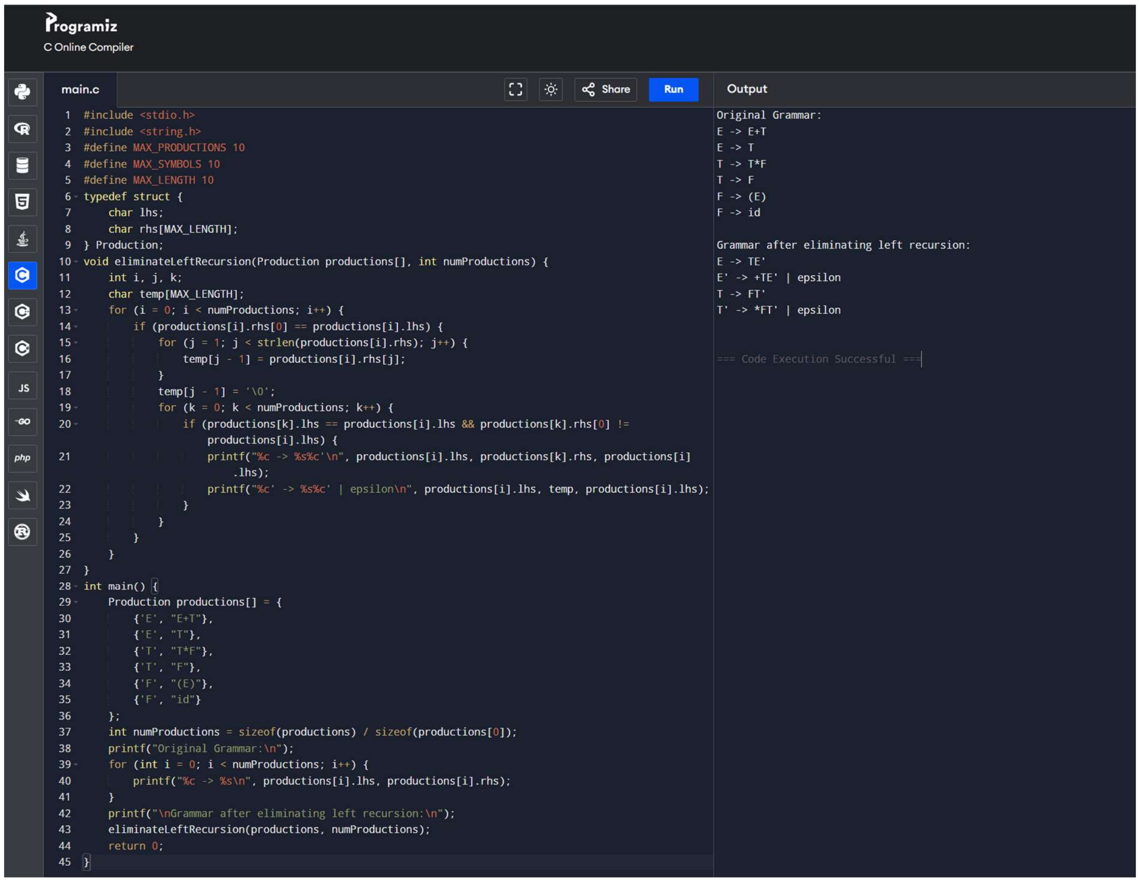
# Compiler Design (10/02/20250

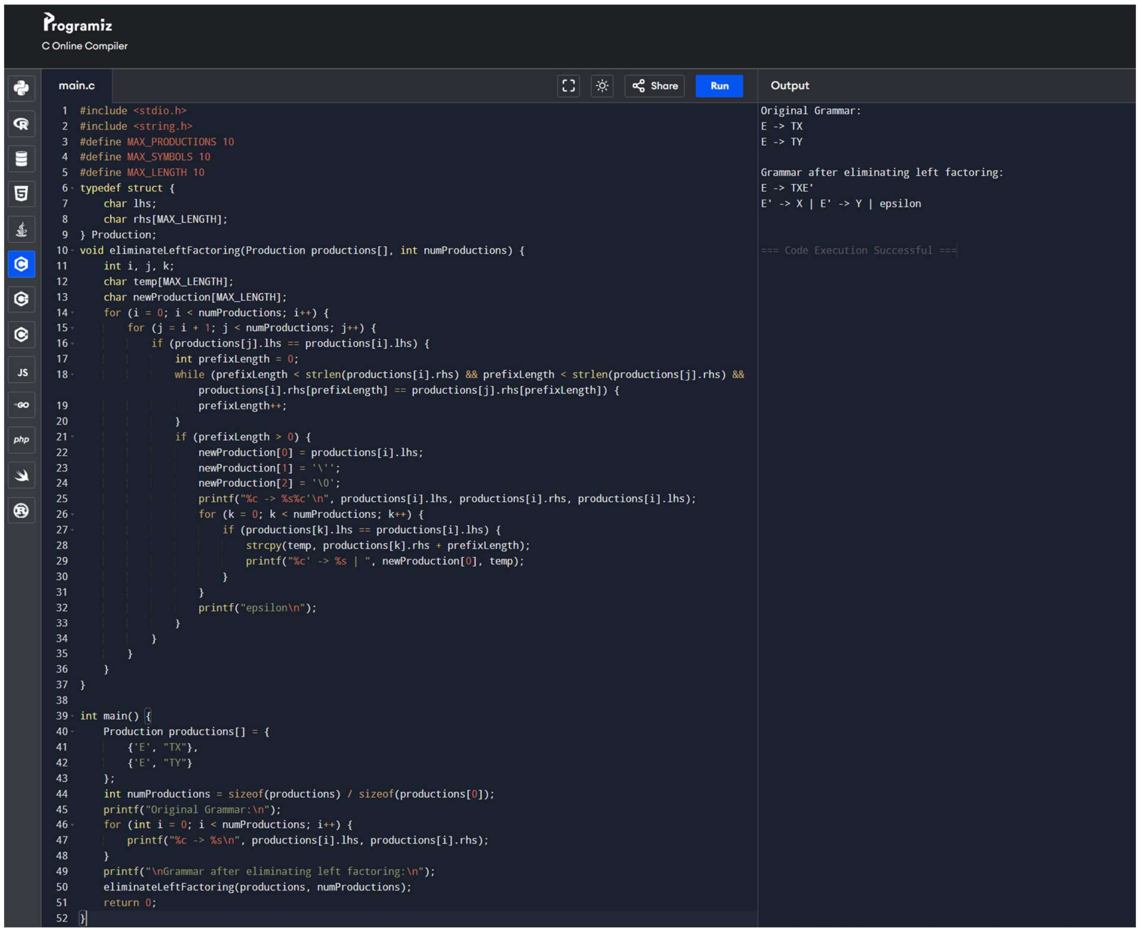
## 6. Implement a C program to eliminate left recursion.



```c
#include <stdio.h>
#include <string.h>
#define MAX_PRODUCTIONS 10
#define MAX_SYMBOLS 10
#define MAX_LENGTH 10
typedef struct {
    char lhs;
    char rhs[MAX_LENGTH];
} Production;
void eliminateLeftRecursion(Production productions[], int numProductions) {
    int i, j, k;
    char temp[MAX_LENGTH];
    for (i = 0; i < numProductions; i++) {
        if (productions[i].rhs[0] == productions[i].lhs) {
            for (j = 1; j < strlen(productions[i].rhs); j++) {
                temp[j - 1] = productions[i].rhs[j];
            }
            temp[j - 1] = '\0';
            for (k = 0; k < numProductions; k++) {
                if (productions[k].lhs == productions[i].lhs && productions[k].rhs[0] != productions[i].lhs) {
                    printf("%c -> %s%c'\n", productions[i].lhs, productions[k].rhs, productions[i].lhs);
                    printf("%c' -> %s%c' | epsilon\n", productions[i].lhs, temp, productions[i].lhs);
                }
            }
        }
    }
}
int main() {
    Production productions[] = {
        {'E', "E+T"},
        {'E', "T"},
        {'T', "T*F"},
        {'T', "F"},
        {'F', "(E)"},
        {'F', "id"}
    };
    int numProductions = sizeof(productions) / sizeof(productions[0]);
    printf("Original Grammar:\n");
    for (int i = 0; i < numProductions; i++) {
        printf("%c -> %s\n", productions[i].lhs, productions[i].rhs);
    }
    printf("\nGrammar after eliminating left recursion:\n");
    eliminateLeftRecursion(productions, numProductions);
    return 0;
}
```

```
Original Grammar:
E -> E+T
E -> T
T -> T*F
T -> F
F -> (E)
F -> id

Grammar after eliminating left recursion:
E -> TE'
E' -> +TE' | epsilon
T -> FT'
T' -> *FT' | epsilon


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

