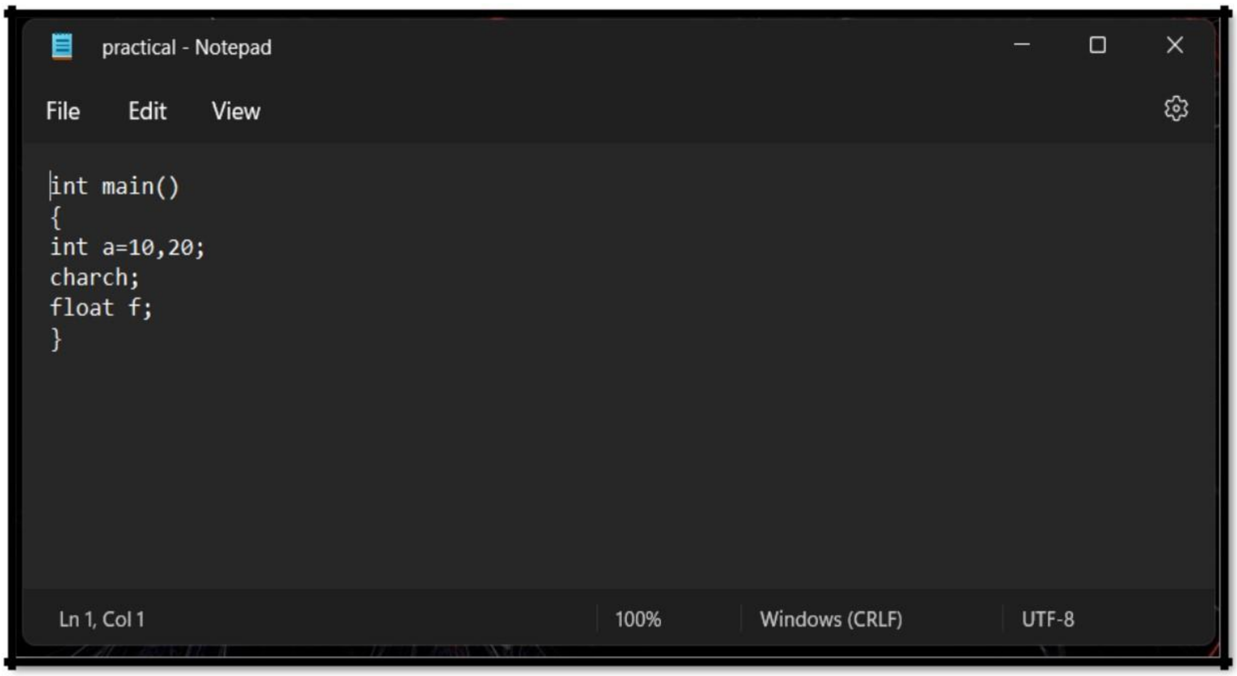
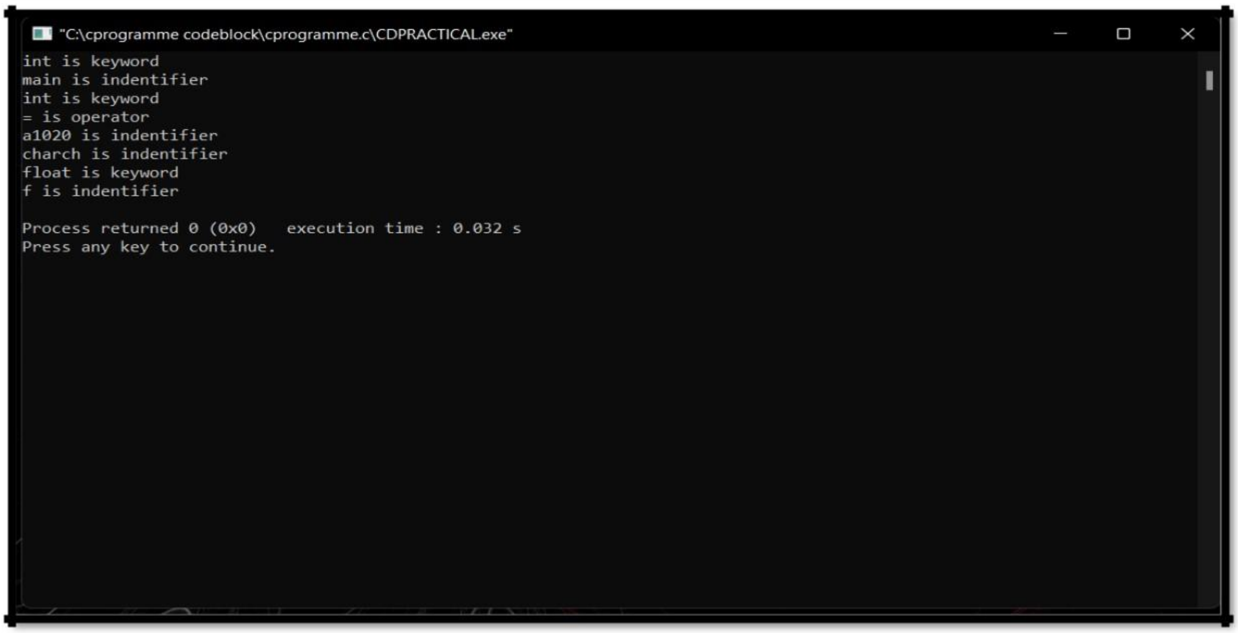
1. Design a lexical analyzer for a given language and the lexical analyzer should ignore redundant spaces, tabs and newlines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long, you may restrict the length to some reasonable value. Simulate the same in C language.

```
#include <stdio.h>
#include <string.h>
int isOperator(char c[])
{
    if (c[0] == '+' || c[0] == '-' || c[0] == '*' ||
        c[0] == '/' || c[0] == '>' || c[0] == '<' ||
        c[0] == '=')
        return 1;
    else
    {
        return 0;
    }
}
int isDelimeter(char c[])
{
    if (c[0] == ',' || c[0] == ';')
        return 1;
    else
    {
        return 0;
    }
}
int isIdentifier(char c[])
{
    if ((c[0] >= 'a' && c[0] <= 'z') || (c[0] >= 'A' && c[0] <= 'Z') || (c[0] == '_'))
        return 1;
    else
    {
        return 0;
    }
}
int isKeyword(char *str)
{
    if (!strcmp(str, "if") || !strcmp(str, "else") || !strcmp(str, "while") || !strcmp(str,
"do") ||
        !strcmp(str, "break") || !strcmp(str, "continue") || !strcmp(str, "int") ||
!strcmp(str, "double") ||
        !strcmp(str, "float") || !strcmp(str, "return") || !strcmp(str, "char") ||
!strcmp(str, "case") ||
        !strcmp(str, "char") || !strcmp(str, "sizeof") || !strcmp(str, "long") ||
!strcmp(str, "short") ||
        !strcmp(str, "typedef") || !strcmp(str, "switch") || !strcmp(str, "unsigned") ||
!strcmp(str, "void") || !strcmp(str, "static") || !strcmp(str, "struct") || !strcmp(str,
"goto"))
        return 1;
    else
    {
        return 0;
    }
}
int isConstant(char *str)
{
    int i, len = strlen(str);
