**DAY 4**

**EXPERIMENT-25**: LEX program to count the frequency of the given word in a given sentence.

%{

#include<stdio.h>

#include<string.h>

char word [] = "geeks";

int count = 0;

%}

%%

[a-zA-Z]+ { if(strcmp(yytext, word)==0)

count++; }

. ;

%%

int yywrap()

{

return 1;

}

int main()

{

extern FILE \*yyin, \*yyout;

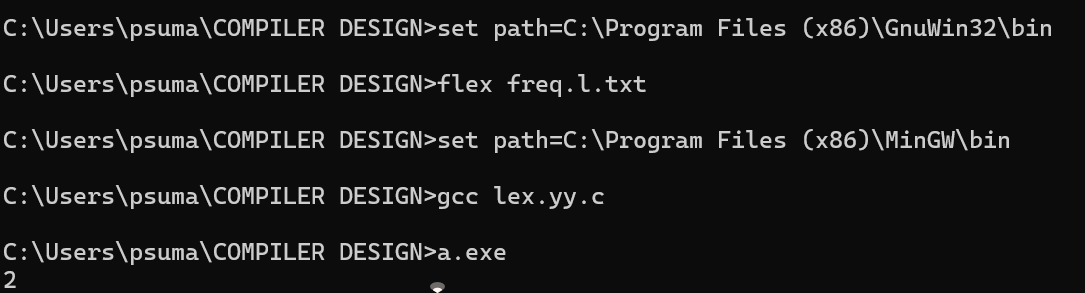
yyin=fopen("input1.txt", "r");

yylex();

printf("%d", count);

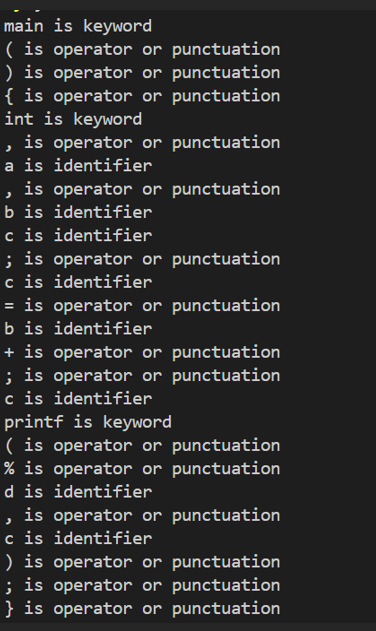
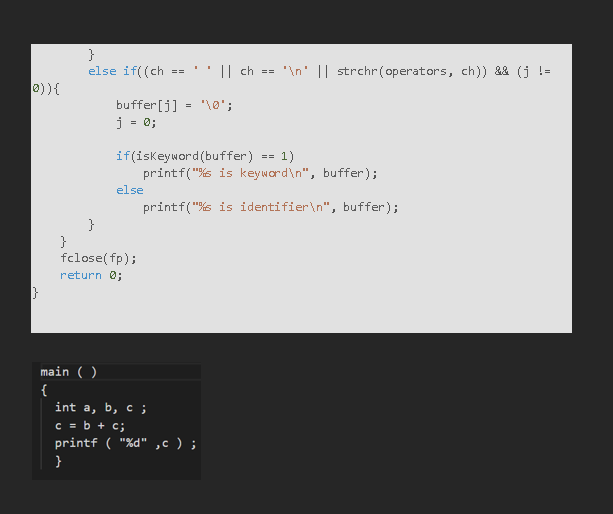
}



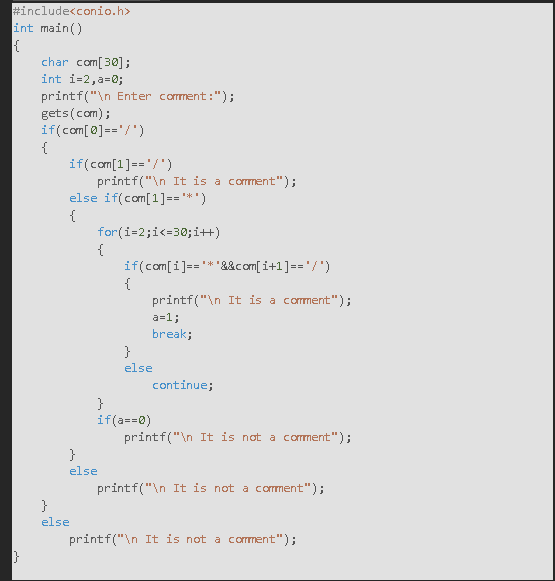


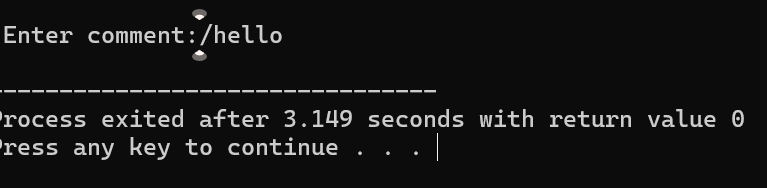
**EXPERIMENT-26**: Develop a lexical Analyzer to identify identifiers, constants, operators using C program.



