

## Practical 1

**Design a lexical analyzer for a given language and the lexical analyzer should ignore redundant spaces, tabs and newlines. 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. Simulate the same in C language.**

- **Code:**

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

int isKeyword(char buffer[])
{
    char keywords[32][10] = {"auto", "break", "case", "char", "const", "continue", "default", "do",
                             "double", "else", "enum", "extern", "float", "for", "goto", "if", "int", "long",
                             "register", "return",
                             "short", "signed", "sizeof", "static", "struct", "switch", "typedef", "union",
                             "unsigned", "void", "volatile", "while"};

    int i, flag = 0;
    for (i = 0; i < 32; ++i)
    {
        if (strcmp(keywords[i], buffer) == 0)
        {
            flag = 1;
            break;
        }
    }
    return flag;
}

int main()
{
    char ch, buffer[15], operators[] = "+-*/%=";
    FILE *fp;
    int i, j = 0;
```

```
fp = fopen("sample1.txt", "r");
if (fp == NULL)
{
    printf("Error while opening the file\n");
    exit(1); // You should exit with a non-zero value to indicate an error.
}

// Read characters from the file until EOF
while ((ch = fgetc(fp)) != EOF)
{
    for (i = 0; i < 6; ++i)
    {
        if (ch == operators[i])
            printf("%c is an operator\n", ch);
    }

    if (isalnum(ch) || ch == '_')
    {
        buffer[j++] = ch;
    }
    else if ((ch == ' ' || ch == '\n' || ch == '\t') && (j != 0))
    {
        buffer[j] = '\0';
        j = 0;
        if (isKeyword(buffer) == 1)
            printf("%s is a keyword\n", buffer);
        else
            printf("%s is an identifier\n", buffer);
    }
}

fclose(fp);
return 0;
}
```

- **Text File:**



```
1 int main ()
2 {
3     float a=10;
4     a++;
5     printf(" value is %.2f\n ",a);
6
7 }
```

- **Output:**

Output:

```
int is a keyword
main is an identifier
float is a keyword
= is an operator
a10 is an identifier
+ is an operator
+ is an operator
a is an identifier
printf is an identifier
value is an identifier
is is an identifier
% is an operator
2fn is an identifier
a is an identifier
```

## Practical 2

**Implement generic DFA to recognize any regular any regular expression and perform string validation.**

- **Code:**

```
#include <stdio.h>
#include <conio.h> if required

int ninputs;
int dfa[10][10];
char c[10], string[10];

int check(char b, int d);

int main() {
    int nstates, nfinals;
    int f[10];
    int i, j, s = 0, final = 0;

    printf("Enter the number of states that your DFA consists of: ");
    scanf("%d", &nstates);
    printf("Enter the number of input symbols that DFA has: ");
    scanf("%d", &ninputs);

    printf("\nEnter input symbols:\n");
    for (i = 0; i < ninputs; i++) {
        printf("Input %d: ", i + 1);
        scanf(" %c", &c[i]);
    }

    printf("\nEnter the number of final states: ");
    scanf("%d", &nfinals);

    for (i = 0; i < nfinals; i++) {
        printf("Final state %d: q", i + 1);
        scanf("%d", &f[i]);
    }

    printf("\nDefine transition rules as (initial state, input symbol) = final state:\n");
    for (i = 0; i < nstates; i++) {
        for (j = 0; j < ninputs; j++) {
            printf("(q%d, %c) = q", i, c[j]);
            scanf("%d", &dfa[i][j]);
        }
    }
}
```

```

}

do {
    i = 0;
    printf("\nEnter Input String: ");
    scanf("%s", string);

    while (string[i] != '\0') {
        s = check(string[i++], s);
        if (s < 0) break;
    }

    for (i = 0; i < nfinals; i++) {
        if (f[i] == s) {
            final = 1;
            break;
        }
    }

    if (final == 1)
        printf("Valid string\n");
    else
        printf("Invalid string\n");


    printf("Do you want to continue? (y/n): ");
} while (getchar() == 'y');

return 0;
}

int check(char b, int d) {
    int j;
    for (j = 0; j < ninputs; j++) {
        if (b == c[j]) {
            return dfa[d][j];
        }
    }
    return -1;
}

```

- **Output:**

 Terminal

```
Enter the number of states that your DFA consists of: 3
Enter the number of input symbols that DFA has: 2
Enter input symbols:
Input 1: 0
Input 2: 1
Enter the number of final states: 1
Final state 1: q2
Define transition rules as (initial state, input symbol) = final state:
(q0, 0) = q0
(q0, 1) = q2
(q1, 0) = q2
(q1, 1) = q1
(q2, 0) = q1
(q2, 1) = q2
Enter Input String: 1001
Valid string
Do you want to continue? (y/n): y
Enter Input String: 100101
Valid string
```

### **Practical 3**

**Implement generic DFA to recognize any regular any regular expression and perform string validation.**

- **What is LEX tool?**

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

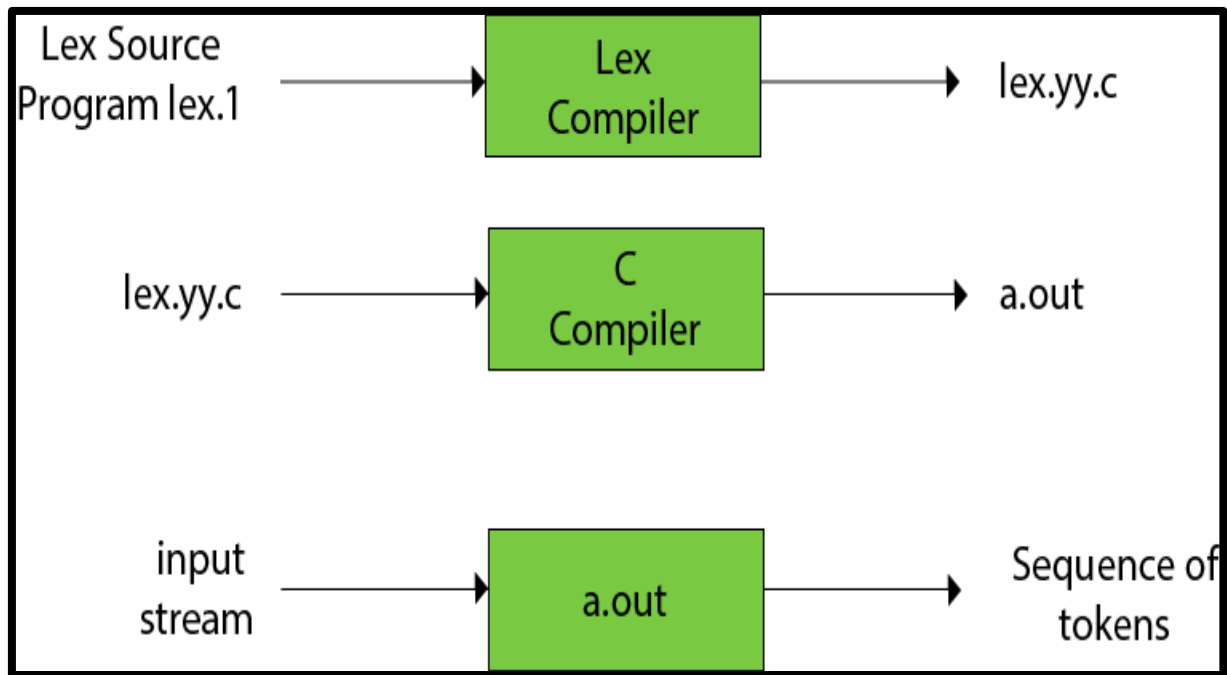
- **The functions of LEX are follows:**

- Firstly, lexical analyzer creates a program lex.l in the Lex language. Then Lex compiler runs the lex.l program and produces a C program lex.yy.c.
- Finally, C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.

- **Importance of LEX tool:**

- Lex is a lexical analysis tool that can be used to identify specific text strings in a structured way from source text. Yacc is a grammar parser; it reads text and can be used to turn a sequence of words into a structured format for processing.

- **Use of LEX tool:**



- **lex.l** is an a input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms lex.l to a C program known as lex.yy.c.
- **lex.yy.c** is compiled by the C compiler to a file called a.out.
- The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.
- **yylval** is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.
- The attribute value can be numeric code, pointer to symbol table or nothing.
- Another tool for lexical analyzer generation is Flex.

- **Structure of LEX program:**

Lex program will be in following form

**declarations**

%%

**Translation rules**

%%

**auxiliary functions**



- **Declarations:** This section includes declaration of variables, constants and regular definitions.

- **Translation rules:** It contains regular expressions and code segments.

Form: Pattern { Action }

Pattern is a regular expression or regular definition.

Action refers to segments of code.

- **Auxiliary functions:** This section holds additional functions which are used in actions. These functions are compiled separately and loaded with lexical analyzer. Lexical analyzer produced by lex starts its process by reading one character at a time until a valid match for a pattern is found. Once a match is found, the associated action takes place to produce token. The token is then given to parser for further processing.

- **Conflict resolution in lex tool:**

Conflict arises when several prefixes of input matches one or more patterns. This can be resolved by the following:

- Always prefer a longer prefix than a shorter prefix.
- If two or more patterns are matched for the longest prefix, then the first pattern listed in lex program is preferred.

- **Lookahead Operators:**

- Lookahead operator is the additional operator that is read by lex in order to distinguish additional pattern for a token.
- Lexical analyzer is used to read one character ahead of valid lexeme and then retracts to produce token.
- At times, it is needed to have certain characters at the end of input to match with a pattern.

Regular Expression	Meaning
*	Matches with zero or more occurrences of preceding expressions. For example, 1* occurrence of 1 for any number of times.
.	Matches any single character other than new line character.
[ ]	A character class which matches any character within the bracket. For example, [a-z] matches with any alphabet in lower case.
( )	Group of regular expressions together put into a new regular expression.
“ ”	The string written in quotes matches literally. For example, “Hanumaan” matches with the string Hanumaan.
\$	Matches with the end of line as last character.
+	Matches with one or more occurrences of preceding expression. For example, [0 – 9] + any number but not empty string.
?	Matches zero or one occurrence of preceding regular expression. For example, [+ -] ? [0 – 9] + a number with unary operator.
^	Matches the beginning of a line as first character.
[^]	Used as for negation. For example, [^verb] means except verb match with anything else.
\	Used as escape metacharacter. For example, \n is a new line character. \# prints the # literally
	To represent the or i.e. another alternative. For example, a   b means match with either a or b.

### • LEX Actions:

There are various LEX actions that can be used for ease of programming using LEX tool.

**1. BEGIN** – It indicates the start state. The lexical analyzer starts at state 0.

**2. ECHO** – It emits the input as it is.

**3. yytext** – When lexer matches or recognize the token from input token then the lexeme is stored in a null terminated string called yytext.

As soon as new token is found the contents of yytext are replaced by new token.

**4. yylex()** – This is an important function. As soon as call to yytext() is encountered scanner starts scanning the source program.

**5. yywrap()** – The function yywrap() is called when scanner encounters end of file. If yywrap() returns 0 then scanner continues scanning. When yywrap() returns 1 that means end of file is encountered.

**6. yyin** – It is the standard input file that stores input source program.

**7. yyleng** – When a lexer recognize token then the lexeme is stored in a null terminated string called yytext. And yyleng stores the length or number of characters in input string. The value in yyleng is same as strlen() functions.

## 8. How to write main() in LEX?

The main() can be written as