    for (i = 0; i < len; i++)
    {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2' && str[i] != '3' && str[i] != '4'
&& str[i] != '5' && str[i] != '6' && str[i] != '7' && str[i] != '8' && str[i] != '9' &&
str[i] != '.' || (str[i] == '-' && i > 0))
```

```
        {
            return (0);
        }
        else
        {
            return 1;
        }
    }
}
int main()
{
    char str[100];
    char c;
    int i = 0;
    FILE *file = fopen("temp.txt", "r");
    while ((c = fgetc(file)) != EOF)
    {
        if (c != ' ')
        {
            // printf("%c\n", c);
            str[i] = c;
            i++;
        }
        else
        {
            str[i] = '\0';
            // printf("%s\n", str);
            if (isOperator(str) == 1)
            {
                printf("'%s' IS A OPERATOR\n", str);
            }
            else if (isKeyword(str) == 1)
            {
                printf("'%s' IS A KEYWORD\n", str);
            }
            else if (isIdentifier(str) == 1)
            {
                printf("'%s' IS A IDENTIFIRE\n", str);
            }
            else if (isDelimeter(str) == 1)
            {
                printf("'%s' IS A DELIMETER\n", str);
            }
            else if (isConstant(str) == 1)
            {
                printf("'%s' IS A CONSTANT\n", str);
            }
            i = 0;
        }
    }
    return 0;
}
```



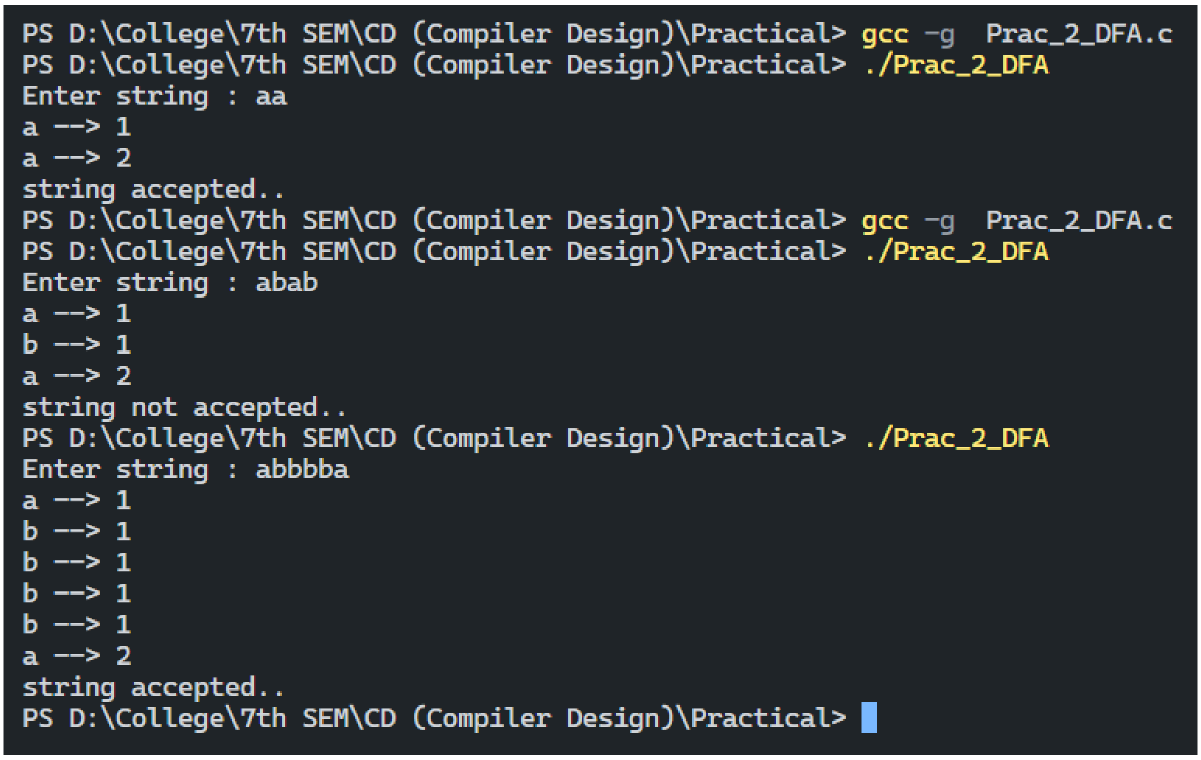
```
int main()
{
int a=10,20;
charch;
float f;
}
```



```
int is keyword
main is indentifier
int is keyword
= is operator
a1020 is indentifier
charch is indentifier
float is keyword
f is indentifier

Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

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

```
#include <stdio.h>
int main()
{
    int dfa = 0;
    char input[50];
    printf("Enter string : ");
    scanf("%s", input);

    for (int i = 0; input[i] != '\0'; i++)
    {
        if (dfa == 0 && input[i] == 'a')
        {
            dfa = 1;
            printf("%c --> %d\n", input[i], dfa);
        }
        else if (dfa == 0 && input[i] == 'b')
        {
            dfa = 3;
            // printf("String Dose not match with RE..\n");
            // return 0;
        }
        else if (dfa == 1 && input[i] == 'b')
        {
            dfa = 1;
            printf("%c --> %d\n", input[i], dfa);
        }
        else if (dfa == 1 && input[i] == 'a')
        {
            dfa = 2;
            // printf("%c\n", input[i]);             //printf("%d\n", dfa);
            printf("%c --> %d\n", input[i], dfa);
        }
        else if (dfa == 2 && input[i] == 'b')
        {
            dfa = 3;
            // printf("String Dose not match with RE..\n");             // return 0;
        }
        else if (dfa == 2 && input[i] == 'a')
        {
            dfa = 3;
            // printf("String Dose not match with RE..\n");
            // return 0;
        }
    }
    if (dfa != 2)
    {
        printf("string not accepted..\n");
    }
    else
    {
        printf("string accepted..\n");
    }
    return 0;
}
```


```
PS D:\College\7th SEM\CD (Compiler Design)\Practical> gcc -g  Prac_2_DFA.c
PS D:\College\7th SEM\CD (Compiler Design)\Practical> ./Prac_2_DFA
Enter string : aa
a --> 1
a --> 2
string accepted..
