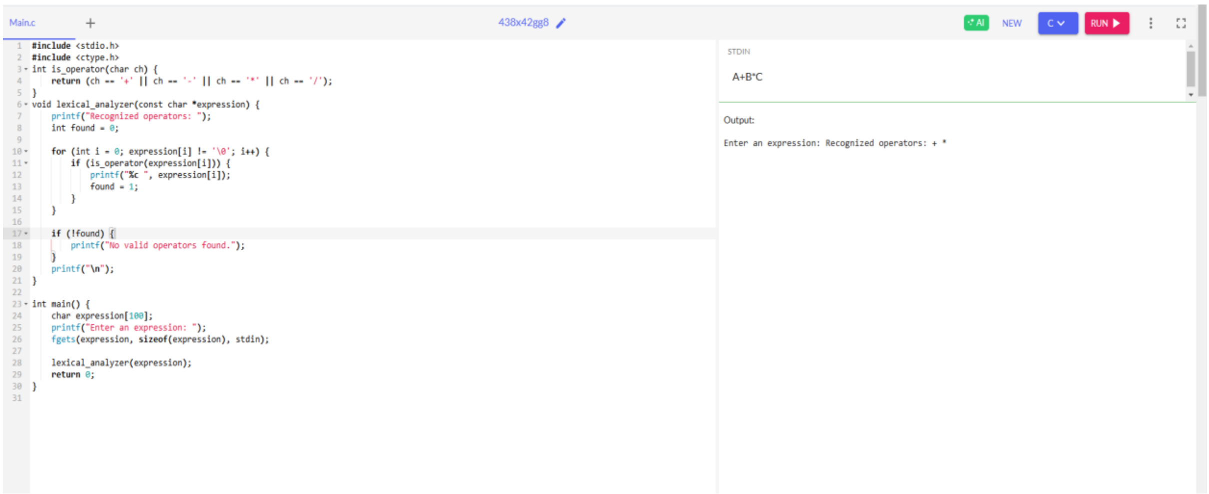
1. Design a lexical Analyzer to validate operators to recognize the operators +,-,*,/ using regular Arithmetic operators .

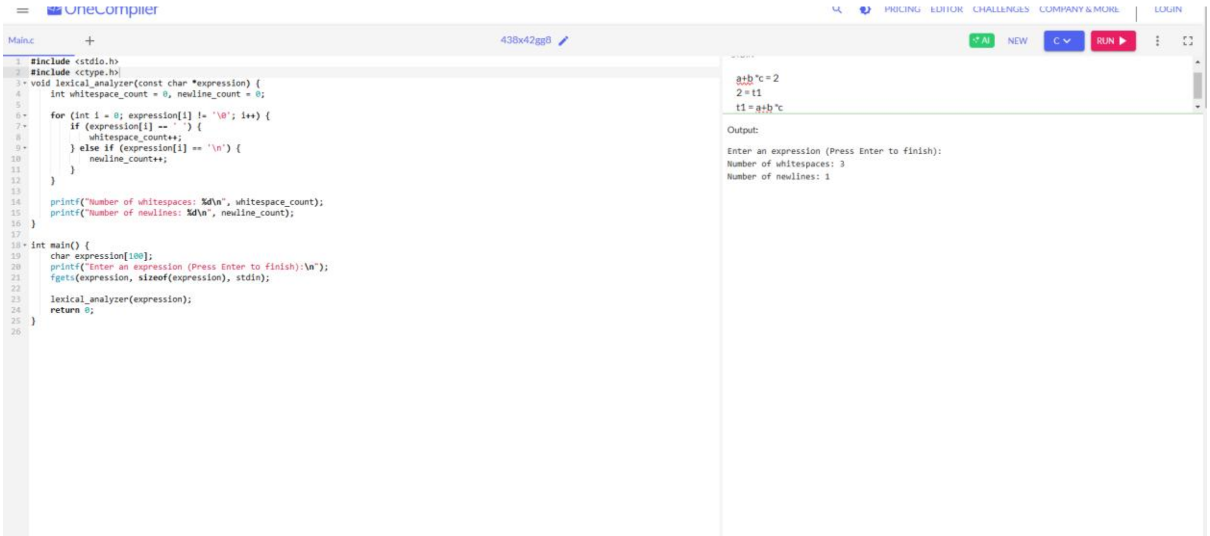


```c
#include <stdio.h>
#include <ctype.h>
int is_operator(char ch) {
    return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
}
void lexical_analyzer(const char *expression) {
    printf("Recognized operators: ");
    int found = 0;

    for (int i = 0; expression[i] != '\0'; i++) {
        if (is_operator(expression[i])) {
            printf("%c ", expression[i]);
            found = 1;
        }
    }

    if (!found) {
        printf("No valid operators found.");
    }
    printf("\n");
}

int main() {
    char expression[100];
    printf("Enter an expression: ");
    fgets(expression, sizeof(expression), stdin);

    lexical_analyzer(expression);
    return 0;
}
```

```
A+B*C
```

```
Enter an expression: Recognized operators: + *
```

