21. Write a LEX program to count the frequency of the given word in a given sentence

PROGRAM:

```
%{
#include <stdio.h>
#include <string.h>
char word[100];
int count = 0;
%}
%%
([a-zA-Z0-9]+) {
    if (strcmp(yytext, word) == 0) {
        count++;
    }
}
.|\n { } // Ignore other characters
%%
int yywrap() { return 1; }
int main()
{
    printf("Enter the word to count: ");
    scanf("%s", word);
    printf("Enter the sentence: ");
    getchar();  // Clear the newline character left in the buffer
    yylex();
    printf("The word '%s' occurred %d times.\n", word, count);
    return 0;
}
```

OUTPUT:

```
C:\Users\bteja\Downloads\p18>set path=C:\Program Files\GnuWin32\bin

C:\Users\bteja\Downloads\p18>flex countword.l.txt

C:\Users\bteja\Downloads\p18>set path=C:\MinGW\bin

C:\Users\bteja\Downloads\p18>gcc lex.yy.c

C:\Users\bteja\Downloads\p18>a.exe
Enter the word to count: is
Enter the sentence: he is a good dancer and he is a good singer
^Z
The word 'is' occurred 2 times.
```

22. The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Develop a lexical Analyzer to identify identifiers, constants, operators using C program

PROGRAM:

#include <stdio.h>

#include <ctype.h>

#include <string.h>

typedef enum {

   IDENTIFIER,

   CONSTANT,

   OPERATOR,

   UNKNOWN

} TokenType;

int isOperator(char c) {

   return (c == '+' || c == '-' || c == '*' || c == '/' || c == '=');

}

int isIdentifierStart(char c) {

   return (isalpha(c) || c == '_');

}

```c
int isIdentifierChar(char c) {
    return (isalnum(c) || c == '_');
}

int isConstant(char c) {
    return isdigit(c);
}

void printToken(char *token, TokenType type) {
    switch (type) {
        case IDENTIFIER:
            printf("Identifier: %s\n", token);
            break;
        case CONSTANT:
            printf("Constant: %s\n", token);
            break;
        case OPERATOR:
            printf("Operator: %s\n", token);
            break;
        default:
            printf("Unknown: %s\n", token);
            break;
    }
}

void lexer(const char *input) {
    int length = strlen(input);
    char token[100];
    int tokenIndex = 0;
```

```
TokenType currentType = UNKNOWN;

for (int i = 0; i < length; i++) {
    char c = input[i];

    if (isOperator(c)) {
        if (currentType != UNKNOWN) {
            token[tokenIndex] = '\0';
            printToken(token, currentType);
            tokenIndex = 0;
        }
        token[tokenIndex++] = c;
        token[tokenIndex] = '\0';
        printToken(token, OPERATOR);
        tokenIndex = 0;
        currentType = UNKNOWN;
    } else if (isIdentifierStart(c) || isIdentifierChar(c)) {
        if (currentType != IDENTIFIER) {
            if (currentType != UNKNOWN) {
                token[tokenIndex] = '\0';
                printToken(token, currentType);
                tokenIndex = 0;
            }
            currentType = IDENTIFIER;
        }
        token[tokenIndex++] = c;
    } else if (isConstant(c)) {
        if (currentType != CONSTANT) {
            if (currentType != UNKNOWN) {
                token[tokenIndex] = '\0';
```

```c
                printToken(token, currentType);
                tokenIndex = 0;
            }
            currentType = CONSTANT;
        }
        token[tokenIndex++] = c;
    } else if (isspace(c)) {
        if (currentType != UNKNOWN) {
            token[tokenIndex] = '\0';
            printToken(token, currentType);
            tokenIndex = 0;
            currentType = UNKNOWN;
        }
    } else {
        token[tokenIndex++] = c;
        currentType = UNKNOWN;
    }
    }

    if (currentType != UNKNOWN) {
        token[tokenIndex] = '\0';
        printToken(token, currentType);
    }
}

int main() {
    char input[256];
    printf("Enter a string to tokenize: ");
    fgets(input, sizeof(input), stdin);
```
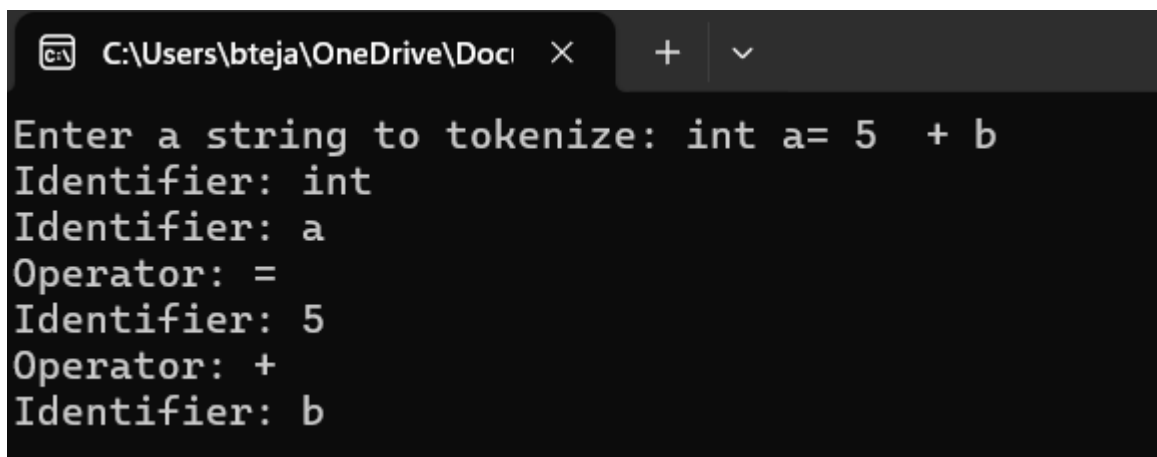
```c
    // Remove newline character from the input if it exists

    size_t len = strlen(input);

    if (len > 0 && input[len - 1] == '\n') {

        input[len - 1] = '\0';

    }


    lexer(input);

    return 0;

}
```