PS D:\College\7th SEM\CD (Compiler Design)\Practical> gcc -g  Prac_2_DFA.c
PS D:\College\7th SEM\CD (Compiler Design)\Practical> ./Prac_2_DFA
Enter string : abab
a --> 1
b --> 1
a --> 2
string not accepted..
PS D:\College\7th SEM\CD (Compiler Design)\Practical> ./Prac_2_DFA
Enter string : abbbba
a --> 1
b --> 1
b --> 1
b --> 1
b --> 1
a --> 2
string accepted..
PS D:\College\7th SEM\CD (Compiler Design)\Practical> 
```

3. To Study about Lexical Analyzer Generators (LEX).

- It is the first step of compiler design; it takes the input as a stream of characters and gives the output as tokens also known as tokenization. The tokens can be classified into identifiers, Separators, Keywords, Operators, Constant and Special Characters.
- It has three phases:
  - Tokenization: It takes the stream of characters and converts it into tokens.
  - Error Messages: It gives errors related to lexical analysis such as exceeding length, unmatched string, etc.
  - Eliminate Comments: Eliminates all the spaces, blank spaces, new lines, and indentations.

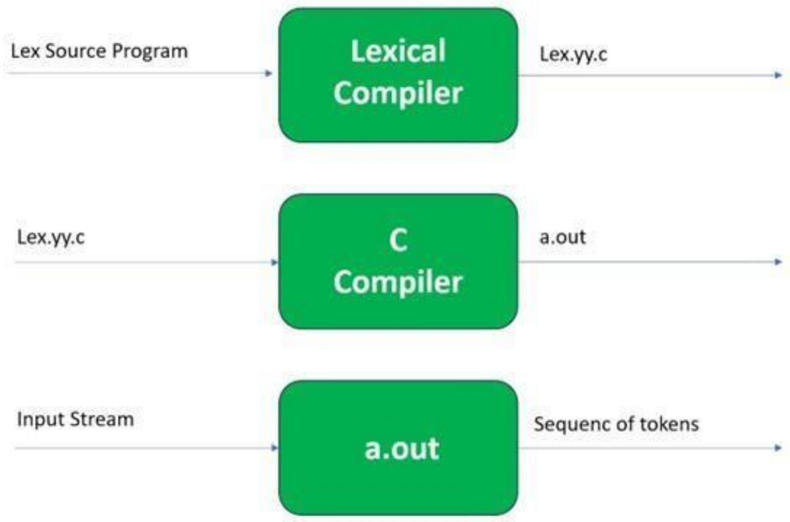
Lex

- Lex is a tool or a computer program that generates Lexical Analyzers (converts the stream of characters into tokens). The Lex tool itself is a compiler.
- The Lex compiler takes the input and transforms that input into input patterns. It is commonly used with YACC (Yet Another Compiler Compiler). It was written by Mike Lesk and Eric Schmidt.

Function of Lex

- In the first step the source code which is in the Lex language having the file name 'File.l' gives as input to the Lex Compiler commonly known as Lex to get the output as lex.yy.c.
- After that, the output lex.yy.c will be used as input to the C compiler which gives the output in the form of an 'a.out' file, and finally, the output file a.out will take the stream of character and generates tokens as output.

