1.Develop a lexical Analyzer to identify identifiers, constants, operators using C program.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAX_IDENTIFIER_LENGTH 100
void analyze(const char *input) { char identifier[MAX_IDENTIFIER_LENGTH]; int i = 0, j =
0;
while (input[i] != '\0')
{
   if (isspace(input[i]))
   {
      i++;
      continue;
   }
   if (input[i] == '/' && input[i + 1] == '/')
   {
      while (input[i] != '\n' && input[i] != '\0') i++;
      continue;
   }
   if (isalpha(input[i]))
   {
      j = 0;
      while (isalnum(input[i]) && j < MAX_IDENTIFIER_LENGTH - 1)
 {
```

```
            identifier[j++] = input[i++];
        }
        identifier[j] = '\0';
        printf("Identifier: %s\n", identifier);
        continue;
    }
    if (isdigit(input[i]))
    {
        j = 0;
        while (isdigit(input[i]) && j < MAX_IDENTIFIER_LENGTH - 1)
        {
            identifier[j++] = input[i++];
        }
        identifier[j] = '\0';
        printf("Constant: %s\n", identifier);
        continue;
    }
    if (strchr("+-*/=<>!", input[i]))
    {
        printf("Operator: %c\n", input[i]);
        i++;
        continue;
    }
    i++;
}
```

```
}
int main()
 {
const char *code = "int x = 10;
// This is a comment\nfloat y = 20.5;"; analyze(code); return 0; }
```

Output:

```
Identifier: int
Identifier: x
Operator: =
Constant: 10
Identifier: float
Identifier: y
Operator: =
Constant: 20
Constant: 5
```

2. Develop a lexical Analyzer to identify whether a given line is a comment or not.

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

void checkComment(const char *line)
{
    if (strstr(line, "//") != NULL)
```

```
    {
        printf("Single-line comment detected: %s\n", line);
    }
    else if (strstr(line, "/*") != NULL && strstr(line, "*/") != NULL)
    {
        printf("Multi-line comment detected: %s\n", line);
    }
    else
    {
        printf("No comment detected: %s\n", line);
    }
}
int main()
{
    const char *lines[] = {
        "int main() { // This is a comment",
        "printf(\"Hello, World!\"); /* This is a multi-line comment */",
        "return 0;",
        "/* Start of comment\n   Still in comment */"
    };
    for (int i = 0; i < 4; i++)
    {
        checkComment(lines[i]);
    }
    return 0;
```

}

Output:

```
Single-line comment detected: int main() { // This is a comment
Multi-line comment detected: printf("Hello, World!"); /* This is a multi-line comment */
No comment detected: return 0;
Multi-line comment detected: /* Start of comment
   Still in comment */
```

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

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

void lexicalAnalyzer(const char *input)
{
   for (int i = 0; i < strlen(input); i++)
   {
      if (input[i] == '+' || input[i] == '-' || input[i] == '*' || input[i] == '/')
      {
         printf("Operator found: %c\n", input[i]);
      }
      else if (isspace(input[i]))
      {
         continue;
```

```
        }
        else
        {
            printf("Invalid character: %c\n", input[i]);
        }
    }
}
int main()
{
    const char *expression = "3 + 5 - 2 * 4 / 2";
    lexicalAnalyzer(expression);
    return 0;
}
```

Output:

```
Invalid character: 3
Operator found: +
Invalid character: 5
Operator found: -
Invalid character: 2
Operator found: *
Invalid character: 4
Operator found: /
Invalid character: 2
```

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

```
#include <stdio.h>
int main()
{
    char ch;
```

```
    int whitespace_count = 0;
    int newline_count = 0;
    printf("Enter text (Ctrl+D to end):\n");
    while ((ch = getchar()) != EOF)
    {
        if (ch == ' ' || ch == '\t')
        {
            whitespace_count++;
        } else if (ch == '\n')
        {
            newline_count++;
        }
    }
    printf("Number of whitespace characters: %d\n", whitespace_count);
    printf("Number of newline characters: %d\n", newline_count);
    return 0;
}
```

Output:

```
Invalid character: 3
Operator found: +
Invalid character: 5
Operator found: -
Invalid character: 2
Operator found: *
Invalid character: 4
Operator found: /
Invalid character: 2
```

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

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

int isValidIdentifier(const char *identifier)
{
    if (!isalpha(identifier[0]) && identifier[0] != '_')
    {
        return 0;
    }
    for (int i = 1; i < strlen(identifier); i++)
    {
        if (!isalnum(identifier[i]) && identifier[i] != '_')
        {
            return 0;
        }
    }
    return 1;
}
int main()
{
    const char *testIdentifier = "valid_identifier1";
    if (isValidIdentifier(testIdentifier))
    {
        printf("%s is a valid identifier.\n", testIdentifier);
    }
    else
```

```
    {
        printf("%s is not a valid identifier.\n", testIdentifier);
    }
    return 0;
}
```

Output:

```
valid_identifier1 is a valid identifier.
```

6.Implement a C program to eliminate left recursion.

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

void eliminateLeftRecursion(char *nonTerminal, char productions[][10], int count)
{
    char newNonTerminal[10];
    printf(newNonTerminal, "%s'", nonTerminal);

    printf("New Productions:\n");
    for (int i = 0; i < count; i++) {
        if (productions[i][0] == nonTerminal[0])
        {
            printf("%s -> %s%s\n", newNonTerminal, productions[i] + 1, newNonTerminal);
        }
        else
        {
```

```
            printf("%s -> %s%s\n", nonTerminal, productions[i], newNonTerminal);
        }
    }
    printf("%s -> ε\n", newNonTerminal);
}
int main()
{
    char nonTerminal[] = "A";
    char productions[][10] = {"Aab", "Ac", "A"};
    int count = sizeof(productions) / sizeof(productions[0]);
    eliminateLeftRecursion(nonTerminal, productions, count);
    return 0;
}
```

Output:

```
New Productions:
 -> ab
 -> c
 ->
 -> ε
```

7.Implement a C program to eliminate left factoring.

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

void leftFactor(char productions[][10], int count)
{
    char commonPrefix[10];
```

```
    int i, j;

    for (i = 0; i < count; i++)
    {
        for (j = i + 1; j < count; j++)
        {
            int k = 0;
            while (productions[i][k] == productions[j][k] && productions[i][k] != '\0')
            {
                k++;
            }
            if (k > 0)
            {
                strncpy(commonPrefix, productions[i], k);
                commonPrefix[k] = '\0';
                printf("Left Factored: %s -> %sX\n", productions[i], commonPrefix);
                printf("X -> %s | %s\n", productions[i] + k, productions[j] + k);
            }
        }
    }
}
int main()
{
    char productions[5][10] = {"abc", "abx", "acd", "xyz", "xy"};
    int count = 5;
    leftFactor(productions, count);
    return 0;
```

}

Output:

```
Left Factored: abc -> abX
X -> c | x
Left Factored: abc -> aX
X -> bc | cd
Left Factored: abx -> aX
X -> bx | cd
Left Factored: xyz -> xyX
X -> z |
```