```
void main  
{  
    yylex();  
}
```

Or it can be written with command line parameters such as argc and argv. If command line parameters are passed to the main then while executing the program we give input source file name.

For example – if the source program which is to be scanned is x.c then

```
#!/a.out x.c
```

Will scan the input source **x.c**.

## 9. Where to write C code –

- We can write a valid 'C' code between  
**%{%}**
- We can write any C function in a subroutine section.
- We can write valid C code in the action part for corresponding regular expression.

We have to construct recognizer that looks for the lexemes stored in the input buffer.

It works using these two rules -

1. If more than one pattern matches then recognizer has to choose the longest lexeme matched.
2. If there are two or more pattern that match the longest lexeme, the first listed matching pattern is chosen.

### • Using Command Line Parameters

- The command line parameters that are appearing on the shell prompt.
- The command line interface is the interface which allows the user to interact with the computer by typing the commands.
- In C we can pass these parameters to the main function to the form of character array.

For example:

**\$ cp abc.txt pqr.txt**

Here

**argv[0] = cp**

**argv[1] = abc.txt**

**argv[2] = pqr.txt**

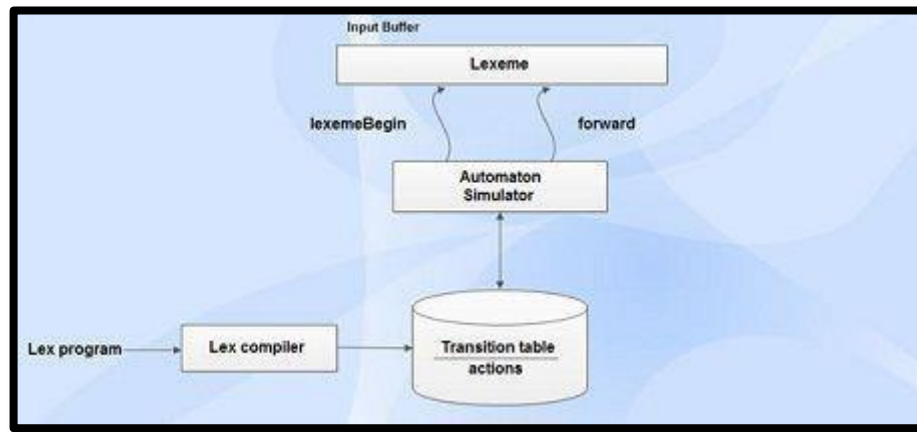
As there are three such parameters that are present at the command line interface **argc** value will be 3.

### • Design of lexical analyzer:

- Lexical analyzer can either be generated by NFA or by DFA.
- DFA is preferable in the implementation of lex.

- **Structure of generated analyzer:**

- Architecture of lexical analyzer generated by lex is given in fig.



- **Lexical analyzer program includes:**

- Program to simulate automata.
- Components created from lex program by lex itself which are listed as follows:
  - A transition table for automaton.
  - Functions that are passed directly through lex to the output.
  - Actions from input program (fragments of code) which are invoked by automation simulator when needed.

## LEX Program Tool:

### 1. Lex Program to accept string starting with vowel

- **CODE:**

```

% {
int flag = 0;
% }

%%

```

```
[aeiouAEIOU].[a-zA-Z0-9.]+ flag=1;[a-zA-Z0-9.]+
```

```
%%
```

```
main()
{
    yylex();
    if (flag == 1) printf("Accepted");
    else
        printf("Not Accepted");
}
```

## 2. Lex program to count number of words.

- **CODE:**

```
%{
#include<stdio.h>#include<string.h> inti = 0;
%}
```

```
/* Rules Section*/
```

```
%%
([a-zA-Z0-9])* {i++;} /* Rule for counting
number of words*/
```

```
"\n" {printf("%d\n", i); i = 0;}
%%
```

```
int yywrap(void){}int main()
{
//Thefunctionthatstartstheanalysisyylex();

return 0;
}
```

## 3. Lex program to count noof lines, words ,digits,small letters, capital letters and special characters.

- **CODE:**

```
%{
#include<stdio.h>
int lines=0, words=0,s_letters=0,c_letters=0, num=0, spl_char=0,total=0;
%}
```

```

%%
\n { lines++; words++; } [\t ' ] words++;
[A-Z]
c_letters++; [a-z] s_letters++; [0-9] num++;
. spl_char++;
%%
int main()
{
yyin= fopen("test.txt","r");yylex();
total=s_letters+c_letters+num+spl_char;

printf(" This File contains ..."); printf("\n\t%d lines", lines); printf("\n\t%d words", words);
printf("\n\t%d small letters", s_letters); printf("\n\t%d capital letters", c_letters); printf("\n\t%d
digits", num);
printf("\n\t%d special characters", spl_char); printf("\n\tIn total %d characters..\n", total);
}
int yywrap(void){ }

```

#### 4. Lex program to convert lowercase to uppercase.

- **CODE:**

```

lower [a-z] CAPS [A-Z]
space    [ \t\n]

%%
{lower}    {printf("%c",yytext[0]- 32);}
{CAPS}     {printf("%c",yytext[0]+ 32);}
{space}    ECHO;
.          ECHO;
%%

main()
{
yylex();

}
int yywrap(void){ }

```

### Practical 4

**Implement the following programs using Lex.**

- A. Create a program to take input from a text file and count no of characters, no. of lines & no. of words.**
- B. Count the number of vowels and consonants in a given input string.**
- C. Print only numbers from the given file.**
- D. Convert Roman to Decimal**
- E. Print all HTML tags from the given file.**
- F. Add line numbers to the given file and display them on the standard output.**
- G. Count the number of comment lines in a given C program.**
- H. Eliminate comments from a given C program and create a separate file without comments.**
- I. Generate Histogram of Words**
- J. Caesar Cypher**
- K. Check weather given statement is compound or simple**

**A. Create a program to take input from a text file and count no of characters, no. of lines & no. of words.**

- **Code:**

```
%  
{  
    #include<studio.h> int n_chars=0;  
    int n_lines=0; int n_words = 0;  
}%  
  
%%  
  
\n {n_chars++;n_lines++;}  
[^\n\t]+ {n_words++, n_chars=n_chars+yylength;}
```



```
. {n_chars++;}
```

```
%%
```

```
int yywrap(){}
```

```
int main(int argc[],char *argv[])
```

```
{
```

```
yyin=fopen("c:\\practical.txt", "r");
```

```
yylex();
```

```
printf("no of characters: %d",n_chars); printf("\n");
```

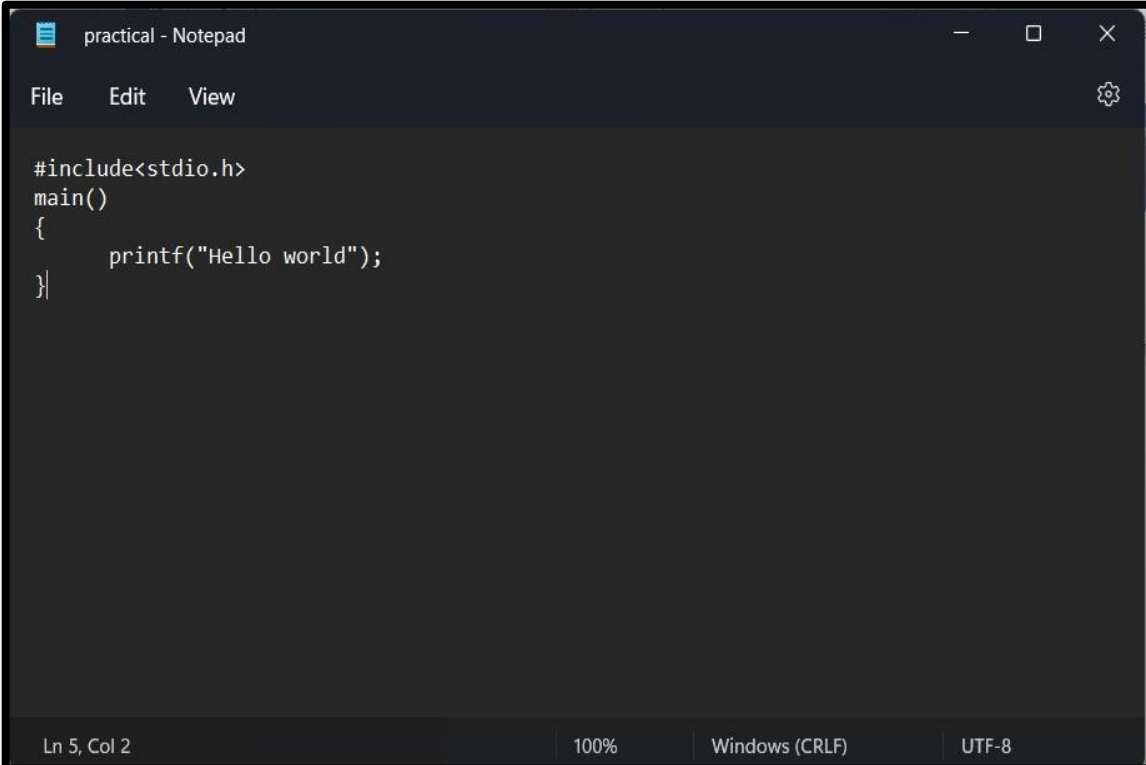
```
printf("no of lines: %d",n_lines); printf("\n");
```