Lex File Format



A Lex program consists of three parts and is separated by %% delimiters: -

Declarations
%%
Translation rules
%%
Auxiliary procedures

Declarations: The declarations include declarations of variables.
Transition rules: These rules consist of Pattern and Action.
Auxiliary procedures: The Auxiliary section holds auxiliary functions used in the actions

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.

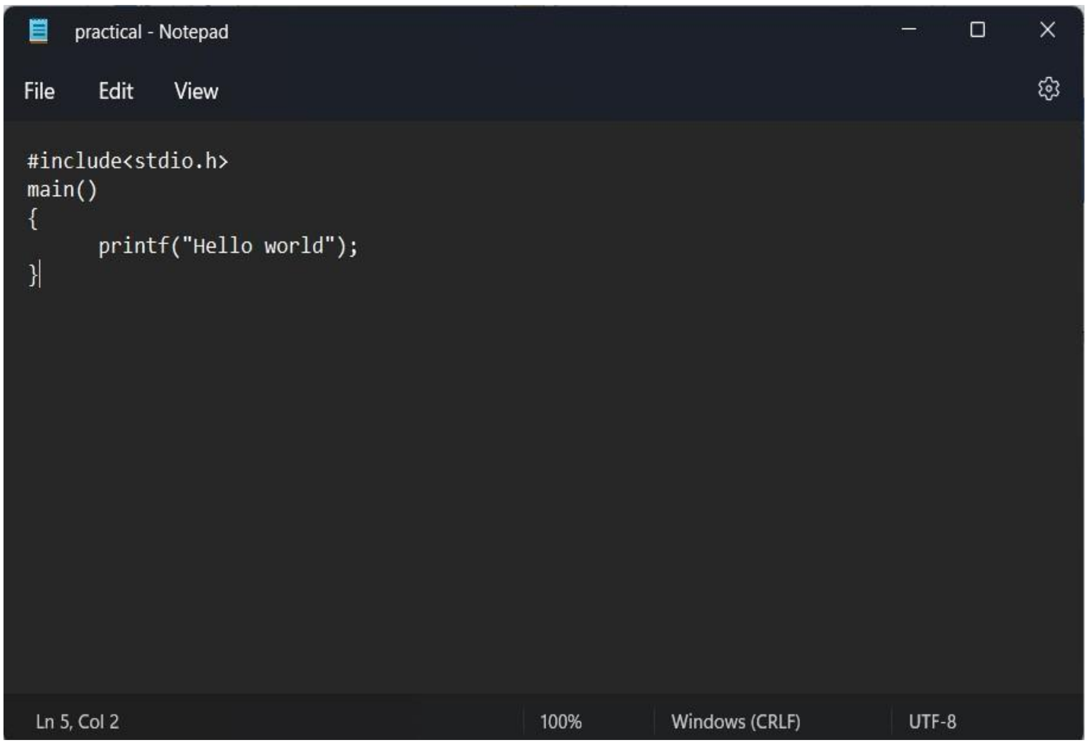
i. Generate Histogram of Words

j. Caesar Cypher

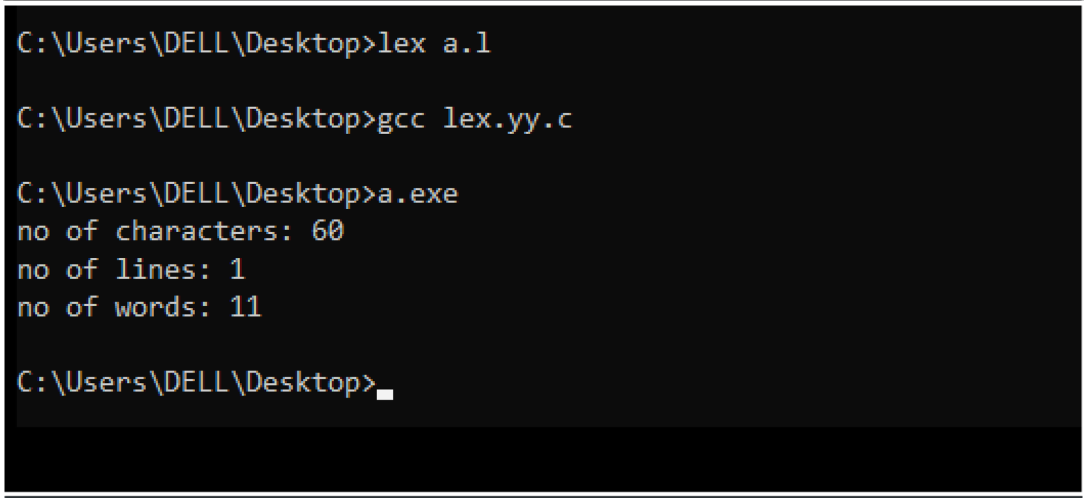
k. **Check weather given statement is compound or simple**

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

```
%
{
    #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+yyleng;}

. {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;
}
```



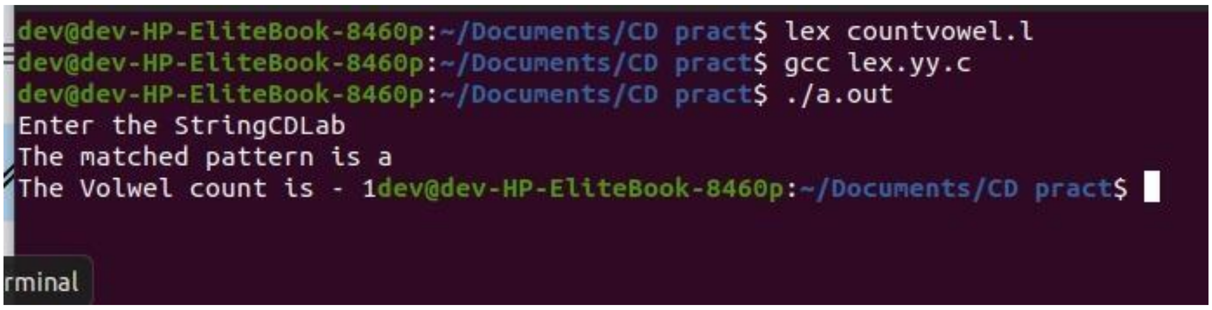
```
#include<stdio.h>
main()
{
	printf("Hello world");
