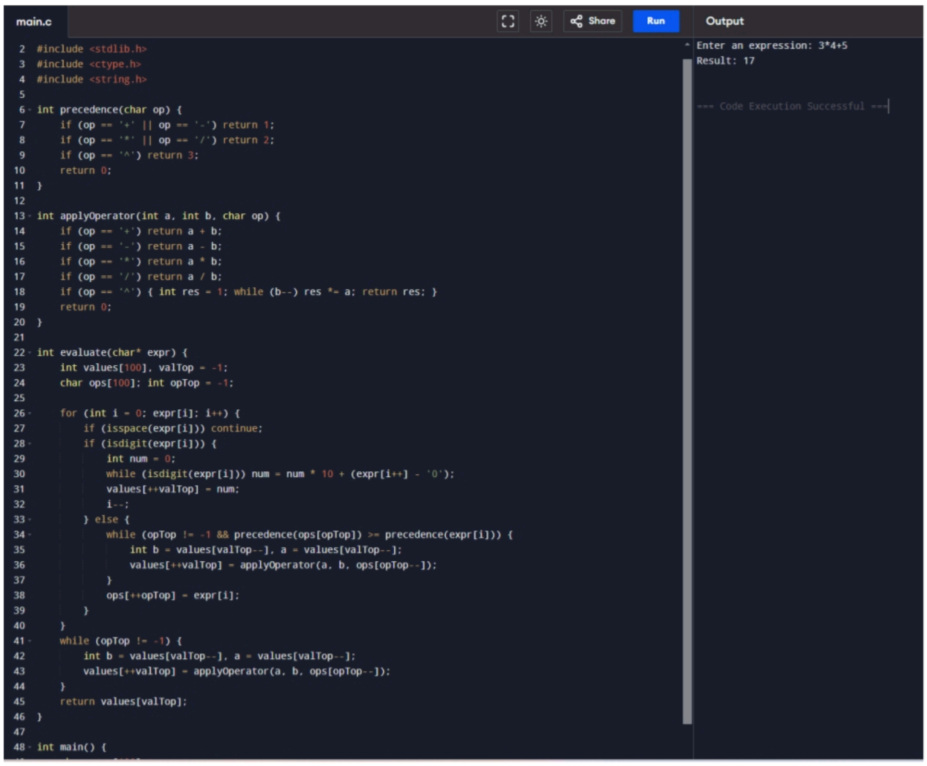
11. In a class of Grade 3, Mathematics Teacher asked for the Acronym PEMDAS?. All of them are thinking for a while. A smart kid of the class Kishore of the class says it is Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction. Can you write a C Program to help the students to understand about the operator precedence parsing for an expression containing more than one operator, the order of evaluation depends on the order of operations

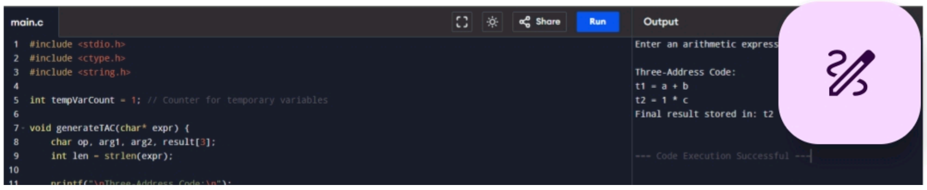


```c
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    if (op == '^') return 3;
    return 0;
}

int applyOperator(int a, int b, char op) {
    if (op == '+') return a + b;
    if (op == '-') return a - b;
    if (op == '*') return a * b;
    if (op == '/') return a / b;
    if (op == '^') { int res = 1; while (b--) res *= a; return res; }
    return 0;
}

int evaluate(char* expr) {
    int values[100], valTop = -1;
    char ops[100]; int opTop = -1;

    for (int i = 0; expr[i]; i++) {
        if (isspace(expr[i])) continue;
        if (isdigit(expr[i])) {
            int num = 0;
            while (isdigit(expr[i])) num = num * 10 + (expr[i++] - '0');
            values[++valTop] = num;
            i--;
        } else {
            while (opTop != -1 && precedence(ops[opTop]) >= precedence(expr[i])) {
                int b = values[valTop--], a = values[valTop--];
                values[++valTop] = applyOperator(a, b, ops[opTop--]);
            }
            ops[++opTop] = expr[i];
        }
    }
    while (opTop != -1) {
        int b = values[valTop--], a = values[valTop--];
        values[++valTop] = applyOperator(a, b, ops[opTop--]);
    }
    return values[valTop];
}

int main() {
```

```
Enter an expression: 3*4+5
Result: 17


=== Code Execution Successful ===
```

