triples.push_back({"-", operand, ""});
            operands.push(temp);
        }
        else if (expression[i] == '(') {
            operators.push(expression[i]);
        }
        else if (expression[i] == ')') {
            while (!operators.empty() && operators.top() !=
'(') {
                handleOperation(operators.top());
                operators.pop();
            }
            if (!operators.empty()) {
                operators.pop();
            }
        }
        else if (precedence(expression[i]) > 0) {
            while (!operators.empty() &&
precedence(operators.top()) >=
precedence(expression[i])) {
                handleOperation(operators.top());
                operators.pop();
            }
            operators.push(expression[i]);
        }
        else if (expression[i] == '=') {
            lhsVar = operands.top();
            operands.pop();
        }
    }

    while (!operators.empty()) {
        handleOperation(operators.top());
        operators.pop();
    }

    if (!operands.empty()) {
        string result = operands.top();
        operands.pop();
        quadruples.push_back({"=", result, "", lhsVar});
        triples.push_back({"=", result, ""});
    }
}
```

```
vector<IndirectTriple> generateIndirectTriples(const
vector<Triple>& triples) {
    vector<IndirectTriple> indirectTriples;
    for (size_t i = 0; i < triples.size(); ++i) {
        indirectTriples.push_back({static_cast<int>(i),
triples[i]});
    }
    return indirectTriples;
}

int main() {
    string expression;
    string lhsVar;

    cout << "Enter an arithmetic expression: ";
    getline(cin, expression);

    vector<Quadruple> quadruples;
    vector<Triple> triples;
    generateThreeAddressCode(expression, quadruples, triples, lhsVar);

    vector<IndirectTriple> indirectTriples = generateIndirectTriples(triples);

    printQuadruples(quadruples);
    printTriples(triples);
    printIndirectTriples(indirectTriples);

    return 0;
}
```