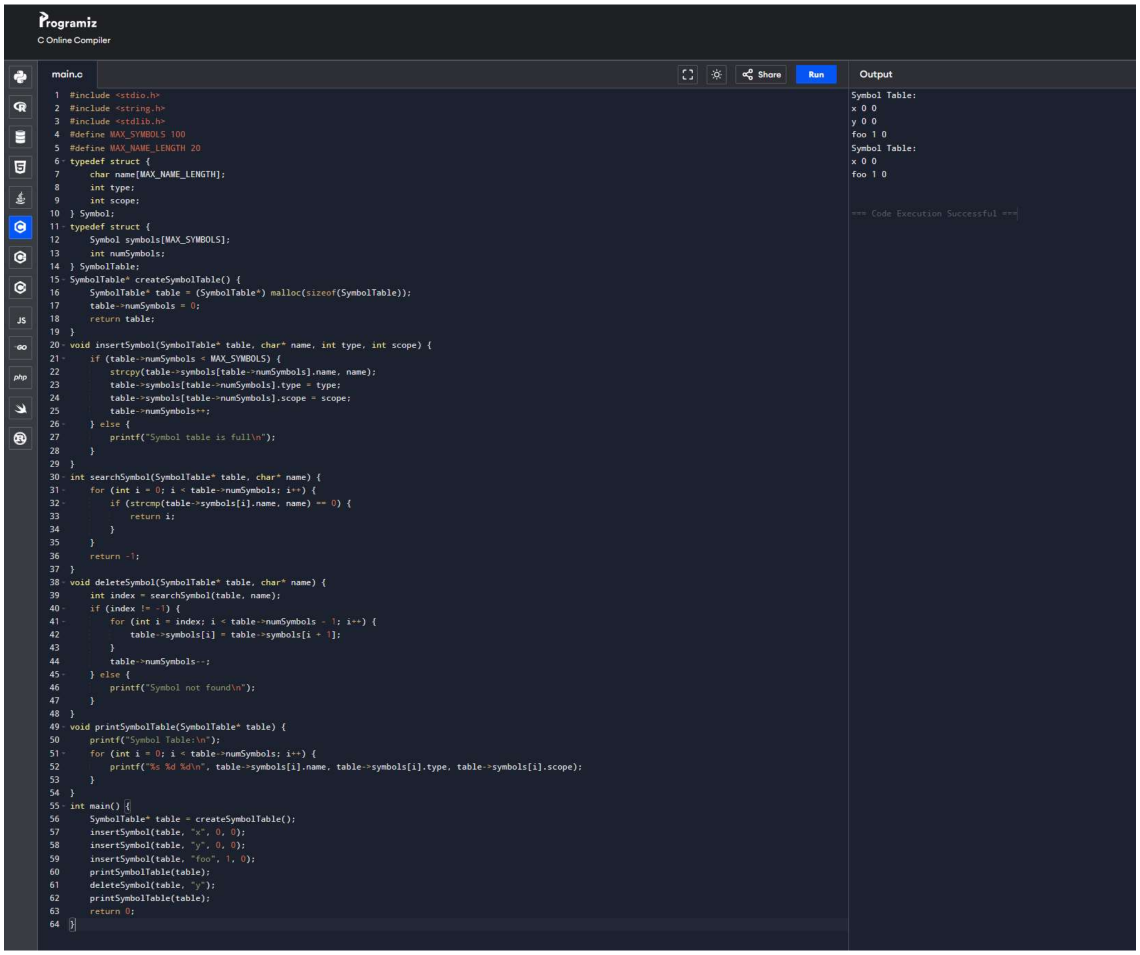
# 7.Implement a C program to eliminate left factoring.



```c
#include <stdio.h>
#include <string.h>
#define MAX_PRODUCTIONS 10
#define MAX_SYMBOLS 10
#define MAX_LENGTH 10
typedef struct {
    char lhs;
    char rhs[MAX_LENGTH];
} Production;
void eliminateLeftFactoring(Production productions[], int numProductions) {
    int i, j, k;
    char temp[MAX_LENGTH];
    char newProduction[MAX_LENGTH];
    for (i = 0; i < numProductions; i++) {
        for (j = i + 1; j < numProductions; j++) {
            if (productions[j].lhs == productions[i].lhs) {
                int prefixLength = 0;
                while (prefixLength < strlen(productions[i].rhs) && prefixLength < strlen(productions[j].rhs) &&
                    productions[i].rhs[prefixLength] == productions[j].rhs[prefixLength]) {
                    prefixLength++;
                }
                if (prefixLength > 0) {
                    newProduction[0] = productions[i].lhs;
                    newProduction[1] = '\'';
                    newProduction[2] = '\0';
                    printf("%c -> %s%c'\n", productions[i].lhs, productions[i].rhs, productions[i].lhs);
                    for (k = 0; k < numProductions; k++) {
                        if (productions[k].lhs == productions[i].lhs) {
                            strcpy(temp, productions[k].rhs + prefixLength);
                            printf("%c' -> %s | ", newProduction[0], temp);
                        }
                    }
                    printf("epsilon\n");
                }
            }
        }
    }
}

int main() {
    Production productions[] = {
        {'E', "TX"},
        {'E', "TY"}
    };
    int numProductions = sizeof(productions) / sizeof(productions[0]);
    printf("Original Grammar:\n");
    for (int i = 0; i < numProductions; i++) {
        printf("%c -> %s\n", productions[i].lhs, productions[i].rhs);
    }
    printf("\nGrammar after eliminating left factoring:\n");
    eliminateLeftFactoring(productions, numProductions);
    return 0;
}
```

```
Original Grammar:
E -> TX
E -> TY

Grammar after eliminating left factoring:
E -> TXE'
E' -> X | E' -> Y | epsilon


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