12. The main function of the Intermediate code generation is producing three address code statements for a given input expression. The three address codes help in determining the sequence in which operations are actioned by the compiler. The key work of Intermediate code generators is to simplify the process of Code Generator. Write a C Program to

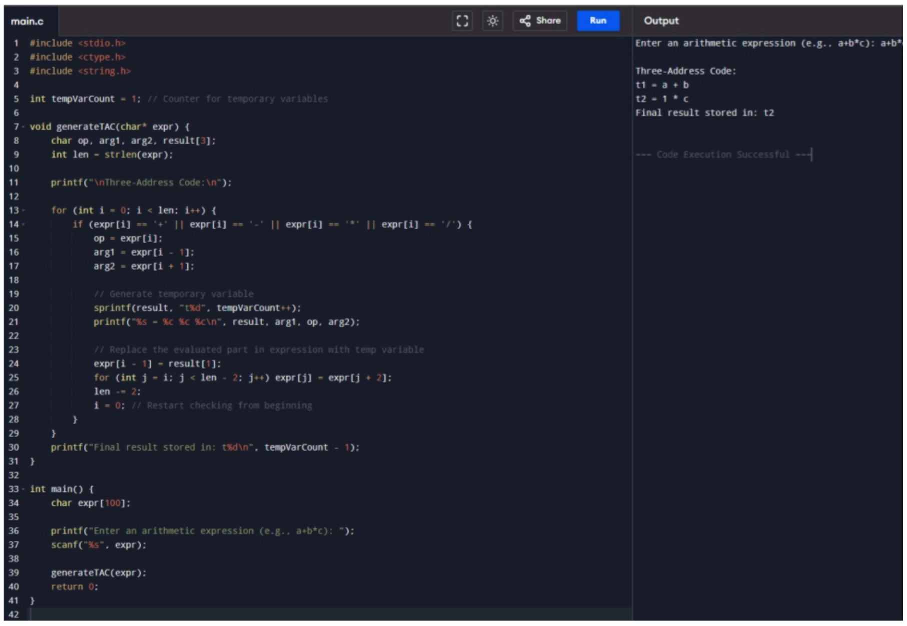
Generate the Three address code representation for the given input statement.



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

int tempVarCount = 1; // Counter for temporary variables

void generateTAC(char* expr) {
    char op, arg1, arg2, result[3];
    int len = strlen(expr);