```
printf("no of words: %d",n_words); printf("\n");
```

```
return 0;
```

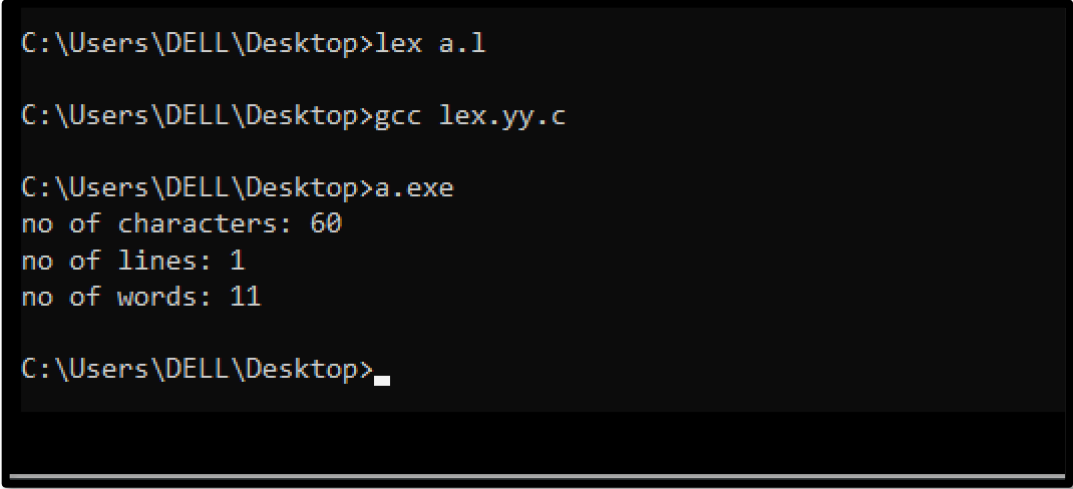
```
}
```

- **Text File:**



```
#include<stdio.h>
main()
{
    printf("Hello world");
}
```

- **Output:**



```
C:\Users\DELL\Desktop>lex a.1

C:\Users\DELL\Desktop>gcc lex.yy.c

C:\Users\DELL\Desktop>a.exe
no of characters: 60
no of lines: 1
no of words: 11

C:\Users\DELL\Desktop>_
```

## **B. Count the number of vowels and consonants in a given input string.**

- **Code:**

```
%
{
    int vow_count=0; int const_count =0;
}%

%%
[aeiouAEIOU] {vow_count++;} [a-zA-Z] {const_count++;}
%%
int yywrap(){} int main()
{
    printf("Enter the string of vowels and consonants:"); yylex();
    printf("Number of vowels are: %d\n", vow_count); printf("Number of consonants are: %d\n",
    const_count); return 0;
}
```

- **Output:**

```
C:\Flex Windows\EditPlusPortable>a.exe
Enter the string of vowels and consonants:SHIVAM SIDDHAPURA

Number of vowels are: 6
Number of consonants are: 10
```

### C. Print only numbers from the given file.

- **Code:**

```
%
{
#include<stdio.h> int num_count=0;
%}

num [0-9]+

%%
{num} {num_count++; printf("%s",yytext);
}
%%
int yywrap(void){ } int main()
{
yyin = fopen("c:\\practical.txt", "r");
yylex();
printf("\nTotal of numbers: %d",num_count);
}
```

- **Text File:**

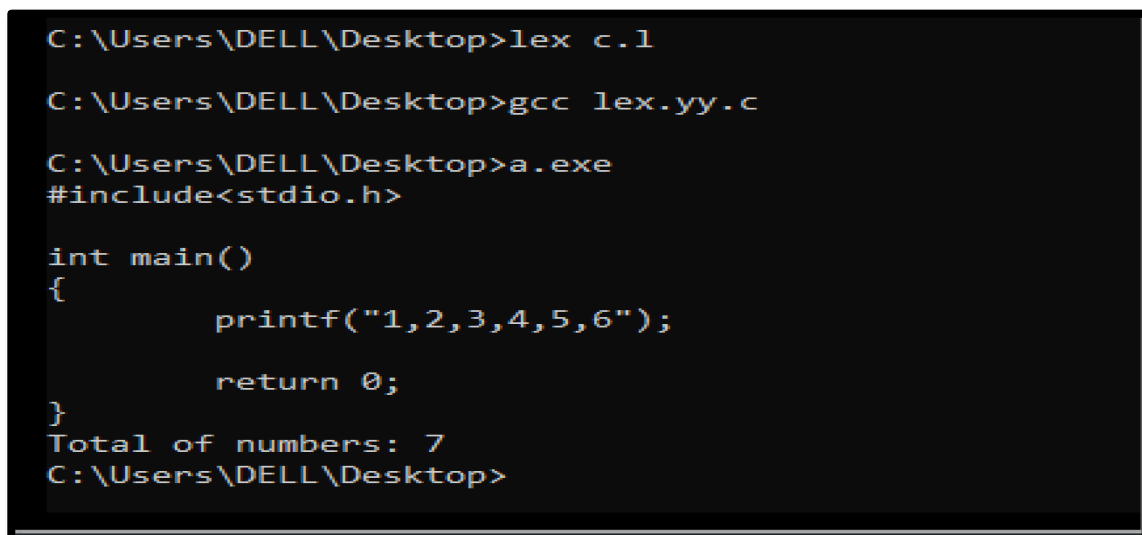


```
*practical - Notepad
File Edit View
#include<stdio.h>
#include<conio.h>

void main()
{
    printf("1,2,3,4,5");
}

Ln 6, Col 20 100% Windows (CRLF) UTF-8
```

- **Output:**



```
C:\Users\DELL\Desktop>lex c.1
C:\Users\DELL\Desktop>gcc lex.yy.c
C:\Users\DELL\Desktop>a.exe
#include<stdio.h>

int main()
{
    printf("1,2,3,4,5,6");

    return 0;
}
Total of numbers: 7
C:\Users\DELL\Desktop>
```

**E. Print all HTML tags from the given file.**

- **Code:**

```
%{  
%}  
  
%%  
"<[^>]*> {printf("%s\n", yytext); } /* if anything enclosed in  
these < > occur print text*/  
.  
; // else do nothing  
%%  
  
int yywrap(){  
  
int main(int argc, char*argv[])  
{  
    // Open tags.txt in read mode  
    extern FILE *yyin = fopen("tags.txt","r");  
  
    // The function that starts the analysis  
    yylex();  
  
    return 0;  
}
```

- **Output:**

```
<html>
<head>
</head>
<body>
<h1>
</h1>
<p>
</p>
</body>
</html>
```

**F. Add line numbers to the given file and display them on the standard output.**

- **Code:**

```
%
{
    int line_number = 1;

    line .*\n

    %%
    {line} { printf("%10d %s", line_number++, yytext); }

    %%

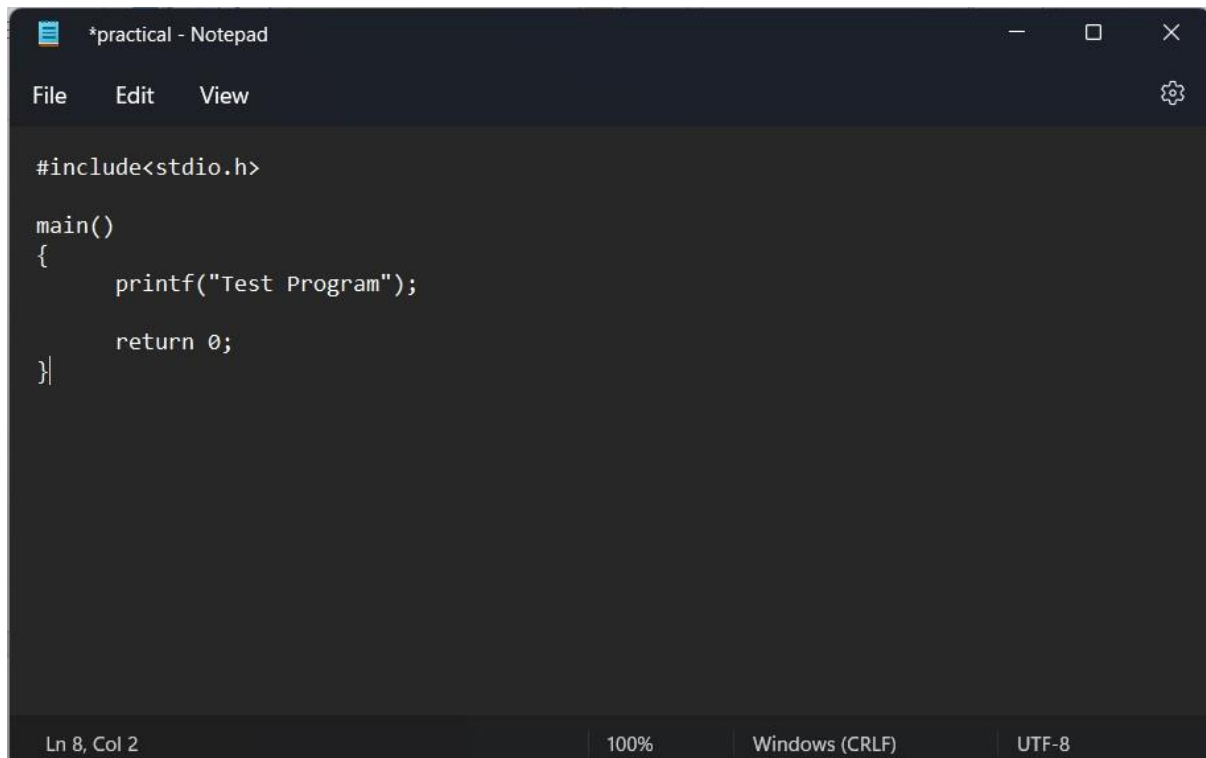
    int yywrap(){ }

    int main(int argc, char*argv[])
    {
        extern FILE *yyin;

        yyin = fopen("c:\\practical.txt", "r");
        yylex();
```

```
}
```

- **Text File:**



A screenshot of a Notepad window titled '\*practical - Notepad'. The window has a menu bar with 'File', 'Edit', and 'View'. The text area contains the following C code:

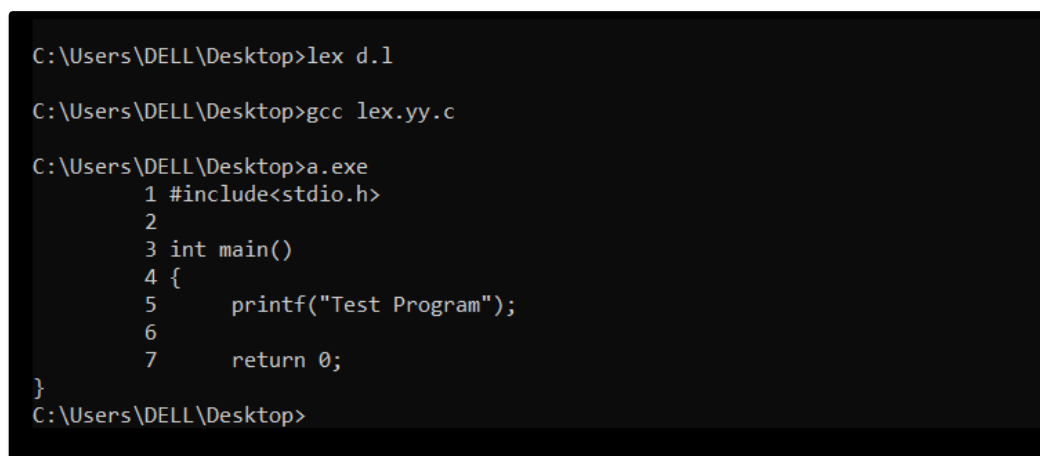
```
#include<stdio.h>

main()
{
    printf("Test Program");

    return 0;
}
```

The status bar at the bottom shows 'Ln 8, Col 2', '100%', 'Windows (CRLF)', and 'UTF-8'.

- **Output:**



A screenshot of a Windows command prompt window showing the following commands and their output:

```
C:\Users\DELL\Desktop>lex d.1
C:\Users\DELL\Desktop>gcc lex.yy.c
C:\Users\DELL\Desktop>a.exe
1 #include<stdio.h>
2
3 int main()
4 {
5     printf("Test Program");
6
7     return 0;
}
C:\Users\DELL\Desktop>
```

**G. Count the number of comment lines in a given C program.**

- Code:**

```
% {#include<stdio.h>
int count = 0;
% }

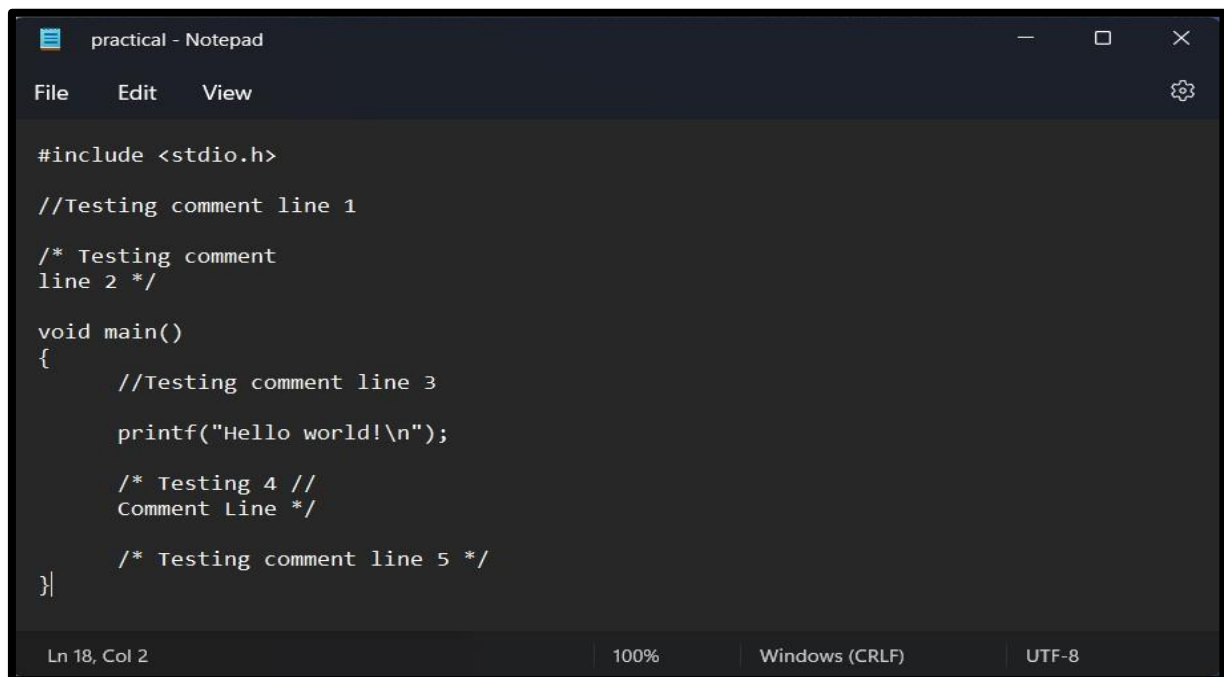
% x C
%%
"/[*"][.]*"/ {count++;}
"/[*"      {BEGIN C;}
<C>"/      {BEGIN 0; count++;}
<C>\n      {;}
<C>.       {;}
\\.*       {count++;}
%%

void main()
{
    char file[] = "data.c";
    yyin = fopen("c:\\practical.txt", "r");
    yylex();
    printf("Number of comment lines in c file %s is %d\n", file, count);
}

int yywrap()
{
    return 1;
}
```



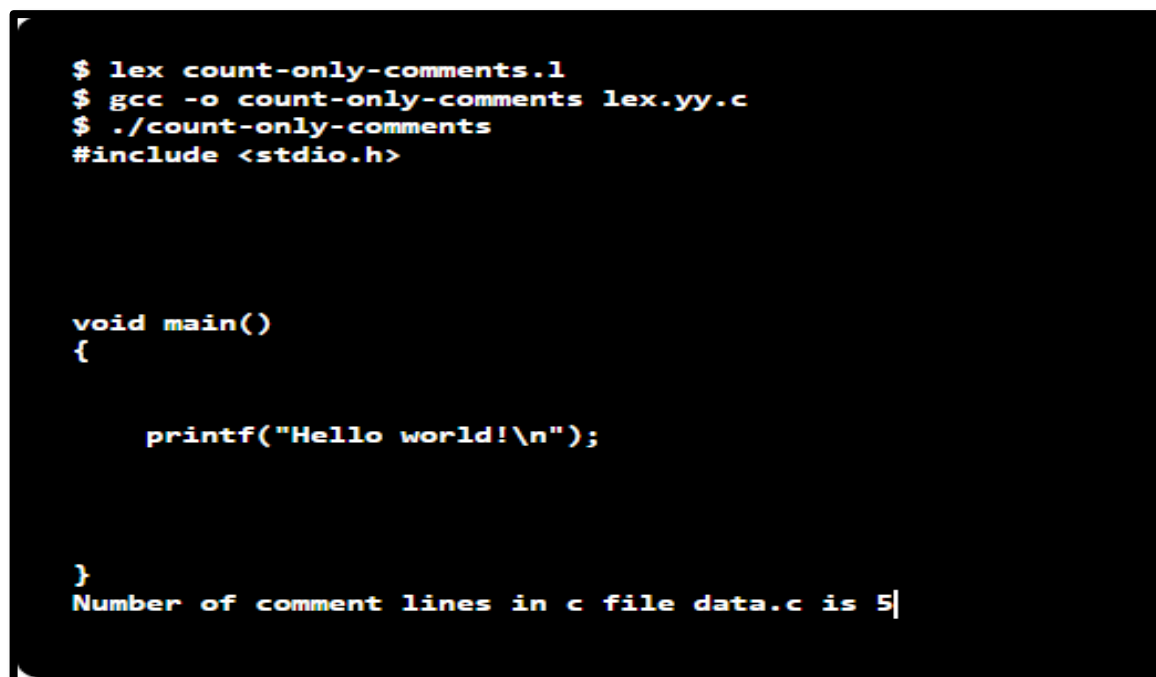
- **Text File:**



```
practical - Notepad
File Edit View
#include <stdio.h>
//Testing comment line 1
/* Testing comment
line 2 */
void main()
{
    //Testing comment line 3
    printf("Hello world!\n");
    /* Testing 4 //
    Comment Line */
    /* Testing comment line 5 */
}
```

Ln 18, Col 2 | 100% | Windows (CRLF) | UTF-8

- **Output:**



```
$ lex count-only-comments.l
$ gcc -o count-only-comments lex.yy.c
$ ./count-only-comments
#include <stdio.h>

void main()
{

    printf("Hello world!\n");

}
Number of comment lines in c file data.c is 5|
```

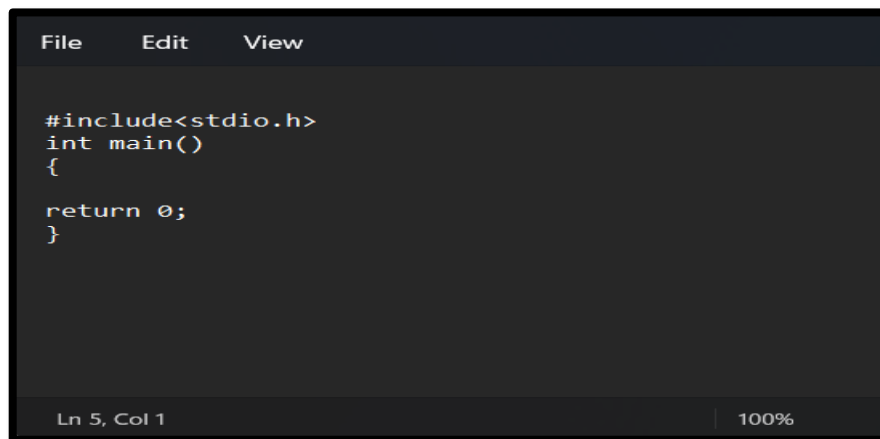
**H. Eliminate comments from a given C program and create a separate file without comments.**

- **Code:**

```
%  
{  
    #include<stdio.h>  
}%  
  
%%  
  
\\(.*) {};  
  
\\*(.*\\n)*.*\\*\\ {};  
%%  
  
int yywrap()  
{  
    return 1;  
}  
  
int main()  
{  
    yyin=fopen("c:\\practical.txt", "r");  
    yyout=fopen("C:\\out.c", "w");  
  
    yylex();  
    return 0;  
}
```

- **Output:**

```
//testing
#include<stdio.h>
int main()
{
/*multipleline comment
continue....
*/
return 0;
}
```



```
File Edit View

#include<stdio.h>
int main()
{

return 0;
}

Ln 5, Col 1 100%
```

## I. Generate Histogram of Words

- **Code:**

```
%
{
#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("input.txt", "r");
    yylex();

    printf("%d", count);
}
```

- **Output:**

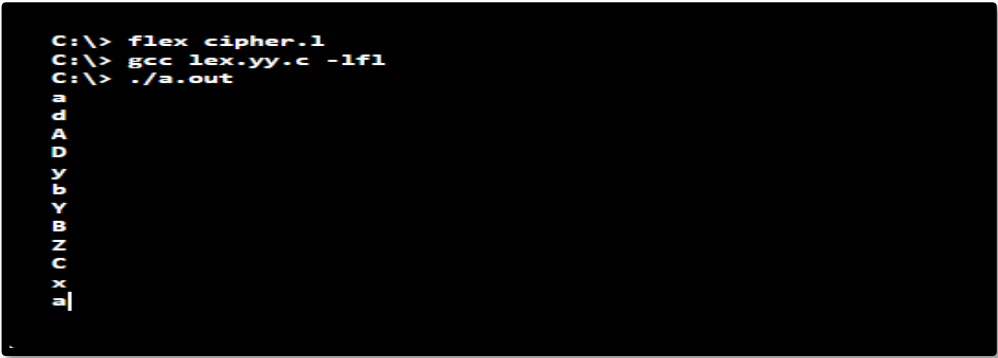
**3**

## J. Caesar Cypher

- **Code:**

```
%%  
[a-z] { char ch = yytext[0];  
      ch += 3;  
      if (ch> 'z') ch -= ('z'+1- 'a');  
      printf ("%c" ,ch );  
}  
[A-Z] { char ch = yytext[0] ;  
      ch += 3;  
      if (ch> 'Z') ch -= ('Z'+1- 'A');  
      printf ("%c",ch);  
}  
%%
```

- **Output:**