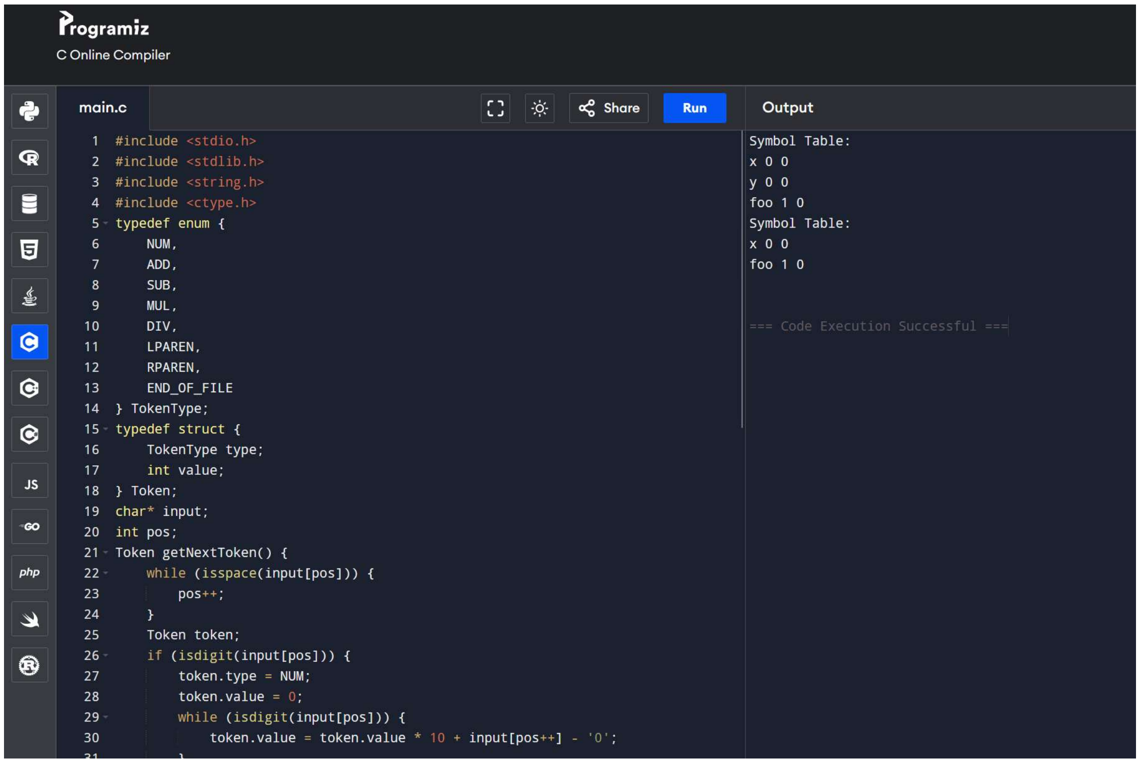
## 8. Implement a C program to perform symbol table operations.



```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_SYMBOLS 100
#define MAX_NAME_LENGTH 20
typedef struct {
    char name[MAX_NAME_LENGTH];
    int type;
    int scope;
} Symbol;
typedef struct {
    Symbol symbols[MAX_SYMBOLS];
    int numSymbols;
} SymbolTable;
SymbolTable* createSymbolTable() {
    SymbolTable* table = (SymbolTable*) malloc(sizeof(SymbolTable));
    table->numSymbols = 0;
    return table;
}
void insertSymbol(SymbolTable* table, char* name, int type, int scope) {
    if (table->numSymbols < MAX_SYMBOLS) {
        strcpy(table->symbols[table->numSymbols].name, name);
        table->symbols[table->numSymbols].type = type;
        table->symbols[table->numSymbols].scope = scope;
        table->numSymbols++;
    } else {
        printf("Symbol table is full\n");
    }
}
int searchSymbol(SymbolTable* table, char* name) {
    for (int i = 0; i < table->numSymbols; i++) {
        if (strcmp(table->symbols[i].name, name) == 0) {
            return i;
        }
    }
    return -1;
}
void deleteSymbol(SymbolTable* table, char* name) {
