

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

The screenshot shows a C++ IDE with a file named 'Main.cpp'. The code defines a function `is_operator` that checks if a character is one of '+', '-', '*', or '/'. It also defines a function `lexical_analyzer` that iterates through a string and prints out any recognized operators. The `main` function prompts the user to enter an expression and calls `lexical_analyzer`. The output shows the input 'A+B*C' and the recognized operators '+ *'.

```
1 #include <stdio.h>
2 #include <ctype.h>
3 int is_operator(char ch) {
4     return (ch == '+' || ch == '-' || ch == '*' || ch == '/');
5 }
6 void lexical_analyzer(const char *expression) {
7     printf("Recognized operators: ");
8     int found = 0;
9
10    for (int i = 0; expression[i] != '\0'; i++) {
11        if (is_operator(expression[i])) {
12            printf("%c ", expression[i]);
13            found = 1;
14        }
15    }
16
17    if (!found) {
18        printf("No valid operators found.");
19    }
20    printf("\n");
21 }
22
23 int main() {
24     char expression[100];
25     printf("Enter an expression: ");
26     fgets(expression, sizeof(expression), stdin);
27
28     lexical_analyzer(expression);
29     return 0;
30 }
```

Output:

Enter an expression: Recognized operators: + *

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

The screenshot shows a C++ IDE with a file named 'Main.cpp'. The code defines a function `lexical_analyzer` that counts the number of whitespace characters and newline characters in a string. The `main` function prompts the user to enter an expression and calls `lexical_analyzer`. The output shows the input 'a+b*c=2' and the counts: Number of whitespaces: 3, Number of newlines: 1.

```
1 #include <stdio.h>
2 #include <ctype.h>
3 void lexical_analyzer(const char *expression) {
4     int whitespace_count = 0, newline_count = 0;
5
6     for (int i = 0; expression[i] != '\0'; i++) {
7         if (expression[i] == ' ') {
8             whitespace_count++;
9         } else if (expression[i] == '\n') {
10            newline_count++;
11        }
12    }
13
14    printf("Number of whitespaces: %d\n", whitespace_count);
15    printf("Number of newlines: %d\n", newline_count);
16 }
17
18 int main() {
19     char expression[100];
20     printf("Enter an expression (Press Enter to finish):\n");
21     fgets(expression, sizeof(expression), stdin);
22
23     lexical_analyzer(expression);
24     return 0;
25 }
```

Output:

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.

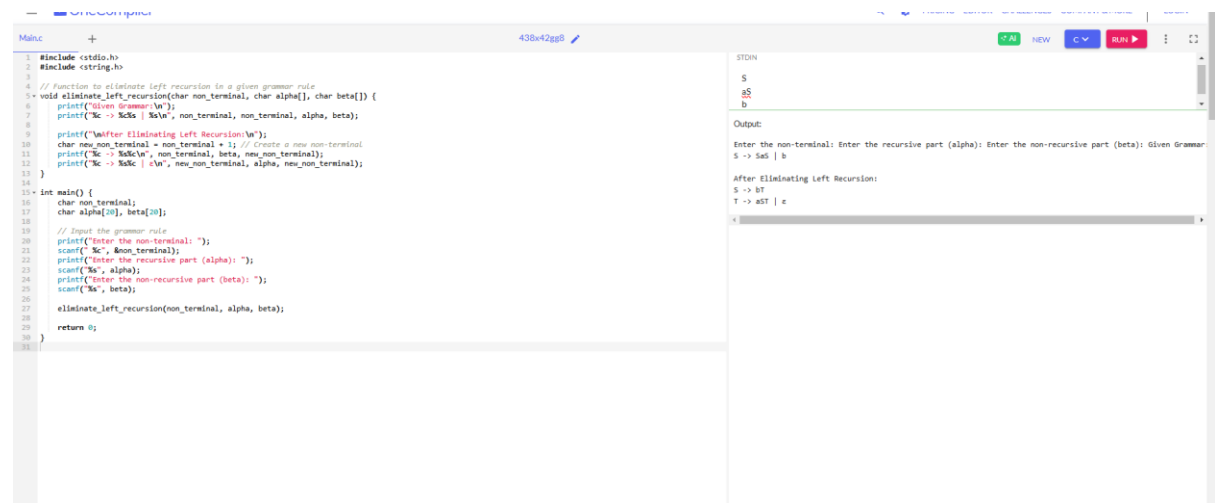
The screenshot shows a C++ IDE with a file named 'Main.cpp'. The code defines a function `is_valid_identifier` that checks if a string is a valid identifier according to C++ rules. It then defines a function `lexical_analyzer` that uses `is_valid_identifier` to check the input. The `main` function prompts the user to enter an identifier and calls `lexical_analyzer`. The output shows the input 'student198' and the result 'Valid identifier: student198'.

```
1 #include <stdio.h>
2 #include <ctype.h>
3 #include <string.h>
4 int is_valid_identifier(const char *identifier) {
5     if (!isalpha(identifier[0]) || identifier[0] == '_' || identifier[0] == '.') {
6         return 0;
7     }
8
9     for (int i = 1; identifier[i] != '\0'; i++) {
10        if (!isalnum(identifier[i]) || identifier[i] == '_' || identifier[i] == '.') {
11            return 0;
12        }
13    }
14
15    return 1;
16 }
17
18 void lexical_analyzer(const char *identifier) {
19     if (is_valid_identifier(identifier)) {
20         printf("Valid identifier: %s\n", identifier);
21     } else {
22         printf("Invalid identifier: %s\n", identifier);
23     }
24 }
25
26 int main() {
27     char identifier[100];
28     printf("Enter an identifier: ");
29     fgets(identifier, sizeof(identifier), stdin);
30     identifier[strcspn(identifier, "\n")] = 0;
31
32     lexical_analyzer(identifier);
33     return 0;
34 }
```