OUTPUT:

22. The lexical analyzer should ignore redundant spaces, tabs and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Develop a lexical Analyzer to identify identifiers, constants, operators using C program

PROGRAM:

```c
#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAX_IDENTIFIER_LENGTH 31

void print_token(const char *token_name, const char *token_value) {

    printf("%s: %s\n", token_name, token_value);

}

int is_operator(char ch) {

    return ch == '+' || ch == '-' || ch == '*' || ch == '/';
```

```c
}
int is_whitespace(char ch) {
    return ch == ' ' || ch == '\t' || ch == '\n' || ch == '\r';
}
void process_identifier_or_keyword(char *buffer, int *index, char *input, int *pos) {
    int length = 0;
    while (isalnum(input[*pos]) && length < MAX_IDENTIFIER_LENGTH) {
        buffer[length++] = input[*pos];
        (*pos)++;
    }
    buffer[length] = '\0';
    if (length == MAX_IDENTIFIER_LENGTH && isalnum(input[*pos])) {
        printf("Identifier too long: %.31s...\n", buffer);
        while (isalnum(input[*pos])) {
            (*pos)++;
        }
    } else {
        print_token("Identifier", buffer);
    }
}
void process_constant(char *buffer, int *index, char *input, int *pos) {
    int length = 0;
    while (isdigit(input[*pos])) {
        buffer[length++] = input[*pos];
        (*pos)++;
    }
    buffer[length] = '\0';
    print_token("Constant", buffer);
}
void process_comment(char *input, int *pos) {
    if (input[*pos] == '/' && input[*pos + 1] == '/') {
```

```c
        while (input[*pos] != '\n' && input[*pos] != '\0') {

            (*pos)++;