    int index = searchSymbol(table, name);
    if (index != -1) {
        for (int i = index; i < table->numSymbols - 1; i++) {
            table->symbols[i] = table->symbols[i + 1];
        }
        table->numSymbols--;
    } else {
        printf("Symbol not found\n");
    }
}
void printSymbolTable(SymbolTable* table) {
    printf("Symbol Table:\n");
    for (int i = 0; i < table->numSymbols; i++) {
        printf("%s %d %d\n", table->symbols[i].name, table->symbols[i].type, table->symbols[i].scope);
    }
}
int main() {
    SymbolTable* table = createSymbolTable();
    insertSymbol(table, "x", 0, 0);
    insertSymbol(table, "y", 0, 0);
    insertSymbol(table, "foo", 1, 0);
    printSymbolTable(table);
    deleteSymbol(table, "y");
    printSymbolTable(table);
    return 0;
}
```

```
Symbol Table:
x 0 0
y 0 0
foo 1 0
Symbol Table:
x 0 0
foo 1 0


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

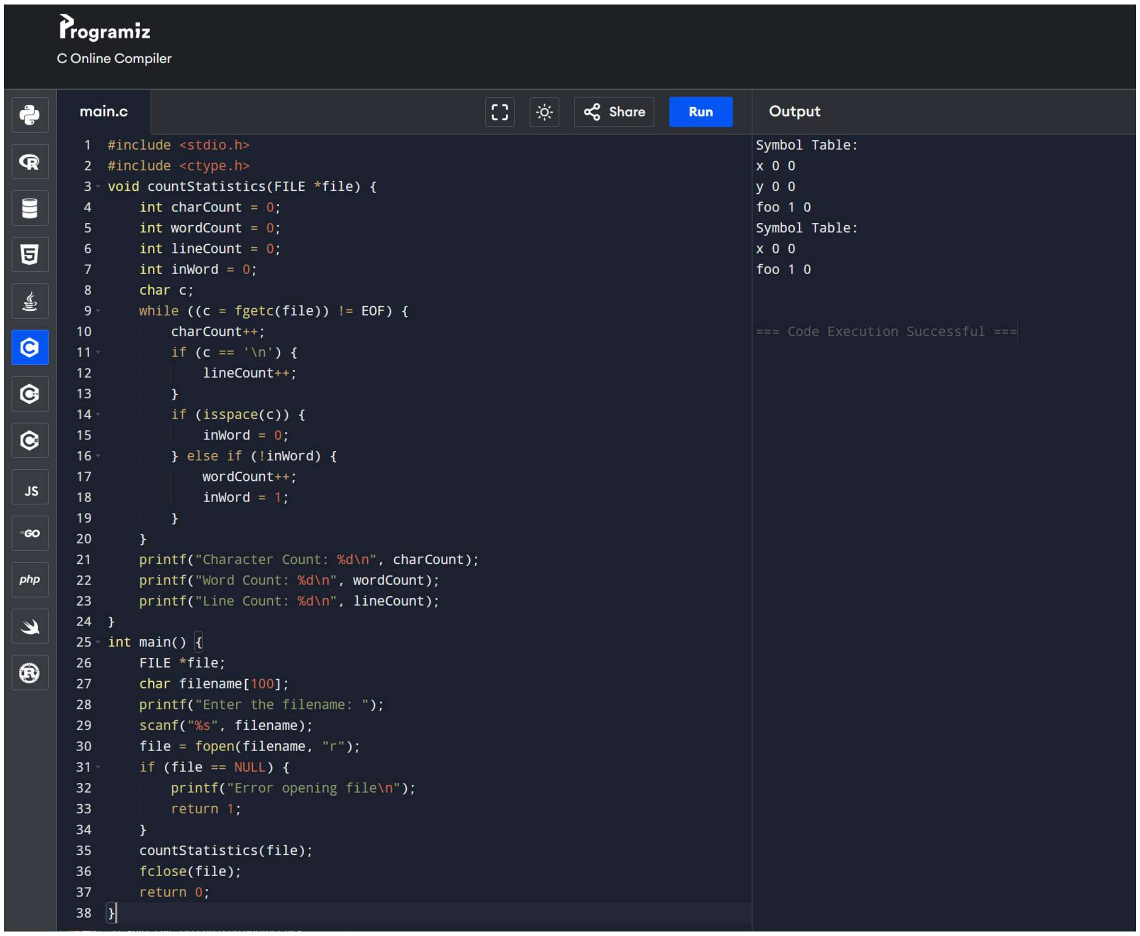
## 9. Write a C program to construct recursive descent parsing.



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
typedef enum {
    NUM,
    ADD,
    SUB,
    MUL,
    DIV,
    LPAREN,
    RPAREN,
    END_OF_FILE