Output:

Enter an identifier: Valid identifier: student198

4. Implement a C program to eliminate left recursion.



```
1 #include <stdio.h>
2 #include <string.h>
3
4 // function to eliminate left recursion in a given grammar rule
5 void eliminate_left_recursion(char non_terminal, char alpha[], char beta[]) {
6     printf("Given Grammar:\n");
7     printf("%c -> %s | %s\n", non_terminal, alpha, beta);
8
9     printf("\nAfter Eliminating Left Recursion:\n");
10    char new_non_terminal = non_terminal + 1; // Create a new non-terminal
11    printf("%c -> %s\n", non_terminal, beta, new_non_terminal);
12    printf("%c -> %s | %s\n", new_non_terminal, alpha, new_non_terminal);
13 }
14
15 int main() {
16     char non_terminal;
17     char alpha[50], beta[50];
18
19     // Input the grammar rule
20     printf("Enter the non-terminal: ");
21     scanf("%c", &non_terminal);
22     printf("Enter the recursive part (alpha): ");
23     scanf("%s", alpha);
24     printf("Enter the non-recursive part (beta): ");
25     scanf("%s", beta);
26
27     eliminate_left_recursion(non_terminal, alpha, beta);
28
29     return 0;
30 }
```

STDIN

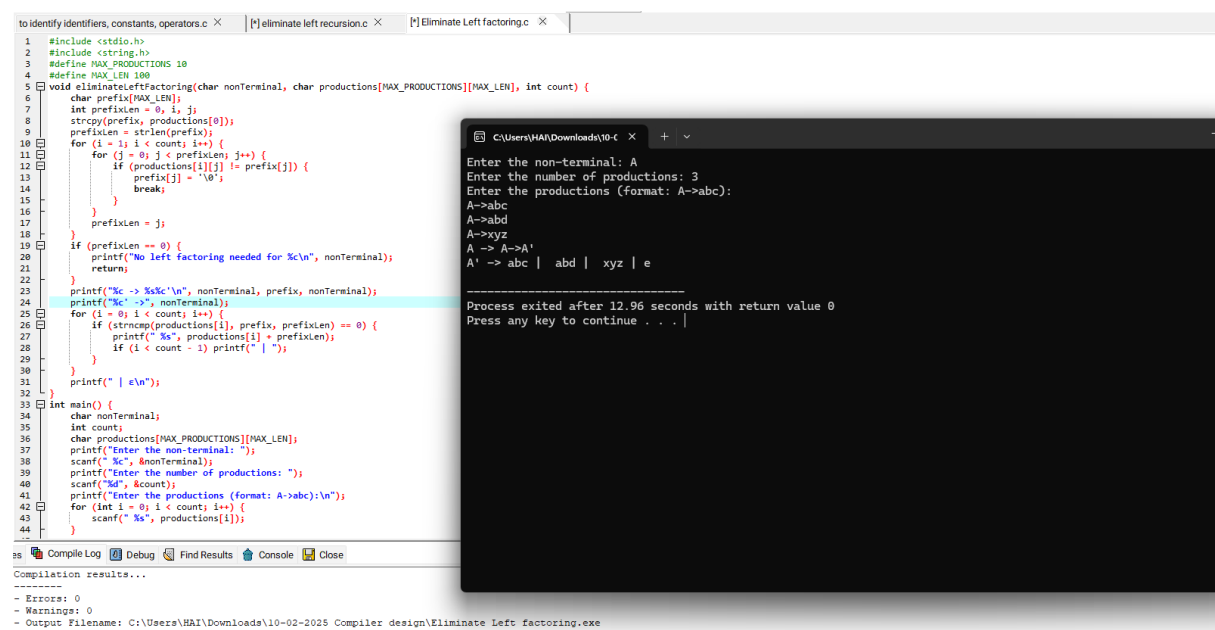
```
S
aS
b
```

Output:

```
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.



```
1 #include <stdio.h>
2 #include <string.h>
3 #define MAX_PRODUCTIONS 10
4 #define MAX_LEN 100
5 void eliminate_left_factoring(char nonTerminal, char productions[MAX_PRODUCTIONS][MAX_LEN], int count) {
6     char prefix[MAX_LEN];
7     int prefixlen = 0, i, j;
8     strcpy(prefix, productions[0]);
9     prefixlen = strlen(prefix);
10    for (i = 1; i < count; i++) {
11        for (j = 0; j < prefixlen; j++) {
12            if (productions[i][j] != prefix[j]) {
13                prefix[j] = '\0';
14                break;
15            }
16        }
17        prefixlen = j;
18    }
19    if (prefixlen == 0) {
20        printf("No left factoring needed for %c\n", nonTerminal);
21        return;
22    }
23    printf("%c -> %s\n", nonTerminal, prefix, nonTerminal);
24    printf("%c -> ", nonTerminal);
25    for (i = 0; i < count; i++) {
26        if (strcmp(productions[i], prefix, prefixlen) == 0) {
27            printf("%s", productions[i] + prefixlen);