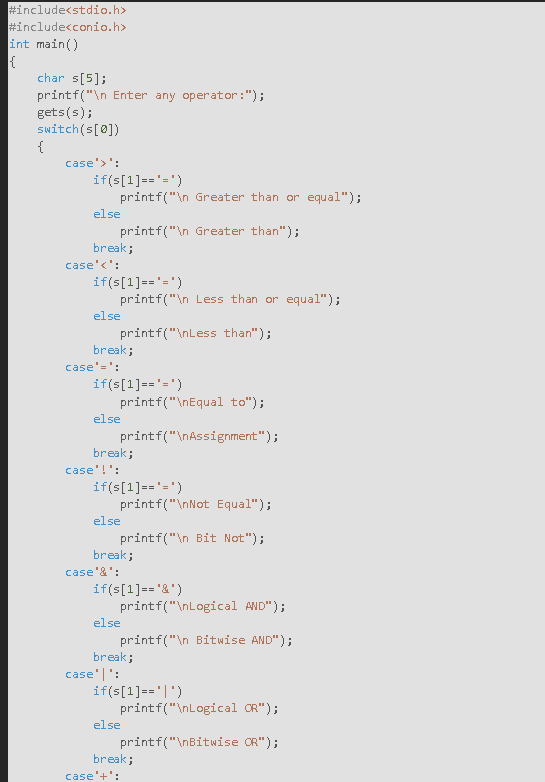


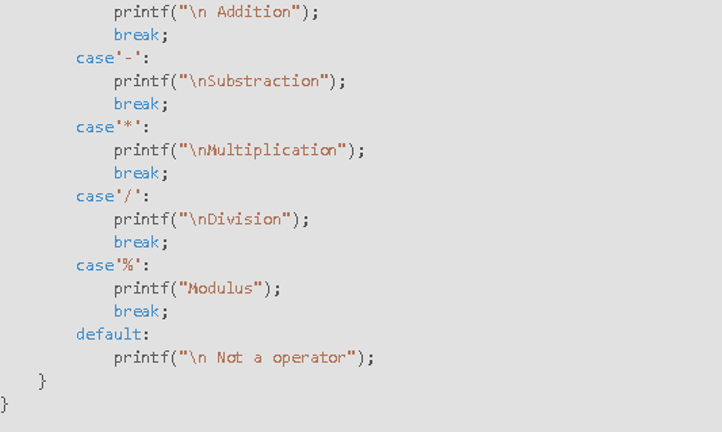
**EXPERIMENT-27**: Develop a lexical Analyzer to identify whether a given line is a comment or not.

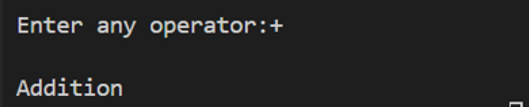




**EXPERIMENT-28**: Design a lexical Analyzer to validate operators to recognize the operators +,-,\*,/ using regular Arithmetic operators .







**EXPERIMENT-29**: Design a lexical Analyzer to find the number of whitespaces and newline characters.

#include <stdio.h>

#include <stdlib.h>

int main() {

    char ch;

    int whiteSpaces = 0, newLines = 0;

    FILE \*fp;

    // Open the file for reading

    fp = fopen("flex\_input.txt", "r");

    if (fp == NULL) {

        printf("Error while opening the file\n");

        exit(0);

    }

    // Read each character from the file

    while ((ch = fgetc(fp)) != EOF) {

        // Check for whitespace characters

        if (ch == ' ' || ch == '\t') {

            whiteSpaces++;

        }

        // Check for newline characters

        else if (ch == '\n') {

            newLines++;

        }

    }

    // Close the file

    fclose(fp);

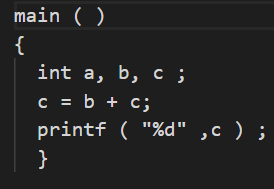
    // Print the results

    printf("Number of whitespaces: %d\n", whiteSpaces);

    printf("Number of newlines: %d\n", newLines);

    return 0;

}





**EXPERIMENT-30**: Develop a lexical Analyzer to test whether a given identifier is valid or not.

#include<stdio.h>

#include<conio.h>

#include<ctype.h>

int main()

{

    char a[10];

    int flag, i=1;

    printf("\n Enter an identifier:");

    gets(a);

    if(isalpha(a[0]))

        flag=1;

    else

        printf("\n Not a valid identifier");

        while(a[i]!='\0')

    {

        if(!isdigit(a[i])&&!isalpha(a[i]))