    printf("\nThree-Address Code:\n");
```

```
Enter an arithmetic express

Three-Address Code:
t1 = a + b
t2 = 1 * c
Final result stored in: t2


--- Code Execution Successful ---
```

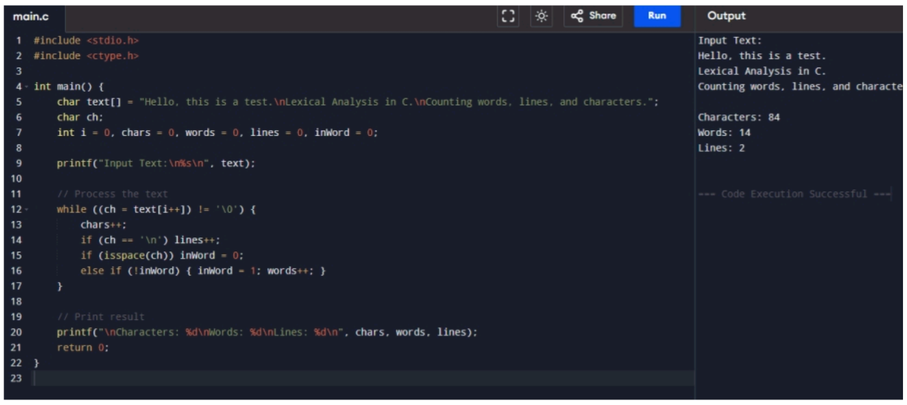
Generate the Three address code representation for the given input statement.



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

int tempVarCount = 1; // Counter for temporary variables

void generateTAC(char* expr) {
    char op, arg1, arg2, result[3];
    int len = strlen(expr);

    printf("\nThree-Address Code:\n");

    for (int i = 0; i < len; i++) {
        if (expr[i] == '+' || expr[i] == '-' || expr[i] == '*' || expr[i] == '/') {
            op = expr[i];
            arg1 = expr[i - 1];
            arg2 = expr[i + 1];

            // Generate temporary variable
            sprintf(result, "t%d", tempVarCount++);
            printf("%s = %c %c %c\n", result, arg1, op, arg2);

            // Replace the evaluated part in expression with temp variable
            expr[i - 1] = result[1];
            for (int j = i; j < len - 2; j++) expr[j] = expr[j + 2];
            len -= 2;
            i = 0; // Restart checking from beginning
        }
    }
    printf("Final result stored in: t%d\n", tempVarCount - 1);
}

int main() {
    char expr[100];

    printf("Enter an arithmetic expression (e.g., a+b*c): ");
    scanf("%s", expr);

    generateTAC(expr);
    return 0;
}
```

```
Enter an arithmetic expression (e.g., a+b*c): a+b*

Three-Address Code:
t1 = a + b
t2 = 1 * c
Final result stored in: t2


--- Code Execution Successful ---
```

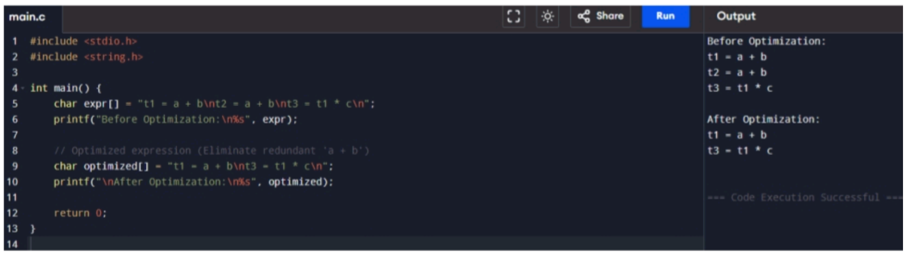
13. Write a C program for implementing a Lexical Analyzer to Count the number of characters, words, and lines



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

int main() {
    char text[] = "Hello, this is a test.\nLexical Analysis in C.\nCounting words, lines, and characters.";
    char ch;
    int i = 0, chars = 0, words = 0, lines = 0, inWord = 0;

    printf("Input Text:\n%s\n", text);

    // Process the text
    while ((ch = text[i++]) != '\0') {
        chars++;
        if (ch == '\n') lines++;
        if (isspace(ch)) inWord = 0;
        else if (!inWord) { inWord = 1; words++; }
    }

    // Print result
    printf("\nCharacters: %d\nWords: %d\nLines: %d\n", chars, words, lines);
    return 0;
}
```

```
Input Text:
Hello, this is a test.
Lexical Analysis in C.
Counting words, lines, and characte

Characters: 84
Words: 14
Lines: 2


--- Code Execution Successful ---
```

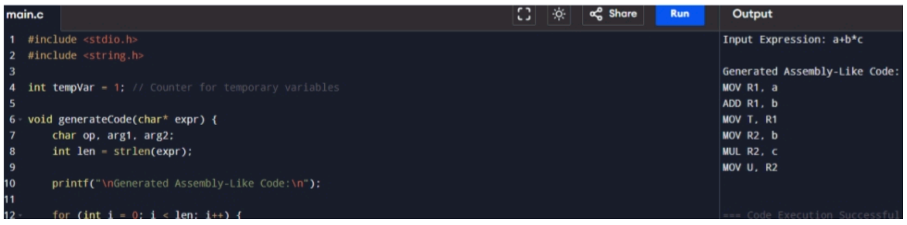
14. Write a C Program for code optimization to eliminate common subexpression.



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

int main() {
    char expr[] = "t1 = a + b\nt2 = a + b\nt3 = t1 * c\n";
    printf("Before Optimization:\n%s", expr);

    // Optimized expression (Eliminate redundant 'a + b')
    char optimized[] = "t1 = a + b\nt3 = t1 * c\n";
    printf("\nAfter Optimization:\n%s", optimized);

    return 0;
}
```