```
C:\> flex cipher.1  
C:\> gcc lex.yy.c -lf1  
C:\> ./a.out  
a  
d  
A  
D  
y  
b  
Y  
B  
Z  
C  
x  
|
```

**K. Check weather given statement is compound or simple**

- **Code:**

```
%{  
    #include<stdio.h>  
    int flag=0;  
}%  
  
%%  
and |  
or |  
but |  
because |  
if |  
then |  
nevertheless { flag=1; }  
· ;  
\n { return 0; }  
%%  
  
int main()  
{  
    printf("Enter the sentence:\n");  
    yylex();  
    if(flag==0)  
        printf("Simple sentence\n");  
    else  
        printf("compound sentence\n");  
}  
  
int yywrap( )  
{
```

```
    return 1;  
}
```

- **Output:**

```
C:\> flex compound.1  
C:\> gcc lex.yy.c  
C:\> ./a.out  
  
Enter the sentence:  
  
My name is Gaurang Revdiwala and I am from India  
  
Compound sentence  
  
C:\> ./a.out  
  
My name is Gaurang Revdiwala  
  
Simple sentence|
```

## Practical 5

**Write a C program to find FIRST and FOLLOW of specific grammar.**

**Input:** The string consists of grammar symbols.

**Output:** The First and Follow set for a given string.  
**Explanation:** The student has to assume a typical grammar. The program when run will ask for the string to be entered. The program will find the First and Follow set of the given string.

- **Code:**

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

char a[8][8];
int n=5 ; struct firTab
{
int n;
char firT[5];
};
struct folTab
{
int n;
char folT[5];
};
struct folTab follow[5]; struct firTab first[5]; int col;
void findFirst(char,char); void findFollow(char,char);
void folTabOperation(char,char); void firTabOperation(char,char); void main()
{
int i,j,c=0,cnt=0; char ip;
char b[8];
printf("\n\t\t=====FIRST AND FOLLOW SET===== \n\nEnter Production Amount : ");
scanf("%d",&n);
printf("\n\tenter %d productions in format E->T+G\n",n); printf("\t\n");
for(i=0; i<n; i++)
{
scanf("%s",&a[i]);
}
for(i=0; i<n; i++)
{ c=0;
for(j=0; j<i+1; j++)
{
if(a[i][0] == b[j])
```



```
{
c=1;
break;
}
}
if(c !=1)
{
b[cnt] = a[i][0]; cnt++;
}

}
printf("\n");

for(i=0; i<cnt; i++)
{
col=1; first[i].firT[0] = b[i]; first[i].n=0; findFirst(b[i],i);
}
for(i=0; i<cnt; i++)
{
col=1; follow[i].folT[0] = b[i]; follow[i].n=0; findFollow(b[i],i);
}

printf("\n"); for(i=0; i<cnt; i++)
{
for(j=0; j<=first[i].n; j++)
{
if(j==0)
{
printf("First(%c) : {",first[i].firT[j]);
}
else
{
printf(" %c",first[i].firT[j]);
}
}
printf(" } ");
printf("\n");
}
printf("\n"); for(i=0; i<cnt; i++)
{

for(j=0; j<=follow[i].n; j++)
{
if(j==0)
{
printf("Follow(%c) : {",follow[i].folT[j]);
}
else
{

```

```
printf(" %c",follow[i].folT[j]);
}
}
printf(" } ");

printf("\n");
}

}
void findFirst(char ip,char pos)
{
int i;
for(i=0; i<n; i++)
{
if(ip == a[i][0])
{
if(isupper(a[i][3]))
{
findFirst(a[i][3],pos);
}
}
else
{

first[pos].firT[col]=a[i][3]; first[pos].n++;
col++;
}
}
}
}
void findFollow(char ip,char row)
{
int i,j;
if(row==0 && col==1)
{
follow[row].folT[col]= '$'; col++;
follow[row].n++;
}
for(i=0; i<n; i++)

{
for(j=3; j<7; j++)
{
if(a[i][j] == ip)
{
if(a[i][j+1] == '\0')
{
if(a[i][j] != a[i][0])
{
folTabOperation(a[i][0],row);
}
}
}
}
}
```

```
}
else if(isupper(a[i][j+1]))
{
if(a[i][j+1] != a[i][0])
{
firTabOperation(a[i][j+1],row);

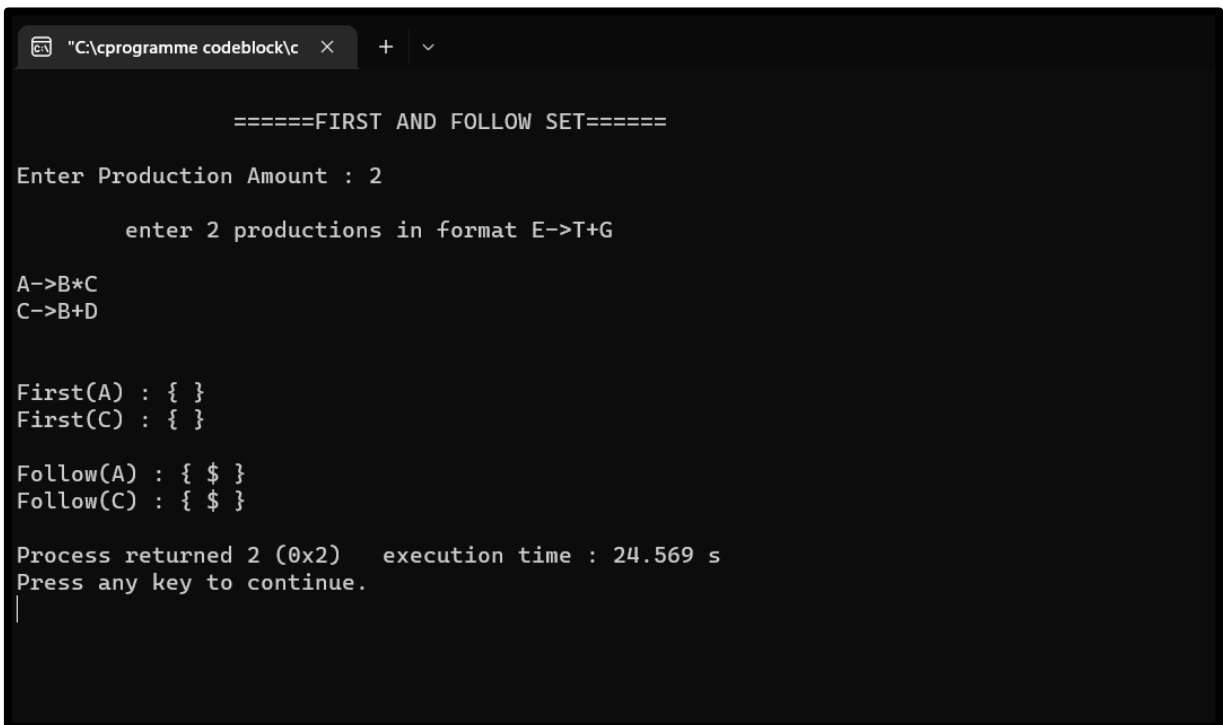
}
}
else
{
follow[row].folT[col] = a[i][j+1]; col++;
follow[row].n++;
}
}
}
}
}
void folTabOperation(char ip,char row)
{
int i,j;
for(i=0; i<5; i++)
{
if(ip == follow[i].folT[0])
{
for(j=1; j<=follow[i].n; j++)
{
follow[row].folT[col] = follow[i].folT[j]; col++;
follow[row].n++;
}
}
}
}

void firTabOperation(char ip,char row)
{
int i,j;
for(i=0; i<5; i++)
{
if(ip == first[i].firT[0])
{
for(j=1; j<=first[i].n; j++)
{
if(first[i].firT[j] != '0')
{
follow[row].folT[col] = first[i].firT[j]; follow[row].n++;
col++;
}
}
else
{

```

```
folTabOperation(ip,row);  
}  
}  
}  
}  
}
```

- **Output:**



```
"C:\programme\codeblock\c" X + v  
  
=====FIRST AND FOLLOW SET=====  
Enter Production Amount : 2  
enter 2 productions in format E->T+G  
A->B*C  
C->B+D  
  
First(A) : { }  
First(C) : { }  
  
Follow(A) : { $ }  
Follow(C) : { $ }  
  
Process returned 2 (0x2) execution time : 24.569 s  
Press any key to continue.  
|
```

## Practical 6

**Write a C program for constructing LL (1) parsing using FIRST and FOLLOW generated in the above program.**

- **Code:**

```
#include<stdio.h>
#include<string.h>
#define TSIZE 128
// table[i][j] stores
// the index of production that must be applied on
// ith variable if the input is
// jth nonterminal
int table[100][TSIZE];
// stores all list of terminals
// the ASCII value if use to index terminals
// terminal[i] = 1 means the character with
// ASCII value is a terminal
char terminal[TSIZE];
// stores all list of terminals
// only Upper case letters from 'A' to 'Z'
// can be nonterminals
// nonterminal[i] means ith alphabet is present as
// nonterminal is the grammar
char nonterminal[26];
// structure to hold each production
// str[] stores the production
// len is the length of production
struct product {
    char str[100];
    int len;
}pro[20];
// no of productions in form A->β
int no_pro;
char first[26][TSIZE];
char follow[26][TSIZE];
// stores first of each production in form A->β
char first_rhs[100][TSIZE];
// check if the symbol is nonterminal
```

```
int isNT(char c) {
    return c >= 'A' && c <= 'Z';
}
// reading data from the file
void readFromFile() {
    FILE* fptr;
    fptr = fopen("text.txt", "r");
    char buffer[255];
    int i;
    int j;
    while (fgets(buffer, sizeof(buffer), fptr)) {
        printf("%s", buffer);
        j = 0;
        nonterminal[buffer[0] - 'A'] = 1;
        for (i = 0; i < strlen(buffer) - 1; ++i) {
            if (buffer[i] == '|') {
                ++no_pro;
                pro[no_pro - 1].str[j] = '\0';
                pro[no_pro - 1].len = j;
                pro[no_pro].str[0] = pro[no_pro - 1].str[0];
                pro[no_pro].str[1] = pro[no_pro - 1].str[1];
                pro[no_pro].str[2] = pro[no_pro - 1].str[2];
                j = 3;
            }
            else {
                pro[no_pro].str[j] = buffer[i];
                ++j;
                if (!isNT(buffer[i]) && buffer[i] != '-' && buffer[i] != '>') {
                    terminal[buffer[i]] = 1;
                }
            }
        }
        pro[no_pro].len = j;
        ++no_pro;
    }
}

void add_FIRST_A_to_FOLLOW_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            follow[B - 'A'][i] = follow[B - 'A'][i] || first[A - 'A'][i];
    }
}
```

```

}
void add_FOLLOW_A_to_FOLLOW_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            follow[B - 'A'][i] = follow[B - 'A'][i] || follow[A - 'A'][i];
    }
}
void FOLLOW() {
    int t = 0;
    int i, j, k, x;
    while (t++ < no_pro) {
        for (k = 0; k < 26; ++k) {
            if (!nonterminal[k]) continue;
            char nt = k + 'A';
            for (i = 0; i < no_pro; ++i) {
                for (j = 3; j < pro[i].len; ++j) {
                    if (nt == pro[i].str[j]) {
                        for (x = j + 1; x < pro[i].len; ++x) {
                            char sc = pro[i].str[x];
                            if (isNT(sc)) {
                                add_FIRST_A_to_FOLLOW_B(sc, nt);
                                if (first[sc - 'A']['^'])
                                    continue;
                            }
                        }
                        else {
                            follow[nt - 'A'][sc] = 1;
                        }
                    }
                    break;
                }
                if (x == pro[i].len)
                    add_FOLLOW_A_to_FOLLOW_B(pro[i].str[0], nt);
            }
        }
    }
}
void add_FIRST_A_to_FIRST_B(char A, char B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^') {