28            if (i < count - 1) printf(" | ");
29        }
30    }
31    printf("\n");
32 }
33 int main() {
34     char nonTerminal;
35     int count;
36     char productions[MAX_PRODUCTIONS][MAX_LEN];
37     printf("Enter the non-terminal: ");
38     scanf("%c", &nonTerminal);
39     printf("Enter the number of productions: ");
40     scanf("%d", &count);
41     printf("Enter the productions (format: A->abc):\n");
42     for (int i = 0; i < count; i++) {
43         scanf("%s", productions[i]);
44     }
45 }
```

to identify identifiers, constants, operators.c | eliminate left recursion.c | Eliminate Left factoring.c

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 ×
```

```
1 #include <stdio.h>
2 #include <string.h>
3
4 #define MAX_SYMBOLS 100
5
6 typedef struct {
7     char name[50];
8     char type[50];
9     int address;
10 } Symbol;
11
12 Symbol symbolTable[MAX_SYMBOLS];
13 int symbolCount = 0;
14
15 void insertSymbol(char name[], char type[], int address) {
16     for (int i = 0; i < symbolCount; i++) {
17         if (strcmp(symbolTable[i].name, name) == 0) {
18             printf("Symbol %s already exists in the table.\n", name);
19             return;
20         }
21     }
22     strcpy(symbolTable[symbolCount].name, name);
23     strcpy(symbolTable[symbolCount].type, type);
24     symbolTable[symbolCount].address = address;
25     symbolCount++;
26     printf("Symbol inserted: %s, Type: %s, Address: %d\n", name, type, address);
27 }
28
29 void displaySymbolTable() {
30     printf("\nSymbol Table:\n");
31     printf("-----\n");
32     printf("Name\tType\tAddress\n");
33     printf("-----\n");
34     for (int i = 0; i < symbolCount; i++) {
35         printf("%s\t%s\t%d\n", symbolTable[i].name, symbolTable[i].type, symbolTable[i].address);
36     }
37     printf("-----\n");
38 }
39
40 int searchSymbol(char name[]) {
41     for (int i = 0; i < symbolCount; i++) {
42         if (strcmp(symbolTable[i].name, name) == 0) {
43             return i;
44         }
45     }
46     return -1;
47 }
```

```
C:\Users\HAI\Downloads\10-2-2025 Compiler design\Symbol table operation.exe
```

```
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
```