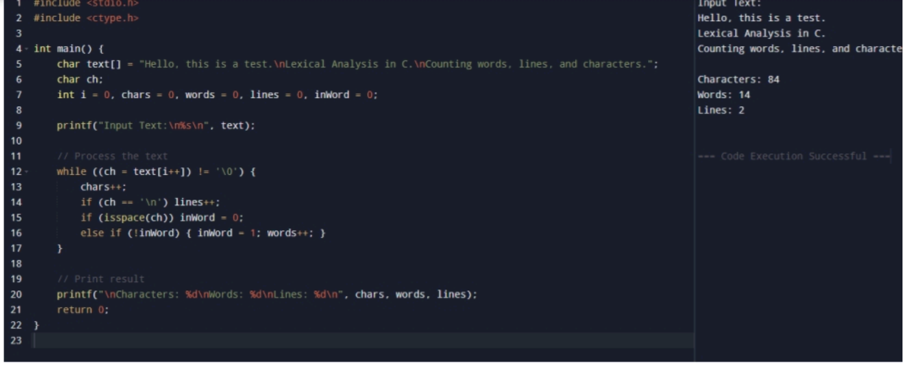
```
Before Optimization:
t1 = a + b
t2 = a + b
t3 = t1 * c

After Optimization:
t1 = a + b
t3 = t1 * c


=== Code Execution Successful ===
```

15. Write a C program to implement the back end of the compiler.



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

int tempVar = 1; // Counter for temporary variables

void generateCode(char* expr) {
    char op, arg1, arg2;
    int len = strlen(expr);

    printf("\nGenerated Assembly-Like Code:\n");

    for (int i = 0; i < len; i++) {
```

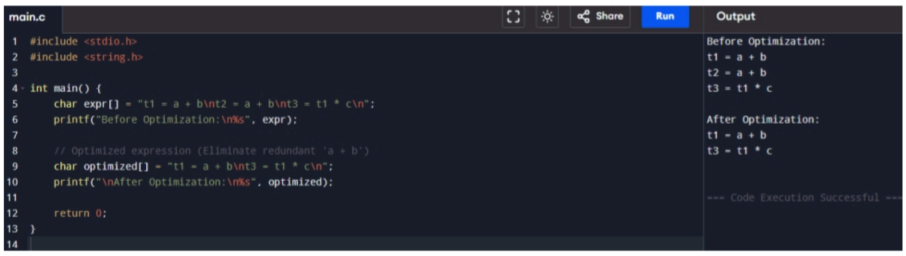
```
Input Expression: a+b*c

Generated Assembly-Like Code:
MOV R1, a
ADD R1, b
MOV T, R1
MOV R2, b
MUL R2, c
MOV U, R2


=== Code Execution Successful
```



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

int main() {
    char text[] = "Hello, this is a test.\nLexical Analysis in C.\nCounting words, lines, and characters.";
    char ch;
    int i = 0, chars = 0, words = 0, lines = 0, inWord = 0;

    printf("Input Text:\n%s\n", text);

    // Process the text
    while ((ch = text[i++]) != '\0') {
        chars++;
        if (ch == '\n') lines++;
        if (isspace(ch)) inWord = 0;
        else if (!inWord) { inWord = 1; words++; }
    }

    // Print result
    printf("\nCharacters: %d\nWords: %d\nLines: %d\n", chars, words, lines);
    return 0;
}
```

```
Input Text:
Hello, this is a test.
Lexical Analysis in C.
Counting words, lines, and characte

Characters: 84
Words: 14
Lines: 2


=== Code Execution Successful ===
```