2. Design a lexical Analyzer to find the number of whitespaces and newline characters.

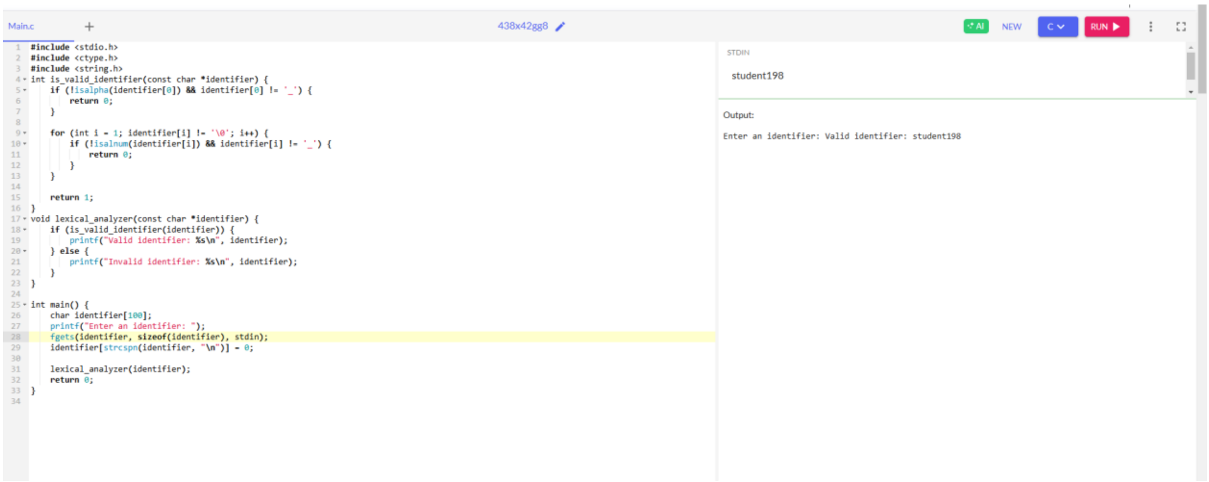


```c
#include <stdio.h>
#include <ctype.h>
void lexical_analyzer(const char *expression) {
    int whitespace_count = 0, newline_count = 0;

    for (int i = 0; expression[i] != '\0'; i++) {
        if (expression[i] == ' ') {
            whitespace_count++;
        } else if (expression[i] == '\n') {
            newline_count++;
        }
    }

    printf("Number of whitespaces: %d\n", whitespace_count);
    printf("Number of newlines: %d\n", newline_count);
}

int main() {
    char expression[100];
    printf("Enter an expression (Press Enter to finish):\n");
    fgets(expression, sizeof(expression), stdin);

    lexical_analyzer(expression);
    return 0;
}
```

```
a+b *c = 2
2 = t1
t1 = a+b *c
```

```
Enter an expression (Press Enter to finish):
Number of whitespaces: 3
Number of newlines: 1
```