```

```

        first[B - 'A'][i] = first[A - 'A'][i] || first[B - 'A'][i];
    }
}
}
void FIRST() {
    int i, j;
    int t = 0;
    while (t < no_pro) {
        for (i = 0; i < no_pro; ++i) {
            for (j = 3; j < pro[i].len; ++j) {
                char sc = pro[i].str[j];
                if (isNT(sc)) {
                    add_FIRST_A_to_FIRST_B(sc, pro[i].str[0]);
                    if (first[sc - 'A']['^'])
                        continue;
                }
                else {
                    first[pro[i].str[0] - 'A'][sc] = 1;
                }
                break;
            }
            if (j == pro[i].len)
                first[pro[i].str[0] - 'A']['^'] = 1;
        }
        ++t;
    }
}
void add_FIRST_A_to_FIRST_RHS_B(char A, int B) {
    int i;
    for (i = 0; i < TSIZE; ++i) {
        if (i != '^')
            first_rhs[B][i] = first[A - 'A'][i] || first_rhs[B][i];
    }
}
// Calculates FIRST( $\beta$ ) for each  $A \rightarrow \beta$ 
void FIRST_RHS() {
    int i, j;
    int t = 0;
    while (t < no_pro) {
        for (i = 0; i < no_pro; ++i) {
            for (j = 3; j < pro[i].len; ++j) {
                char sc = pro[i].str[j];

```



```
        if (isNT(sc)) {
            add_FIRST_A_to_FIRST_RHS_B(sc, i);
            if (first[sc - 'A']['^'])
                continue;
        }
        else {
            first_rhs[i][sc] = 1;
        }
        break;
    }
    if (j == pro[i].len)
        first_rhs[i]['^'] = 1;
    }
    ++t;
}
}

int main() {
    readFromFile();
    follow[pro[0].str[0] - 'A']['$'] = 1;
    FIRST();
    FOLLOW();
    FIRST_RHS();
    int i, j, k;

    // display first of each variable
    printf("\n");
    for (i = 0; i < no_pro; ++i) {
        if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
            char c = pro[i].str[0];
            printf("FIRST OF %c: ", c);
            for (j = 0; j < TSIZE; ++j) {
                if (first[c - 'A'][j]) {
                    printf("%c ", j);
                }
            }
            printf("\n");
        }
    }
}

// display follow of each variable
printf("\n");
for (i = 0; i < no_pro; ++i) {
```

```

    if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
        char c = pro[i].str[0];
        printf("FOLLOW OF %c: ", c);
        for (j = 0; j < TSIZE; ++j) {
            if (follow[c - 'A'][j]) {
                printf("%c ", j);
            }
        }
        printf("\n");
    }
}

// display first of each variable β
// in form A->β
printf("\n");
for (i = 0; i < no_pro; ++i) {
    printf("FIRST OF %s: ", pro[i].str);
    for (j = 0; j < TSIZE; ++j) {
        if (first_rhs[i][j]) {
            printf("%c ", j);
        }
    }
    printf("\n");
}

// the parse table contains '$'
// set terminal['$'] = 1
// to include '$' in the parse table
terminal['$'] = 1;

// the parse table do not read '^'
// as input
// so we set terminal['^'] = 0
// to remove '^' from terminals
terminal['^'] = 0;

// printing parse table
printf("\n");
printf("\n\t***** LL(1) PARSING TABLE *****\n");
printf("\t-----\n");
printf("%-10s", "");
for (i = 0; i < TSIZE; ++i) {
    if (terminal[i]) printf("%-10c", i);
}

```

```

    }
    printf("\n");
    int p = 0;
    for (i = 0; i < no_pro; ++i) {
        if (i != 0 && (pro[i].str[0] != pro[i - 1].str[0]))
            p = p + 1;
        for (j = 0; j < TSIZE; ++j) {
            if (first_rhs[i][j] && j != '^') {
                table[p][j] = i + 1;
            }
            else if (first_rhs[i]['^']) {
                for (k = 0; k < TSIZE; ++k) {
                    if (follow[pro[i].str[0] - 'A'][k]) {
                        table[p][k] = i + 1;
                    }
                }
            }
        }
    }
    k = 0;
    for (i = 0; i < no_pro; ++i) {
        if (i == 0 || (pro[i - 1].str[0] != pro[i].str[0])) {
            printf("%-10c", pro[i].str[0]);
            for (j = 0; j < TSIZE; ++j) {
                if (table[k][j]) {
                    printf("%-10s", pro[table[k][j] - 1].str);
                }
                else if (terminal[j]) {
                    printf("%-10s", "");
                }
            }
            ++k;
            printf("\n");
        }
    }
}

```

- **Text File:**

```

practical - Notepad
File Edit View
E->TA
A->+TA|^
T->FB
B->*FB|^
F->t|(E)

Ln 1, Col 3 100% Windows (CRLF) UTF-8
    
```

- **Output:**

```

E->TA
A->+TA|^
T->FB
B->*FB|^
F->t|(E)

FIRST OF E: ( t
FIRST OF A: + ^
FIRST OF T: ( t
FIRST OF B: * ^
FIRST OF F: ( t

FOLLOW OF E: $ )
FOLLOW OF A: $ )
FOLLOW OF T: $ ) +
FOLLOW OF B: $ ) +
FOLLOW OF F: $ ) * +

FIRST OF E->TA: ( t
FIRST OF A->+TA: +
FIRST OF A->^: ^
FIRST OF T->FB: ( t
FIRST OF B->*FB: *
FIRST OF B->^: ^
FIRST OF F->t: t
FIRST OF F->(E): (

***** LL(1) PARSING TABLE *****
-----
      $      (      )      *      +      t
E      E->TA
A      A->^      A->^      A->+TA
T      T->FB      T->FB
B      B->^      B->^      B->*FB      B->^
F      F->(E)      F->t
    
```

## Practical 7

### **Implementation of Recursive Descent Parser without backtracking**

**Input:** The string to be parsed.

**Output:** Whether string parsed successfully or not.

- **Code:**

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

char input[10];
int i,
error;

void E();
void T();
void Eprime();
void Tprime();
void F();

main()
{
printf("\nRecursive descent parsing for the following grammar\n");
printf("\nE -> E+T | T\nT -> T*F | F\nF -> (E) | id\n");
i = 0;
error = 0;
printf("\nEnter an arithmetic expression : "); // Eg: a+a*a gets(input);
E();
if (strlen(input) == i && error == 0)
    printf("\nParsing Successful..!!!\n");
else
    printf("\nError..!!!\n");
}
```

```
}

void E()
{
    T();
    Eprime();
}
void Eprime()
{
    if (input[i] == '+')
    {
        i++;
        T();
        Eprime();
    }
}
void T()
{
    F();
    Tprime();
}
void Tprime()
{
    if (input[i] == '*')
    {
        i++; F();
        Tprime();
    }
}
void F()
{
    if (isalnum(input[i])) i++;
    else if (input[i] == '(')
```

```
{  
i++; E());  
if(input[i] ==')') i++;  
  
else  
error = 1;  
}  
  
else  
error = 1;  
}
```

## Output:

Recursive descent parsing for the following grammar

```
E -> E+T | T  
T -> T*F | F  
F -> (E) | id
```

Enter an arithmetic expression : a+(a\*a)

Parsing Successful..!!!

Recursive descent parsing for the following grammar

```
E -> E+T | T  
T -> T*F | F  
F -> (E) | id
```

Enter an arithmetic expression : a++a

Error..!!!

**Practical 8**

**Write a C program to implement operator precedence parsing.**

- Code:**

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

char *input;
int i=0;
char lasthandle[6],stack[50],handles[][5]={")E(","E*E","E+E","i","E^E"};
//(E) becomes )E( when pushed to stack

int top=0,l;
char prec[9][9]={

    /*input*/

    /*stack  +  -  *  /  ^  i  (  )  $ */

    /* + */ '>','>','<','<','<','<','<','>','>',
    /* - */ '>','>','<','<','<','<','<','>','>',
    /* * */ '>','>','>','>','<','<','<','>','>',
    /* / */ '>','>','>','>','<','<','<','>','>',
    /* ^ */ '>','>','>','>','<','<','<','>','>',
    /* i */ '>','>','>','>','>','>','e','e','>','>',
```



```
/* ( */ '<', '<', '<', '<', '<', '<', '<', '>', 'e',
```

```
/* ) */ '>', '>', '>', '>', '>', '>', 'e', 'e', '>', '>',
```

```
/* $ */ '<', '<', '<', '<', '<', '<', '<', '<', '>',
```

```
};
```

```
int getIndex(char c)
```

```
{
```

```
switch(c)
```

```
{
```

```
case '+':return 0;
```

```
case '-':return 1;
```

```
case '*':return 2;
```

```
case '/':return 3;
```

```
case '^':return 4;
```

```
case 'i':return 5;
```

```
case '(':return 6;
```

```
case ')':return 7;
```

```
case '$':return 8;
```

```
}
```

```
}
```

```
int shift()
```

```
{
```

```
stack[++top]=*(input+i++);
```

```
stack[top+1]='\0';
```

```
}
```

```
int reduce()
```

```
{
int i,len,found,t;
for(i=0;i<5;i++)//selecting handles
{
len=strlen(handles[i]);
if(stack[top]==handles[i][0]&&top+1>=len)
{
found=1;
for(t=0;t<len;t++)
{
if(stack[top-t]!=handles[i][t])
{
found=0;
break;
}
}
if(found==1)
{
stack[top-t+1]='E';
top=top-t+1;
strcpy(lasthandle,handles[i]);
stack[top+1]='\0';
return 1;//successful reduction
}
}
}
return 0;
}
```

```
void dispstack()
```

```
{
```

```
int j;  
for(j=0;j<=top;j++)  
    printf("%c",stack[j]);  
}
```

```
void dispinput()  
{  
    int j;  
    for(j=i;j<l;j++)  
        printf("%c",*(input+j));  
}
```

```
void main()  
{  
    int j;  
  
    input=(char*)malloc(50*sizeof(char));  
    printf("\nEnter the string\n");  
    scanf("%s",input);  
    input=strcat(input,"$");  
    l=strlen(input);  
    strcpy(stack,"$");  
    printf("\nSTACK\tINPUT\tACTION");  
    while(i<=l)  
    {  
        shift();  
        printf("\n");  
        dispstack();  
        printf("\t");  
    }
```

```
    dispinput();
    printf("\tShift");
    if(prec[getindex(stack[top])][getindex(input[i])]=='>')
    {
        while(reduce())
        {
            printf("\n");
            dispstack();
            printf("\t");
            dispinput();
            printf("\tReduced: E->%s",lasthandle);
        }
    }

    if(strcmp(stack,"$$")==0)
        printf("\nAccepted;");
    else
        printf("\nNot Accepted;");
}
```