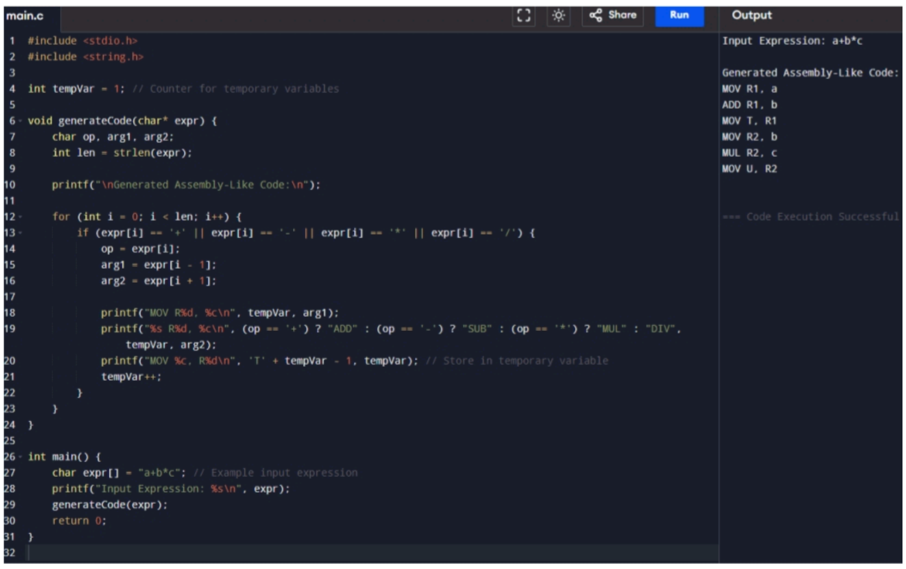
14. Write a C Program for code optimization to eliminate common subexpression.



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

int main() {
    char expr[] = "t1 = a + b\nt2 = a + b\nt3 = t1 * c\n";
    printf("Before Optimization:\n%s", expr);

    // Optimized expression (Eliminate redundant 'a + b')
    char optimized[] = "t1 = a + b\nt3 = t1 * c\n";
    printf("\nAfter Optimization:\n%s", optimized);

    return 0;
}
```

```
Before Optimization:
t1 = a + b
t2 = a + b
t3 = t1 * c

After Optimization:
t1 = a + b
t3 = t1 * c


=== Code Execution Successful ===
```

15. Write a C program to implement the back end of the compiler.

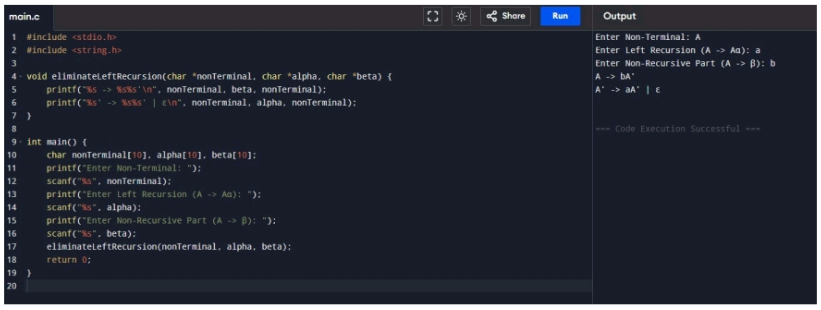


```c
#include <stdio.h>
#include <string.h>

int tempVar = 1; // Counter for temporary variables

void generateCode(char* expr) {
    char op, arg1, arg2;
    int len = strlen(expr);

    printf("\nGenerated Assembly-Like Code:\n");

    for (int i = 0; i < len; i++) {
        if (expr[i] == '+' || expr[i] == '-' || expr[i] == '*' || expr[i] == '/') {
            op = expr[i];
            arg1 = expr[i - 1];
            arg2 = expr[i + 1];

            printf("MOV R%d, %c\n", tempVar, arg1);
            printf("%s R%d, %c\n", (op == '+') ? "ADD" : (op == '-') ? "SUB" : (op == '*') ? "MUL" : "DIV",
                tempVar, arg2);
            printf("MOV %c, R%d\n", 'T' + tempVar - 1, tempVar); // Store in temporary variable
            tempVar++;
        }
    }
}

int main() {
    char expr[] = "a+b*c"; // Example input expression
    printf("Input Expression: %s\n", expr);
    generateCode(expr);
    return 0;
}
```