} TokenType;
typedef struct {
    TokenType type;
    int value;
} Token;
char* input;
int pos;
Token getNextToken() {
    while (isspace(input[pos])) {
        pos++;
    }
    Token token;
    if (isdigit(input[pos])) {
        token.type = NUM;
        token.value = 0;
        while (isdigit(input[pos])) {
            token.value = token.value * 10 + input[pos++] - '0';
        }
```

```
Symbol Table:
x 0 0
y 0 0
foo 1 0
Symbol Table:
x 0 0
foo 1 0


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

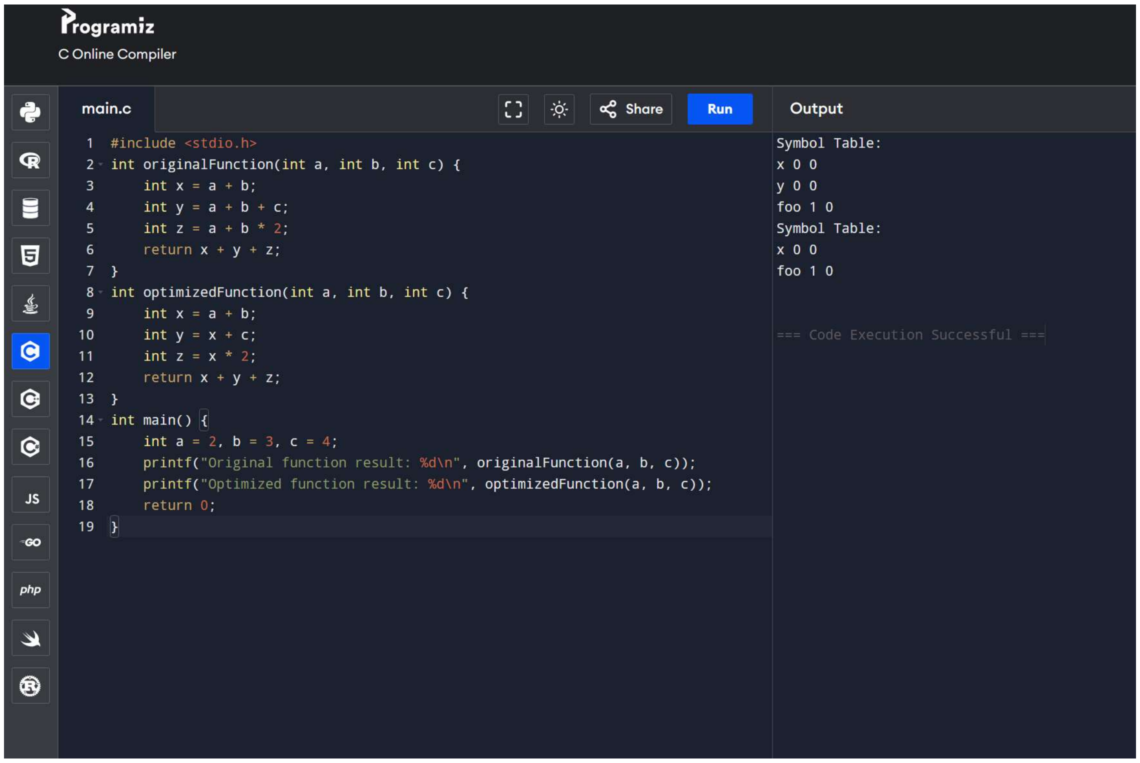
## 13.Write a C program for implementing a Lexical Analyzer to Count the number of characters, words, and lines .



```c
#include <stdio.h>
#include <ctype.h>
void countStatistics(FILE *file) {
    int charCount = 0;
    int wordCount = 0;
    int lineCount = 0;
    int inWord = 0;
    char c;
    while ((c = fgetc(file)) != EOF) {
        charCount++;
        if (c == '\n') {
            lineCount++;
        }
        if (isspace(c)) {
            inWord = 0;
        } else if (!inWord) {
            wordCount++;
            inWord = 1;
        }
    }
    printf("Character Count: %d\n", charCount);
    printf("Word Count: %d\n", wordCount);
    printf("Line Count: %d\n", lineCount);
}
int main() {
    FILE *file;
    char filename[100];
    printf("Enter the filename: ");
    scanf("%s", filename);
    file = fopen(filename, "r");
    if (file == NULL) {
        printf("Error opening file\n");
        return 1;
    }
    countStatistics(file);
    fclose(file);
    return 0;
}
```

```
Symbol Table:
x 0 0
y 0 0
foo 1 0
Symbol Table:
x 0 0
foo 1 0


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

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



```c
#include <stdio.h>
int originalFunction(int a, int b, int c) {
    int x = a + b;
    int y = a + b + c;
    int z = a + b * 2;
    return x + y + z;
}
int optimizedFunction(int a, int b, int c) {
    int x = a + b;
    int y = x + c;
    int z = x * 2;
    return x + y + z;
}
int main() {
    int a = 2, b = 3, c = 4;
    printf("Original function result: %d\n", originalFunction(a, b, c));
    printf("Optimized function result: %d\n", optimizedFunction(a, b, c));
    return 0;
}
```

```
Symbol Table:
x 0 0
y 0 0
foo 1 0
Symbol Table:
x 0 0
foo 1 0


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