- Output:

```

Enter the string
i*(i+i)*i

STACK   INPUT   ACTION
$i      *(i+i)*i$   Shift
$E      *(i+i)*i$   Reduced: E->i
$E*     (i+i)*i$   Shift
$E*(    i+i)*i$   Shift
$E*(i   +i)*i$   Shift
$E*(E   +i)*i$   Reduced: E->i
$E*(E+  i)*i$   Shift
$E*(E+i )*i$   Shift
$E*(E+E )*i$   Reduced: E->i
$E*(E   )*i$   Reduced: E->E+E
$E*(E)  *i$     Shift
$E*E    *i$     Reduced: E->)E(
$E      *i$     Reduced: E->E*E
$E*     i$      Shift
$E*i    $       Shift
$E*E    $       Reduced: E->i
$E      $       Reduced: E->E*E
$E$     $       Shift
$E$     $       Shift
Accepted;
Process returned 10 (0xA)   execution time : 50.604 s
Press any key to continue.
|

```

## Practical 9

### Implement a C program to implement LALR parsing.

- **Code:**

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
void push(char *, int *, char);
char stacktop(char *);
void isproduct(char, char);
int ister(char);
int isnter(char);
int isstate(char);
void error();
void isreduce(char, char); char pop(char *, int *);
void printt(char *, int *, char[], int); void rep(char[], int);
struct action
{
char row[6][5];
};
const struct action A[12] = {
{"sf", "emp", "emp", "se", "emp", "emp"},
{"emp", "sg", "emp", "emp", "emp", "acc"},
{"emp", "rc", "sh", "emp", "rc", "rc"},
{"emp", "re", "re", "emp", "re", "re"},
{"sf", "emp", "emp", "se", "emp", "emp"},
{"emp", "rg", "rg", "emp", "rg", "rg"},
{"sf", "emp", "emp", "se", "emp", "emp"},
{"sf", "emp", "emp", "se", "emp", "emp"},
{"emp", "sg", "emp", "emp", "sl", "emp"},
```

```
{ "emp", "rb", "sh", "emp", "rb", "rb" },
{ "emp", "rb", "rd", "emp", "rd", "rd" },
{ "emp", "rf", "rf", "emp", "rf", "rf" } }; struct gotol
{
char r[3][4];
};
const struct gotol G[12] = {
{ "b", "c", "d" },
{ "emp", "emp", "emp" },
{ "emp", "emp", "emp" },
{ "emp", "emp", "emp" },
{ "i", "c", "d" },
{ "emp", "emp", "emp" },

{ "emp", "j", "d" },
{ "emp", "emp", "k" },
{ "emp", "emp", "emp" },
{ "emp", "emp", "emp" },
};
char ter[6] = { 'i', '+', '*', ')', '(', '$' };
char nter[3] = { 'E', 'T', 'F' }; char
states[12] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'm', 'j', 'k', 'l' };
char stack[100]; int top = -1; char temp[10]; struct grammar
{
char left; char right[5];
};
const struct grammar rl[6] = {
{ 'E', "e+T" },
{ 'E', "T" },
{ 'T', "T*F" },
{ 'T', "F" },
{ 'F', "(E)" },
{ 'F', "i" },
```

```
};  
void main()  
{  
char inp[80], x, p, dl[80], y, bl = 'a'; int i = 0, j, k, l, n, m, c, len;  
printf(" Enter the input :"); scanf("%s", inp);  
len = strlen(inp); inp[len] = '$';  
inp[len + 1] = '\0'; push(stack, &top, bl); printf("\n stack \t\t\t input"); printt(stack, &top, inp,  
i);  
do  
{  
x = inp[i];  
p = stacktop(stack); isproduct(x, p);  
if (strcmp(temp, "emp") == 0) error();  
if (strcmp(temp, "acc") == 0) break;  
else  
{  
  
if (temp[0] == 's')  
{  
push(stack, &top, inp[i]); push(stack, &top, temp[1]); i++;  
}  
else  
{  
if (temp[0] == 'r')  
{  
j = isstate(temp[1]); strcpy(temp, rl[j - 2].right); dl[0] = rl[j - 2].left;  
dl[1] = '\0';  
n = strlen(temp);  
for (k = 0; k < 2 * n; k++) pop(stack, &top);  
for (m = 0; dl[m] != '\0'; m++)  
push(stack, &top, dl[m]); l = top;  
y = stack[l - 1]; isreduce(y, dl[0]);  
for (m = 0; temp[m] != '\0'; m++) push(stack, &top, temp[m]);
```



```

}
}
}
printt(stack, &top, inp, i);
} while (inp[i] != '\0');
if (strcmp(temp, "acc") == 0) printf(" \n accept the input "); else
printf(" \n do not accept the input "); getch();
}
void push(char *s, int *sp, char item)
{
if (*sp == 100) printf(" stack is full "); else
{
*sp = *sp + 1; s[*sp] = item;
}
}
char stacktop(char *s)
{
char i;

i = s[top]; return i;
}
void isproduct(char x, char p)
{
int k, l;
k = ister(x);
l = isstate(p);
strcpy(temp, A[l - 1].row[k - 1]);
}
int ister(char x)
{
int i;
for (i = 0; i < 6; i++) if (x == ter[i]) return i + 1;
return 0;

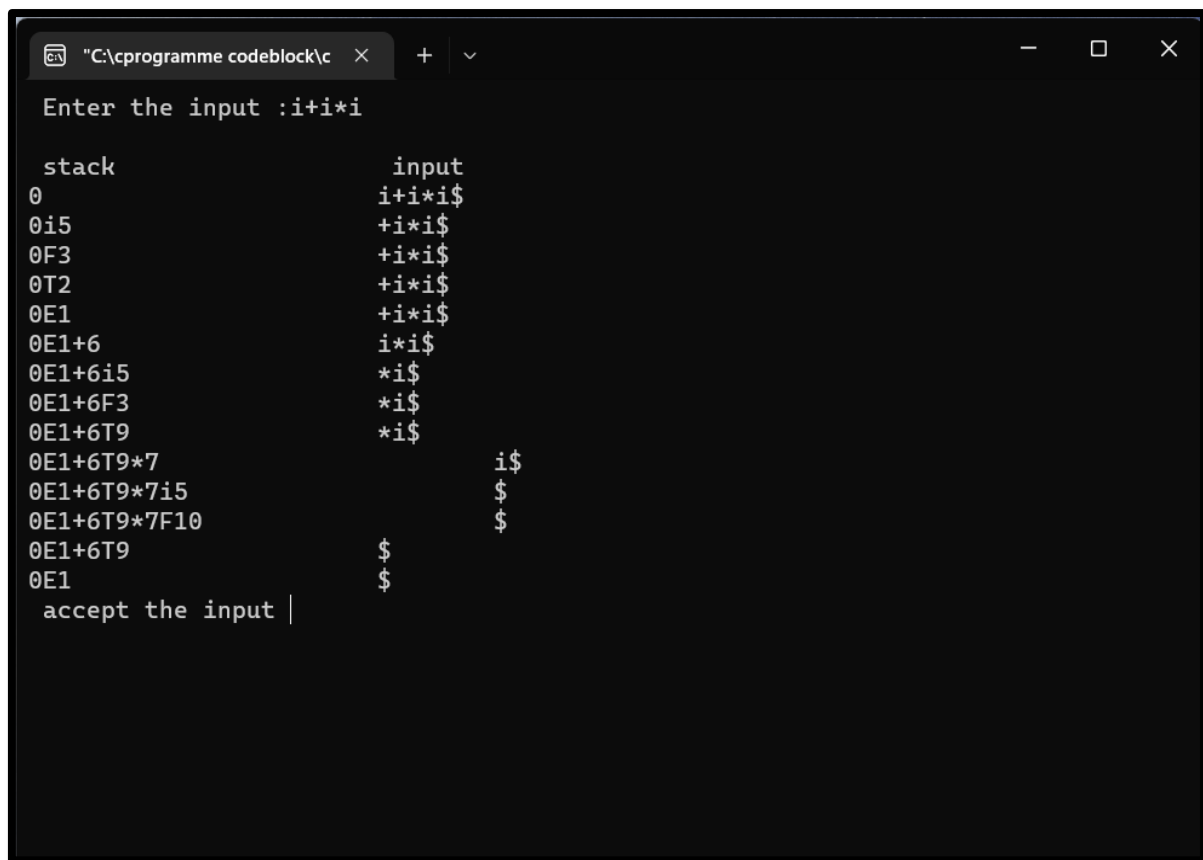
```

```
}  
int isnter(char x)  
{  
    int i;  
    for (i = 0; i < 3; i++) if (x == nter[i]) return i + 1;  
    return 0;  
}  
int isstate(char p)  
{  
    int i;  
    for (i = 0; i < 12; i++) if (p == states[i]) return i + 1;  
    return 0;  
}  
void error()  
{  
    printf(" error in the input "); exit(0);  
}  
void isreduce(char x, char p)  
{  
    int k, l;  
    k = isstate(x); l = isnter(p);  
    strcpy(temp, G[k - 1].r[l - 1]);  
}  
char pop(char *s, int *sp)  
{  
  
    char item;  
    if (*sp == -1)  
        printf(" stack is empty "); else  
    {  
        item = s[*sp];  
        *sp = *sp - 1;  
    }  
}
```

```
return item;
}
void printt(char *t, int *p, char inp[], int i)
{
    int r; printf("\n");
    for (r = 0; r <= *p; r++) rep(t, r); printf("\t\t\t");
    for (r = i; inp[r] != '\0'; r++)
        printf("%c", inp[r]);
}
void rep(char t[], int r)
{
    char c; c = t[r];
    switch (c)
    {
        case 'a': printf("0"); break; case 'b': printf("1"); break; case 'c': printf("2"); break; case 'd':
            printf("3"); break; case 'e': printf("4"); break; case 'f': printf("5"); break; case 'g': printf("6");
            break; case 'h':

            printf("7"); break; case 'm': printf("8"); break; case 'j': printf("9"); break; case 'k':
            printf("10"); break;
            case 'l': printf("11"); break; default:
            printf("%c", t[r]); break;
    }
}
```

- **Output:**



The screenshot shows a Code::Blocks IDE window with the title bar "C:\programme codeblock\c". The main area displays assembly code for the expression `i+i*i`. The code is organized into two columns: `stack` and `input`. The `stack` column shows memory addresses and their corresponding values, while the `input` column shows the input values. The code is as follows:

```
Enter the input :i+i*i

stack      input
0          i+i*i$
0i5        +i*i$
0F3        +i*i$
0T2        +i*i$
0E1        +i*i$
0E1+6      i*i$
0E1+6i5    *i$
0E1+6F3    *i$
0E1+6T9    *i$
0E1+6T9*7  i$
0E1+6T9*7i5 $
0E1+6T9*7F10 $
0E1+6T9    $
0E1        $
accept the input |
```

## Practical 10

### To Study about Yet Another Compiler-Compiler (YACC).

- YACC stands for Yet Another Compiler Compiler.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

#### Input File :

```
/* definitions */  
  
....  
  