        }

    } else if (input[*pos] == '/' && input[*pos + 1] == '*') {

        (*pos) += 2;

        while (input[*pos] != '\0' && !(input[*pos] == '*' && input[*pos + 1] == '/')) {

            (*pos)++;

        }

        if (input[*pos] != '\0') {

            (*pos) += 2;

        }

    }

}

void lexical_analysis(char *input) {

    int pos = 0;

    char buffer[MAX_IDENTIFIER_LENGTH + 1];


    while (input[pos] != '\0') {

        if (isalpha(input[pos])) {

            process_identifier_or_keyword(buffer, NULL, input, &pos);

        } else if (isdigit(input[pos])) {

            process_constant(buffer, NULL, input, &pos);

        } else if (is_operator(input[pos])) {

            buffer[0] = input[pos++];

            buffer[1] = '\0';

            print_token("Operator", buffer);

        } else if (is_whitespace(input[pos])) {

            pos++;

        } else if (input[pos] == '/' && (input[pos + 1] == '/' || input[pos + 1] == '*')) {

            process_comment(input, &pos);

        } else {
```
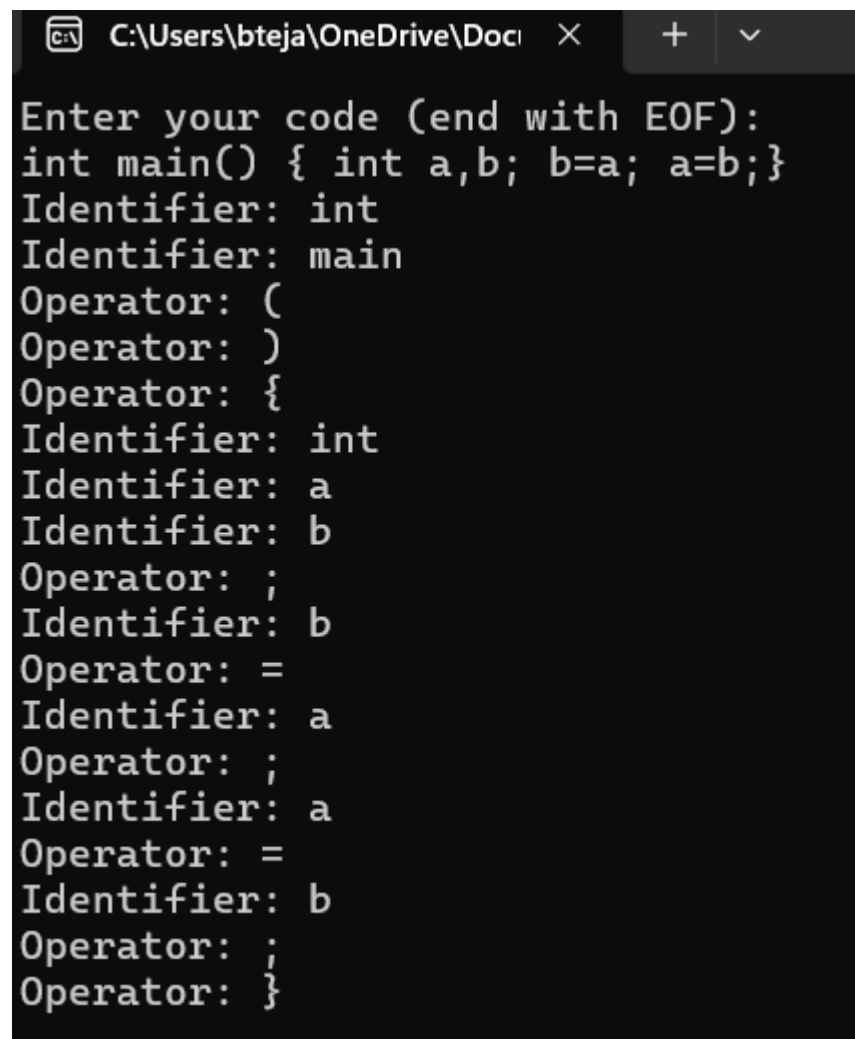
```c
            pos++;
        }
    }
}
int main() {
    char input[1000];


    printf("Enter your code (end with EOF):\n");
    fgets(input, sizeof(input), stdin);
    lexical_analysis(input);
    return 0;
}
```

OUTPUT:


```
Enter your code (end with EOF):
int main() { int a,b; b=a; a=b;}
Identifier: int
Identifier: main
Operator: (
Operator: )
Operator: {
Identifier: int
Identifier: a
Identifier: b
Operator: ;
Identifier: b
Operator: =
Identifier: a
Operator: ;
Identifier: a
Operator: =
Identifier: b
Operator: ;
Operator: }
```