3. Develop a lexical Analyzer to test whether a given identifier is valid or not.

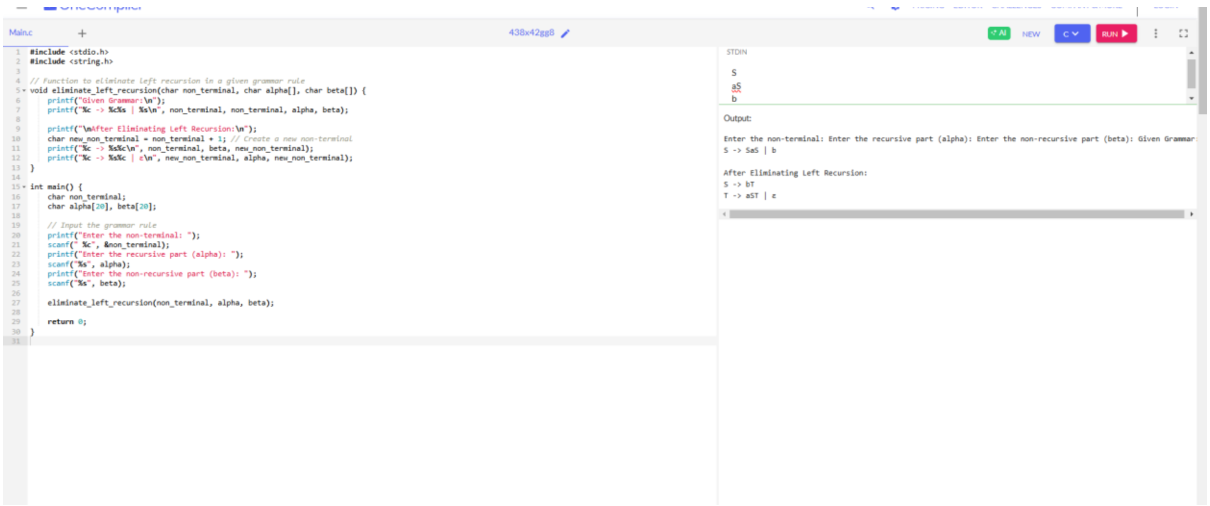


```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>
int is_valid_identifier(const char *identifier) {
    if (!isalpha(identifier[0]) && identifier[0] != '_') {
        return 0;
    }

    for (int i = 1; identifier[i] != '\0'; i++) {
        if (!isalnum(identifier[i]) && identifier[i] != '_') {
            return 0;
        }
    }

    return 1;
}
void lexical_analyzer(const char *identifier) {
    if (is_valid_identifier(identifier)) {
        printf("Valid identifier: %s\n", identifier);
    } else {
        printf("Invalid identifier: %s\n", identifier);
    }
}

int main() {
    char identifier[100];
    printf("Enter an identifier: ");
    fgets(identifier, sizeof(identifier), stdin);
    identifier[strcspn(identifier, "\n")] = 0;

    lexical_analyzer(identifier);
    return 0;
}
```

```
student198
```

```
Enter an identifier: Valid identifier: student198
```