%%  
  
/* rules */  
  
....  
  
%%  
  
/* auxiliary routines */  
  
....
```

#### Definition Part:

The definition part includes information about the tokens used in the syntax definition. Yacc also recognizes single characters as tokens. The definition part can include C code external to the definition of the parser and variable declarations, within %{ and %} in the first column.

**Rule Part:**

The rules part contains grammar definition in a modified BNF form. Actions is C code in { } and can be embedded inside (Translation schemes).

**Auxiliary Routines Part:**

The auxiliary routines part is only C code. It includes function definitions for every function needed in rules part. It can also contain the main() function definition if the parser is going to be run as a program. The main() function must call the function yyparse().

YACC input file generally finishes with: .y

**Output Files :**

The output of YACC is a file named y.tab.c

If it contains the main() definition, it must be compiled to be executable. Otherwise, the code can be an external function definition for the function int yyparse() .

If called with the -d option in the command line, Yacc produces as output a headerfile y.tab.h with all its specific definitions.

If called with the -v option, Yacc produces as output a file y.output containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts

**For Compiling YACC Program :**

1. Write lex program in a file file.l and yacc in a file file.y
2. Open Terminal and Navigate to the Directory where you have saved the files.
3. type lex file.l
4. type yacc file.y
5. type cc lex.yy.c y.tab.h -ll
6. type ./a.out

**Example:**

Yacc File (.y):

```
%{
    #include<ctype.h>
    #include<stdio.h> #define
    YYSTYPE double

}%
%%

Lines : Lines S '\n' { printf("OK \n");}
      | S '\n'
      | error '\n' {yyerror("Error: reenter last line:");
                    yyerrok; };

S      : '(' S ')'
      | '[' S ']'
      |

%%

#include "lex.yy.c"

Void yyerror(char *s){
    Fprintf (stderr, "%s\n", s);
}

int main(void){
    Return yyparse();
}
```

Lex File (.l):

```
%{
%}
```

%%

[ \t] {}

\n]. { return yytext[0];}

%%



## Practical 11

**Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, \* and / .**

- **Code**

**test.l file:**

test.l file:

```
%option noyywrap %{\n#include "stdio.h" #include "test.tab.h" extern int yylval;\n%\n%%\n\n[0-9]+ {yylval=atoi(yytext);return number;} \\+ { return plus;}\n\\- { return minus;}\n\\* { return multiply;}\n\\/ {return divide;}\n. {return yytext[0];} [\\t]+;\n\\n return 0;\n%%
```

**test.y file:**

```
%{\n#include "stdio.h" int result=0;\nvoid yyerror(const char *str)\n{\nfprintf(stderr,"error: %s\\n",str); } int yywrap(void)\n{\nreturn 1;
```

```
}  
%}  
%token number plus minus divide multiply %left plus minus  
%left multiply divide  
%right '^'  
%nonassoc UMINUS %% ae: exp {result=$1;} ; exp: number { $$ = $1;}  
| exp minus exp { $$ = $1 - $3;} | exp plus exp { $$ = $1 + $3;} | exp divide exp { if($3==0)  
yyerror("divide by zero"); else $$ = $1 / $3;}  
| minus exp %prec UMINUS { $$ = -$2; } | exp multiply exp { $$ = $1 * $3 ;} | exp'^'exp{ }  
;  
%%  
  
#include "math.h" int main(void)  
{  
yyparse(); printf("=%d",result);  
}
```

- **Output**



```
Command Prompt  
C:\Users\Administrator123 \CD>lex test.l  
C:\Users\Administrator123 \CD>yacc test.y  
C:\Users\Administrator123 \CD>gcc lex.yy.c test.tab.c  
C:\Users\Administrator123 \CD>a.exe  
7+10/2  
=12  
C:\Users\Administrator123 \CD>
```

## Practical 12

### Generate 3-tuple intermediate code for given infix expression

- **Code:**

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

void pm();
void plus();
void div();
int i,ch,j,l,addr=100;
char ex[10], exp[10],exp1[10],exp2[10],id1[5],op[5],id2[5];
void main()
{
clrscr();
while(1)
{
printf("\n1.assignment\n2.arithmetic\n3.relational\n4.Exit\nEnter the choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\nEnter the expression with assignment operator:");
scanf("%s",exp);
l=strlen(exp);
exp2[0]='\0';
i=0;
while(exp[i]!='=')
{
i++;
}
```

```
strncat(exp2,exp,i);
strrev(exp);
exp1[0]='\0';
strncat(exp1,exp,l-(i+1));
strrev(exp1);
printf("Three address code:\ntemp=%s\n%s=temp\n",exp1,exp2);
break;
```

case 2:

```
printf("\nEnter the expression with arithmetic operator:");
scanf("%s",ex);
strcpy(exp,ex);
l=strlen(exp);
exp1[0]='\0';
```

```
for(i=0;i<l;i++)
{
if(exp[i]=='+'||exp[i]=='-')
{
if(exp[i+2]=='/'||exp[i+2]=='*')
{
pm();
break;
}
else
{
plus();
break;
}
}
else if(exp[i]=='/'||exp[i]=='*')
{
div();
```

```
break;
}
}
break;

case 3:
printf("Enter the expression with relational operator");
scanf("%s%s%s",&id1,&op,&id2);
if(((strcmp(op,"<")==0)||(strcmp(op,">")==0)||(strcmp(op,"<=")==0)||(strcmp(op,">=")==0)||
(strcmp(op,"==")==0)||(strcmp(op,"!=")==0))==0)
printf("Expression is error");
else
{
printf("\n%d\tif %s%s%s goto %d",addr,id1,op,id2,addr+3);
addr++;
printf("\n%d\tT:=0",addr);
addr++;
printf("\n%d\tgoto %d",addr,addr+2);
addr++;
printf("\n%d\tT:=1",addr);
}
break;
case 4:
exit(0);
}
}
}

void pm()
{
strrev(exp);
j=l-i-1;
strncat(exp1,exp,j);
strrev(exp1);
```

```

printf("Three address code:\ntemp=%s\ntemp1=%c%ctemp\n",exp1,exp[j+1],exp[j]);
}
void div()
{
strncat(exp1,exp,i+2);
printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}
void plus()
{
strncat(exp1,exp,i+2);
printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n",exp1,exp[i+2],exp[i+3]);
}

```

- **Output:**

```

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice: 1

Enter the expression with assignment operator: a=b
Three address code:
temp=b
a=temp

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice: 2

Enter the expression with arithmetic operator: a+b-c
Three address code:
temp=a+b
temp1=temp-c

1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice: 3
Enter the expression with relational operator: a
<=
b

100    if a<=b goto 103
101    T:=0
102    goto 104
103    T:=1
1.assignment
2.arithmetic
3.relational
4.Exit
Enter the choice: 4

```

### Practical 13

## **Extract Predecessor and Successor from given Control Flow Graph.**

- **Code:**

```
#include <iostream>
using namespace std;

// BST Node
struct Node
{
    int key;
    struct Node *left, *right;
};

// This function finds predecessor and successor of key in BST.
// It sets pre and suc as predecessor and successor respectively
void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{
    // Base case
    if (root == NULL) return ;

    // If key is present at root
    if (root->key == key)
    {
        // the maximum value in left subtree is predecessor
        if (root->left != NULL)
        {
            Node* tmp = root->left;
            while (tmp->right)
                tmp = tmp->right;
        }
    }
}
```

```
        pre = tmp ;
    }

    // the minimum value in right subtree is successor
    if (root->right != NULL)
    {
        Node* tmp = root->right ;
        while (tmp->left)
            tmp = tmp->left ;
        suc = tmp ;
    }
    return ;
}

// If key is smaller than root's key, go to left subtree
if (root->key > key)
{
    suc = root ;
    findPreSuc(root->left, pre, suc, key) ;
}
else // go to right subtree
{
    pre = root ;
    findPreSuc(root->right, pre, suc, key) ;
}
}

// A utility function to create a new BST node
Node *newNode(int item)
{
    Node *temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
```



```
        return temp;
    }

/* A utility function to insert a new node with given key in BST */
Node* insert(Node* node, int key)
{
    if (node == NULL) return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);
    return node;
}

// Driver program to test above function
int main()
{
    int key = 65; //Key to be searched in BST

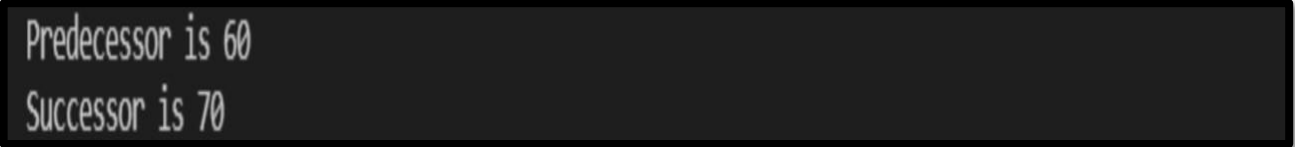
    /* Let us create following BST
            50
           /  \
          30   70
         /\  /\
        20 40 60 80 */
    Node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
```

```
Node* pre = NULL, *suc = NULL;

findPreSuc(root, pre, suc, key);
if (pre != NULL)
    cout << "Predecessor is " << pre->key << endl;
else
    cout << "No Predecessor";

if (suc != NULL)
    cout << "Successor is " << suc->key;
else
    cout << "No Successor";
return 0;
}
```

- **Output**



```
Predecessor is 60
Successor is 70
```