```
Input Expression: a+b*c

Generated Assembly-Like Code:
MOV R1, a
ADD R1, b
MOV T, R1
MOV R2, b
MUL R2, c
MOV U, R2


=== Code Execution Successful
```

LIST OF EXPERIMENTS

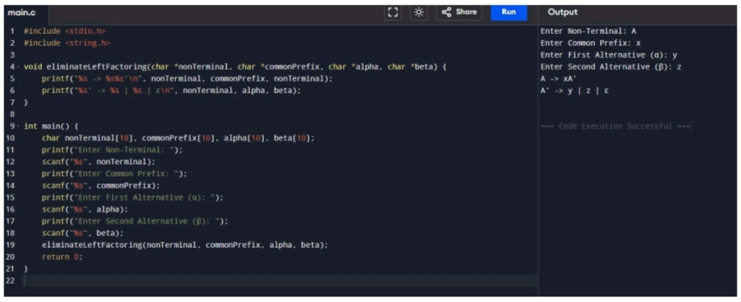
6.Implement a C program to eliminate left recursion



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

void eliminateLeftRecursion(char *nonTerminal, char *alpha, char *beta) {
    printf("%s -> %s%s'\n", nonTerminal, beta, nonTerminal);
    printf("%s' -> %s%s' | ε\n", nonTerminal, alpha, nonTerminal);
}

int main() {
    char nonTerminal[10], alpha[10], beta[10];
    printf("Enter Non-Terminal: ");
    scanf("%s", nonTerminal);
    printf("Enter Left Recursion (A -> Aα): ");
    scanf("%s", alpha);
    printf("Enter Non-Recursive Part (A -> β): ");
    scanf("%s", beta);
    eliminateLeftRecursion(nonTerminal, alpha, beta);
    return 0;
}
```

```
Enter Non-Terminal: A
Enter Left Recursion (A -> Aα): a
Enter Non-Recursive Part (A -> β): b
A -> bA'
A' -> aA' | ε


=== Code Execution Successful ===
```

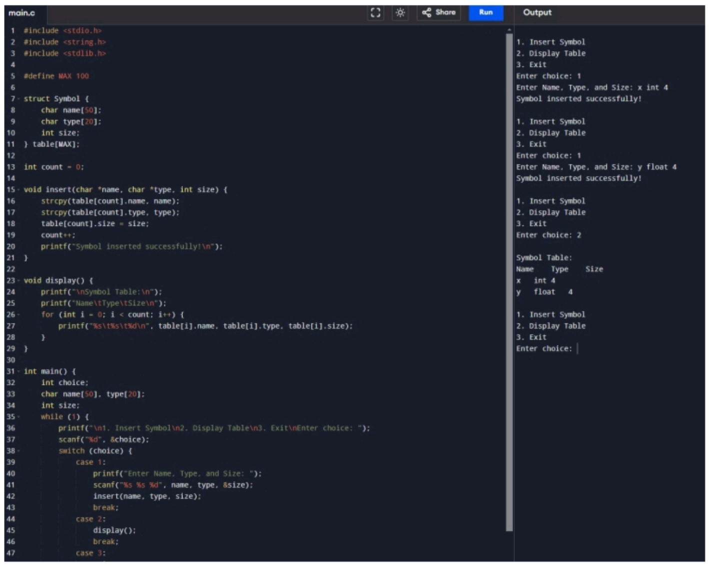
7.Implement a C program to eliminate left factoring.