        {

            flag=0;

            break;

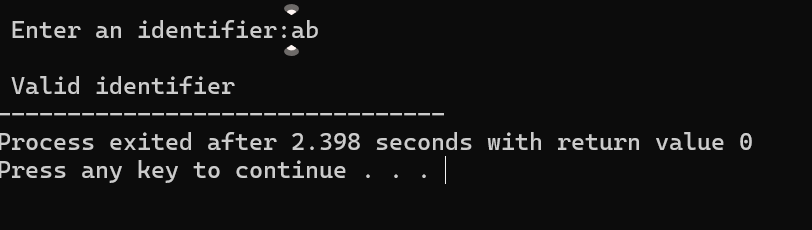
        } i++;

    }

    if(flag==1)

        printf("\n Valid identifier");

}



**EXPERIMENT-31**: Implement a C program to eliminate left recursion.

#include <stdio.h>

#include <string.h>

#define SIZE 10

int main() {

    char non\_terminal;

    char beta[SIZE], alpha[SIZE];

    int num;

    char production[10][SIZE];

    printf("Enter Number of Production: ");

    scanf("%d", &num);

    printf("Enter the grammar as E->E-A :\n");

    for (int i = 0; i < num; i++) {

        scanf("%s", production[i]);

    }

    for (int i = 0; i < num; i++) {

        printf("\nGRAMMAR : : : %s", production[i]);

        non\_terminal = production[i][0];

        int index = 3; // Reset index for each production

        if (non\_terminal == production[i][index]) {

            int alpha\_index = 0;

            int beta\_index = 0;

            index++;

            // Extract alpha

            while (production[i][index] != '\0' && production[i][index] != '|') {

                alpha[alpha\_index++] = production[i][index++];

            }

            alpha[alpha\_index] = '\0';

            // Move past '|'

            if (production[i][index] == '|') {

                index++;

            } else {

                printf(" can't be reduced\n");

                continue;

            }

            // Extract beta

            while (production[i][index] != '\0') {

                beta[beta\_index++] = production[i][index++];

            }

            beta[beta\_index] = '\0';

            printf(" is left recursive.\n");

            printf("Grammar without left recursion:\n");

            printf("%c->%s%c'\n", non\_terminal, beta, non\_terminal);

            printf("%c'->%s%c'|ε\n", non\_terminal, alpha, non\_terminal);

        } else {

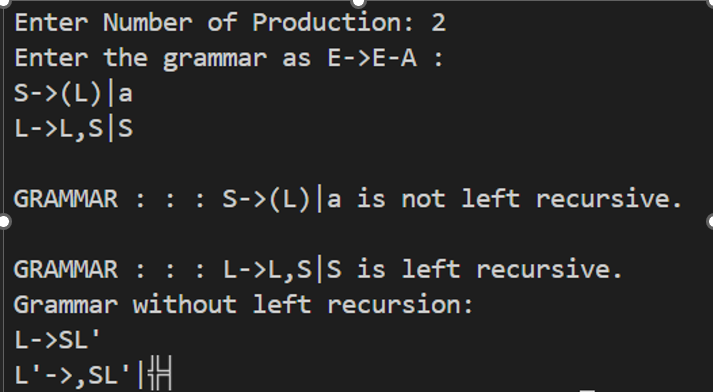
            printf(" is not left recursive.\n");

        }

    }

    return 0;

}



**EXPERIMENT-32**: Implement a C program to eliminate left factoring

#include <stdio.h>

#include <string.h>

int main() {

    char gram[100], \*parts[10], modifiedGram[20], newGram[50];

    int i, j, k = 0, l = 0, pos = 0, num\_productions = 0;

    char \*token;

    printf("Enter Productions (use '|' to separate each production): S->");

    gets(gram);

    // Split the input string by '|'

    token = strtok(gram, "|");

    while (token != NULL) {

        parts[num\_productions++] = token;

        token = strtok(NULL, "|");

    }

    // Find the longest common prefix

    for (i = 0; i < strlen(parts[0]); i++) {

        char current\_char = parts[0][i];

        int match = 1;

        for (j = 1; j < num\_productions; j++) {

            if (parts[j][i] != current\_char) {

                match = 0;

                break;

            }

        }

        if (match) {

            modifiedGram[k++] = current\_char;

            pos = i + 1;

        } else {

            break;

        }

    }

    modifiedGram[k] = 'X';

    modifiedGram[++k] = '\0';

    // Generate the new productions

    for (j = 0; j < num\_productions; j++) {

        if (j > 0) {

            newGram[l++] = '|';

        }

        for (i = pos; parts[j][i] != '\0'; i++) {

            newGram[l++] = parts[j][i];

        }

    }

    newGram[l] = '\0';

    printf("\n S->%s", modifiedGram);

    printf("\n X->%s\n", newGram);

    return 0;

}

****