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

PROGRAM:

```
%{
#include <stdio.h>
#include <ctype.h>

%}
%%
[A-Za-z]+      { printf("Word: %s\n", yytext); }
"=="          { printf("Relational Operator: %s\n", yytext); }
"!="          { printf("Relational Operator: %s\n", yytext); }
"<="          { printf("Relational Operator: %s\n", yytext); }
">="          { printf("Relational Operator: %s\n", yytext); }
"<"           { printf("Relational Operator: %s\n", yytext); }
">"           { printf("Relational Operator: %s\n", yytext); }
[ \t\n]+      { /* ignore whitespace */ }
.             { /* ignore other characters */ }
%%
int yywrap() {
   return 1;
}
int main() {
   yylex();
   return 0;
}
```

OUTPUT:

```
C:\Users\bteja\Downloads\p15>set path=C:\Program Files\GnuWin32\bin

C:\Users\bteja\Downloads\p15>flex wordrelop.l.txt

C:\Users\bteja\Downloads\p15>set path=C:\MinGW\bin

C:\Users\bteja\Downloads\p15>gcc lex.yy.c

C:\Users\bteja\Downloads\p15>a.exe
bhanu
Word: bhanu
==
Relational Operator: ==
<=
Relational Operator: <=
>=
Relational Operator: >=
teja
Word: teja
```

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

PROGRAM:

```
%{
#include <stdio.h>

int spaces = 0;
int tabs = 0;
int newlines = 0;
%}
%%
" "      { spaces++; }
"\t"     { tabs++; }
"\n"     { newlines++; }
.        { /* ignore other characters */ }
%%
int yywrap() {
   return 1;
}
int main() {
   yylex();
   printf("Spaces: %d\n", spaces);
```
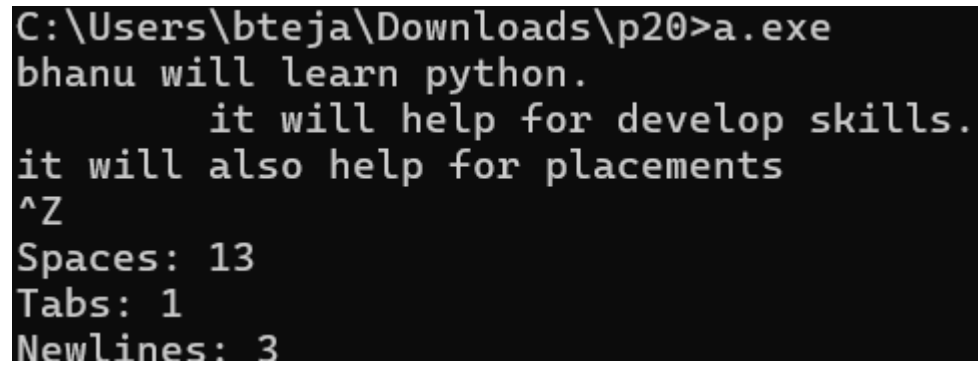
```
    printf("Tabs: %d\n", tabs);

    printf("Newlines: %d\n", newlines);

    return 0;

}
```

OUTPUT:

```
C:\Users\bteja\Downloads\p20>a.exe
bhanu will learn python.
        it will help for develop skills.
it will also help for placements
^Z
Spaces: 13
Tabs: 1
Newlines: 3
```

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

PROGRAM:

```
%{

%}

%%

[a-zA-Z][a-zA-Z0-9]+ {printf("\n is identifier",yytext);}

.+ {printf("\n%s is not identifier",yytext);}

%%

int yywrap(){}

int main()

{

printf("Enter the input:");

yylex();

}
```

OUTPUT:

```
Enter the input:temp

 is identifier

'

, is not a identifier
|
```