4. Implement a C program to eliminate left recursion.

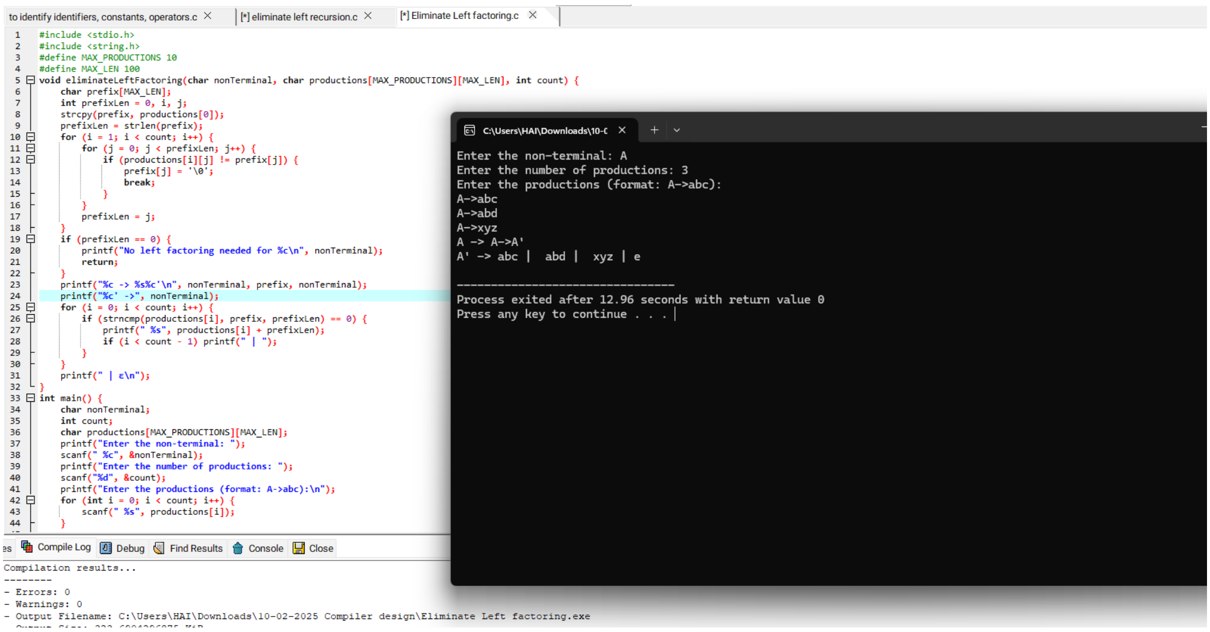


```c
#include <stdio.h>
#include <string.h>

// Function to eliminate left recursion in a given grammar rule
void eliminate_left_recursion(char non_terminal, char alpha[], char beta[]) {
    printf("Given Grammar:\n");
    printf("%c -> %c%s | %s\n", non_terminal, non_terminal, alpha, beta);

    printf("\nAfter Eliminating Left Recursion:\n");
    char new_non_terminal = non_terminal + 1; // Create a new non-terminal
    printf("%c -> %s%c\n", non_terminal, beta, new_non_terminal);
    printf("%c -> %s%c | ε\n", new_non_terminal, alpha, new_non_terminal);
}

int main() {
    char non_terminal;
    char alpha[20], beta[20];

    // Input the grammar rule
    printf("Enter the non-terminal: ");
    scanf(" %c", &non_terminal);
    printf("Enter the recursive part (alpha): ");
    scanf("%s", alpha);
    printf("Enter the non-recursive part (beta): ");
    scanf("%s", beta);

    eliminate_left_recursion(non_terminal, alpha, beta);

    return 0;
}
```

```
S
aS
b
```

```
Enter the non-terminal: Enter the recursive part (alpha): Enter the non-recursive part (beta): Given Grammar
S -> SaS | b

After Eliminating Left Recursion:
S -> bT
T -> aST | ε
```