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

void eliminateLeftFactoring(char *nonTerminal, char *commonPrefix, char *alpha, char *beta) {
    printf("%s -> %s%s'\n", nonTerminal, commonPrefix, nonTerminal);
    printf("%s' -> %s | %s | ε\n", nonTerminal, alpha, beta);
}

int main() {
    char nonTerminal[10], commonPrefix[10], alpha[10], beta[10];
    printf("Enter Non-Terminal: ");
    scanf("%s", nonTerminal);
    printf("Enter Common Prefix: ");
    scanf("%s", commonPrefix);
    printf("Enter First Alternative (α): ");
    scanf("%s", alpha);
    printf("Enter Second Alternative (β): ");
    scanf("%s", beta);
    eliminateLeftFactoring(nonTerminal, commonPrefix, alpha, beta);
    return 0;
}
```

```
Enter Non-Terminal: A
Enter Common Prefix: x
Enter First Alternative (α): y
Enter Second Alternative (β): z
A -> xA'
A' -> y | z | ε


=== Code Execution Successful ===
```

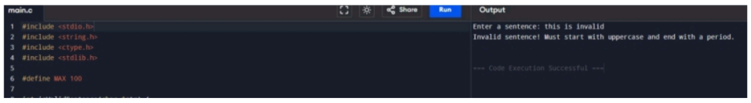
8.Implement a C program to perform symbol table operations.



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

#define MAX 100

struct Symbol {
    char name[50];
    char type[20];
    int size;
} table[MAX];

int count = 0;

void insert(char *name, char *type, int size) {
    strcpy(table[count].name, name);
    strcpy(table[count].type, type);
    table[count].size = size;
    count++;
    printf("Symbol inserted successfully!\n");
}

void display() {
    printf("\nSymbol Table:\n");
    printf("Name\tType\tSize\n");
    for (int i = 0; i < count; i++) {
        printf("%s\t%s\t%d\n", table[i].name, table[i].type, table[i].size);
    }
}

int main() {
    int choice;
    char name[50], type[20];
    int size;
    while (1) {
        printf("\n1. Insert Symbol\n2. Display Table\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter Name, Type, and Size: ");
                scanf("%s %s %d", name, type, &size);
                insert(name, type, size);
                break;
            case 2:
                display();
                break;
            case 3:
```

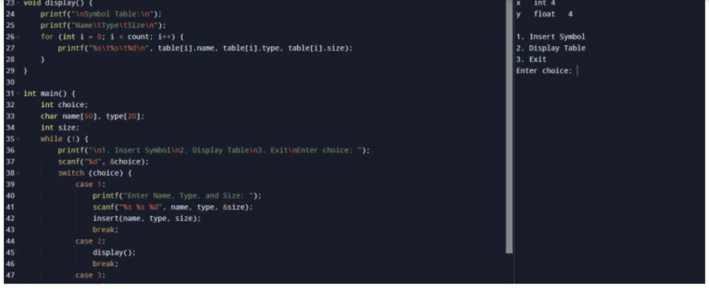
```

1. Insert Symbol
2. Display Table
3. Exit
Enter choice: 1
Enter Name, Type, and Size: x int 4
Symbol inserted successfully!

1. Insert Symbol
2. Display Table
3. Exit
Enter choice: 1
Enter Name, Type, and Size: y float 4
Symbol inserted successfully!

1. Insert Symbol
2. Display Table
3. Exit
Enter choice: 2

Symbol Table:
Name    Type    Size
x   int 4
y   float   4

1. Insert Symbol
2. Display Table
3. Exit
Enter choice:
```

9.All languages have Grammar. When people frame a sentence we usually say whether the sentence is framed as per the rules of the Grammar or Not. Similarly use the same ideology , implement to check whether the given input string is satisfying the grammar or not .



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

#define MAX 100

```

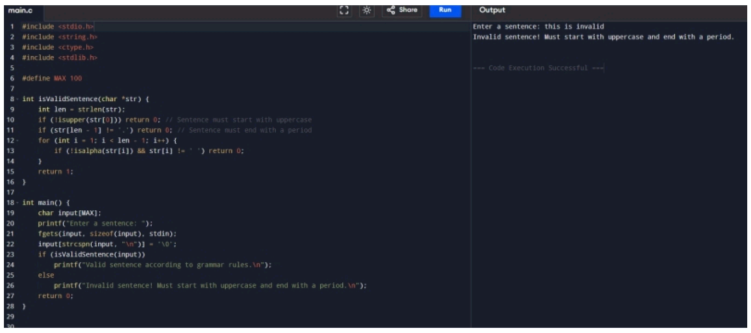
```
Enter a sentence: this is invalid
Invalid sentence! Must start with uppercase and end with a period.