23. Implement a C program to eliminate left recursion

PROGRAM:

```c
#include<stdio.h>
#include<string.h>
int main()  {
    char input[100],l[50],r[50],temp[10],tempprod[20],productions[25][50];
    int i=0,j=0,flag=0,consumed=0;
    printf("Enter the productions: ");
    scanf("%1s->%s",l,r);
    printf("%s",r);
    while(sscanf(r+consumed,"%[^|]s",temp) == 1 && consumed <= strlen(r))  {
        if(temp[0] == l[0])  {
            flag = 1;
            sprintf(productions[i++],"%s->%s%s'\0",l,temp+1,l);
        }
        else
            sprintf(productions[i++],"%s'->%s%s'\0",l,temp,l);
        consumed += strlen(temp)+1;
    }
    if(flag == 1)  {
        sprintf(productions[i++],"%s->e\0",l);
        printf("The productions after eliminating Left Recursion are:\n");
        for(j=0;j<i;j++)
```
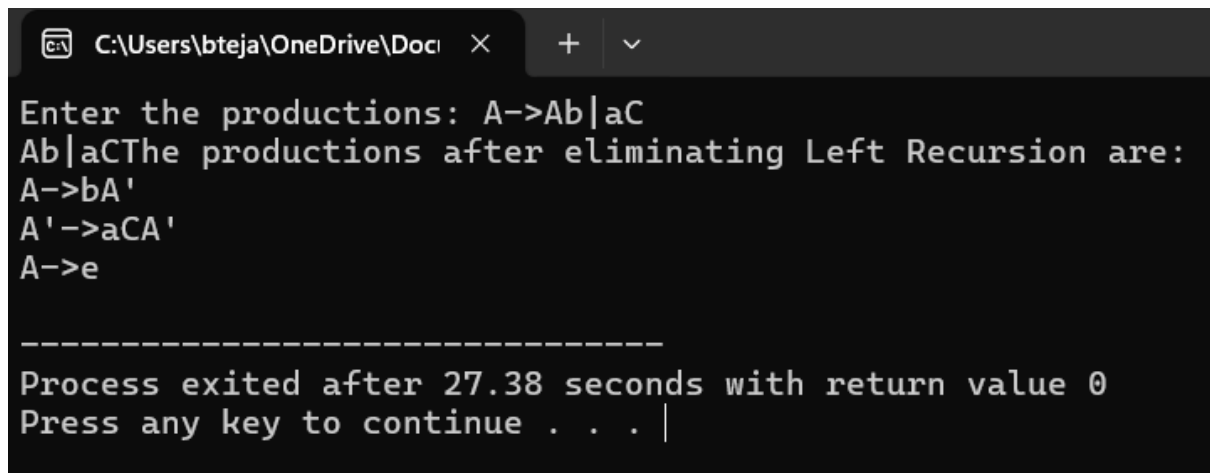
```
        printf("%s\n",productions[j]);

    }

   else

      printf("The Given Grammar has no Left Recursion");

}
```

OUTPUT:

```
🖾  C:\Users\bteja\OneDrive\Doc   ×    +    ∨

Enter the productions: A->Ab|aC
Ab|aCThe productions after eliminating Left Recursion are:
A->bA'
A'->aCA'
A->e

--------------------------------
Process exited after 27.38 seconds with return value 0
Press any key to continue . . . |
```

24. . Implement a C program to eliminate left factoring

PROGRAM:

#include<stdio.h>

#include<string.h>

int main()

{

  char gram[20],part1[20],part2[20],modifiedGram[20],newGram[20],tempGram[20];

  int i,j=0,k=0,l=0,pos;

  printf("Enter Production : A->");

  gets(gram);

  for(i=0;gram[i]!='|';i++,j++)

     part1[j]=gram[i];

   part1[j]='\0';

```c
    for(j=++i,i=0;gram[j]!='\0';j++,i++)
        part2[i]=gram[j];
    part2[i]='\0';
    for(i=0;i<strlen(part1)||i<strlen(part2);i++)
    {
        if(part1[i]==part2[i])
        {
            modifiedGram[k]=part1[i];
            k++;
            pos=i+1;
        }
    }
    for(i=pos,j=0;part1[i]!='\0';i++,j++){
        newGram[j]=part1[i];
    }
    newGram[j++]='|';
    for(i=pos;part2[i]!='\0';i++,j++){
        newGram[j]=part2[i];
    }
    modifiedGram[k]='X';
    modifiedGram[++k]='\0';
    newGram[j]='\0';
    printf("\n A->%s",modifiedGram);
    printf("\n X->%s\n",newGram);
}
```

OUTPUT:

```
Enter Production : A->Bc|Bd|ef

 A->BX
 X->c|d|ef
```

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