5. Implement a C program to eliminate left factoring.

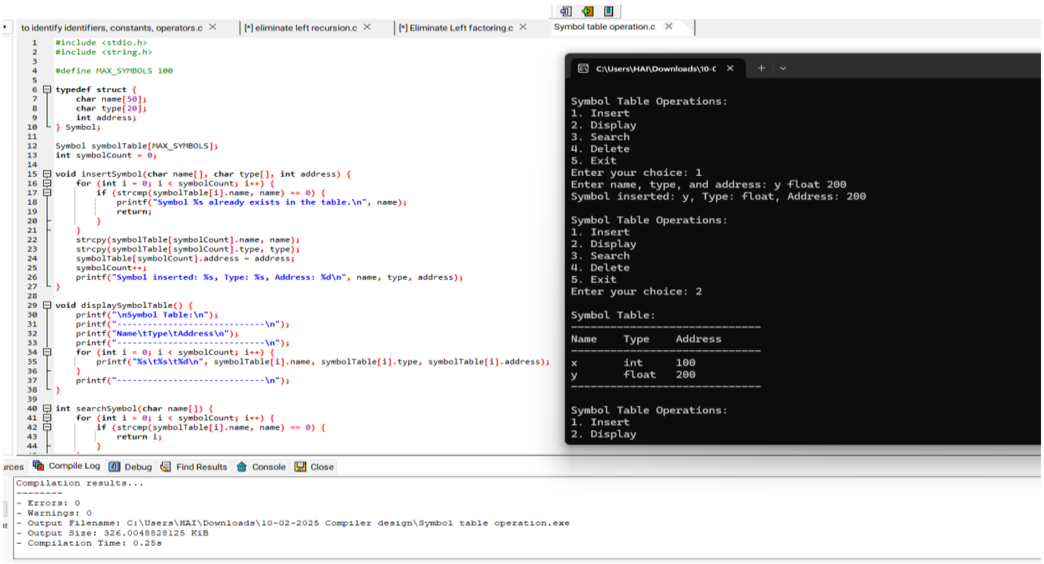

to identify identifiers, constants, operators.c
[*] eliminate left recursion.c
[*] Eliminate Left factoring.c
C:\Users\HAI\Downloads\10-0
es
Compile Log
Debug
Find Results
Console
Close

```c
#include <stdio.h>
#include <string.h>
#define MAX_PRODUCTIONS 10
#define MAX_LEN 100
void eliminateLeftFactoring(char nonTerminal, char productions[MAX_PRODUCTIONS][MAX_LEN], int count) {
    char prefix[MAX_LEN];
    int prefixLen = 0, i, j;
    strcpy(prefix, productions[0]);
    prefixLen = strlen(prefix);
    for (i = 1; i < count; i++) {
        for (j = 0; j < prefixLen; j++) {
            if (productions[i][j] != prefix[j]) {
                prefix[j] = '\0';
                break;
            }
        }
        prefixLen = j;
    }
    if (prefixLen == 0) {
        printf("No left factoring needed for %c\n", nonTerminal);
        return;
    }
    printf("%c -> %s%c'\n", nonTerminal, prefix, nonTerminal);
    printf("%c' ->", nonTerminal);
    for (i = 0; i < count; i++) {
        if (strncmp(productions[i], prefix, prefixLen) == 0) {
            printf(" %s", productions[i] + prefixLen);
            if (i < count - 1) printf(" | ");
        }
    }
    printf(" | ε\n");
}
int main() {
    char nonTerminal;
    int count;
    char productions[MAX_PRODUCTIONS][MAX_LEN];
    printf("Enter the non-terminal: ");
    scanf(" %c", &nonTerminal);
    printf("Enter the number of productions: ");
    scanf("%d", &count);
    printf("Enter the productions (format: A->abc):\n");
    for (int i = 0; i < count; i++) {
        scanf(" %s", productions[i]);
    }
```

```
Enter the non-terminal: A
Enter the number of productions: 3
Enter the productions (format: A->abc):
A->abc
A->abd
A->xyz
A -> A->A'
A' -> abc |  abd |  xyz | e

--------------------------------
Process exited after 12.96 seconds with return value 0
Press any key to continue . . .
```

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\HAI\Downloads\10-02-2025 Compiler design\Eliminate Left factoring.exe
```

6. Implement a C program to perform symbol table operations


to identify identifiers, constants, operators.c
[*] eliminate left recursion.c
[*] Eliminate Left factoring.c
Symbol table operation.c
C:\Users\HAI\Downloads\10-0
urces
Compile Log
Debug
Find Results
Console
Close

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

#define MAX_SYMBOLS 100

typedef struct {
    char name[50];
    char type[20];
    int address;
} Symbol;

Symbol symbolTable[MAX_SYMBOLS];
int symbolCount = 0;

void insertSymbol(char name[], char type[], int address) {
    for (int i = 0; i < symbolCount; i++) {
        if (strcmp(symbolTable[i].name, name) == 0) {
            printf("Symbol %s already exists in the table.\n", name);
            return;
        }
    }
    strcpy(symbolTable[symbolCount].name, name);
    strcpy(symbolTable[symbolCount].type, type);
    symbolTable[symbolCount].address = address;
    symbolCount++;
    printf("Symbol inserted: %s, Type: %s, Address: %d\n", name, type, address);
}

void displaySymbolTable() {
    printf("\nSymbol Table:\n");
    printf("----------------------------\n");
    printf("Name\tType\tAddress\n");
    printf("----------------------------\n");
    for (int i = 0; i < symbolCount; i++) {
        printf("%s\t%s\t%d\n", symbolTable[i].name, symbolTable[i].type, symbolTable[i].address);
    }
    printf("----------------------------\n");
}

int searchSymbol(char name[]) {
    for (int i = 0; i < symbolCount; i++) {
        if (strcmp(symbolTable[i].name, name) == 0) {
            return i;
        }
```

```
Symbol Table Operations:
1. Insert
2. Display
3. Search
4. Delete
5. Exit
Enter your choice: 1
Enter name, type, and address: y float 200
Symbol inserted: y, Type: float, Address: 200

Symbol Table Operations:
1. Insert
2. Display
3. Search
4. Delete
5. Exit
Enter your choice: 2

Symbol Table:
-----------------------------
Name    Type    Address
-----------------------------
x       int     100
y       float   200
-----------------------------

Symbol Table Operations:
1. Insert
2. Display
```

```
Compilation results...
--------
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\HAI\Downloads\10-02-2025 Compiler design\Symbol table operation.exe
- Output Size: 326.0048828125 KiB
- Compilation Time: 0.25s
```