=== Code Execution Successful ===
```



```c
void display() {
    printf("\nSymbol Table:\n");
    printf("Name\tType\tSize\n");
    for (int i = 0; i < count; i++) {
        printf("%s\t%s\t%d\n", table[i].name, table[i].type, table[i].size);
    }
}

int main() {
    int choice;
    char name[50], type[20];
    int size;
    while (1) {
        printf("\n1. Insert Symbol\n2. Display Table\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter Name, Type, and Size: ");
                scanf("%s %s %d", name, type, &size);
                insert(name, type, size);
                break;
            case 2:
                display();
                break;
            case 3:
```

```
x   int 4
y   float   4

1. Insert Symbol
2. Display Table
3. Exit
Enter choice:
```

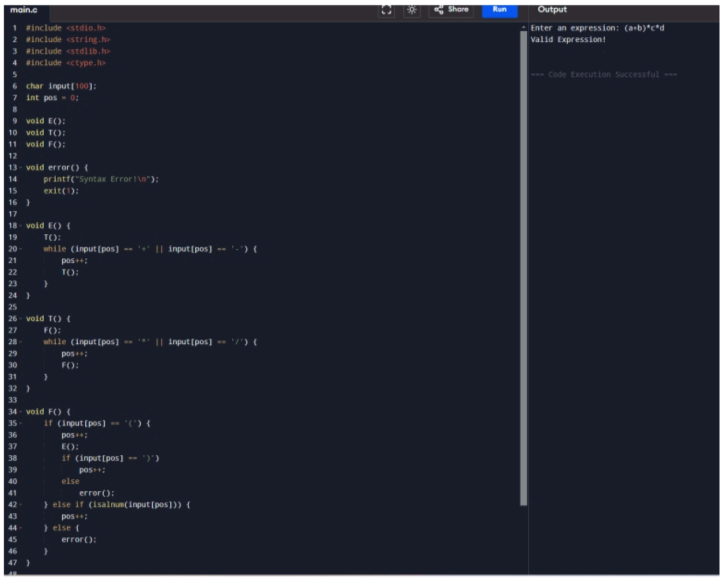
9.All languages have Grammar. When people frame a sentence we usually say whether the sentence is framed as per the rules of the Grammar or Not. Similarly use the same ideology , implement to check whether the given input string is satisfying the grammar or not .



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

#define MAX 100

int isValidSentence(char *str) {
    int len = strlen(str);
    if (!isupper(str[0])) return 0; // Sentence must start with uppercase
    if (str[len - 1] != '.') return 0; // Sentence must end with a period
    for (int i = 1; i < len - 1; i++) {
        if (!isalpha(str[i]) && str[i] != ' ') return 0;
    }
    return 1;
}

int main() {
    char input[MAX];
    printf("Enter a sentence: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = '\0';
    if (isValidSentence(input))
        printf("Valid sentence according to grammar rules.\n");
    else
        printf("Invalid sentence! Must start with uppercase and end with a period.\n");
    return 0;
}
```

```
Enter a sentence: this is invalid
Invalid sentence! Must start with uppercase and end with a period.


=== Code Execution Successful ===
```

10.Write a C program to construct recursive descent parsing.



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

char input[100];
int pos = 0;

void E();
void T();
void F();

void error() {
    printf("Syntax Error!\n");
    exit(1);
}

void E() {
    T();
    while (input[pos] == '+' || input[pos] == '-') {
        pos++;
        T();
    }
}

void T() {
    F();
    while (input[pos] == '*' || input[pos] == '/') {
        pos++;
        F();
    }
}

void F() {
    if (input[pos] == '(') {
        pos++;
        E();
        if (input[pos] == ')')
            pos++;
        else
            error();
    } else if (isalnum(input[pos])) {
        pos++;
    } else {
        error();
    }
}
```

```
Enter an expression: (a+b)*c*d
Valid Expression!


=== Code Execution Successful ===
```