}
```



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

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.

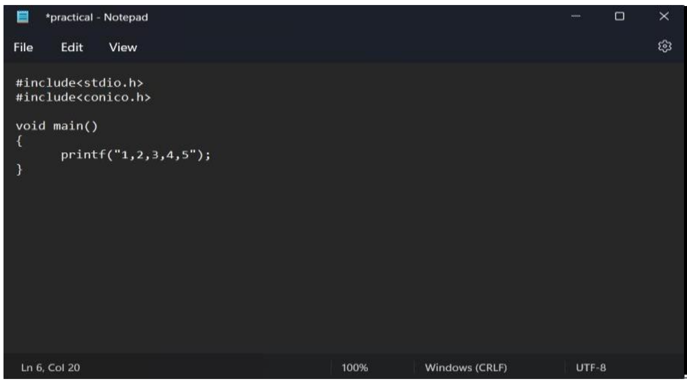
```
%
{
    int vow_count=0;
    int const_count =0;
%}

%%
[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;
}
```



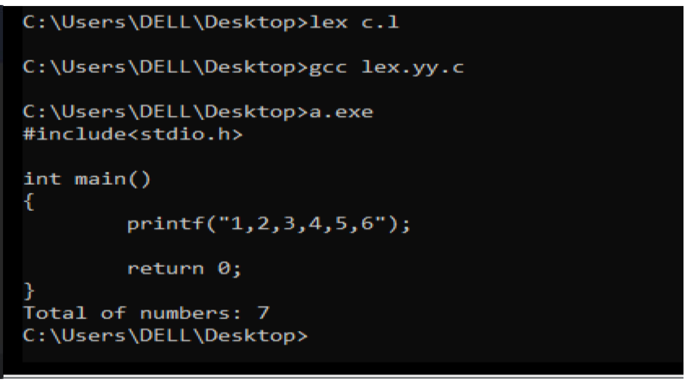
```
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ lex countvowel.l
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ gcc lex.yy.c
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ ./a.out
Enter the StringCDLab
The matched pattern is a
The Volwel count is - 1dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$
```

c. Print only numbers from the given file.

```
%
{
    #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);
}
```



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

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

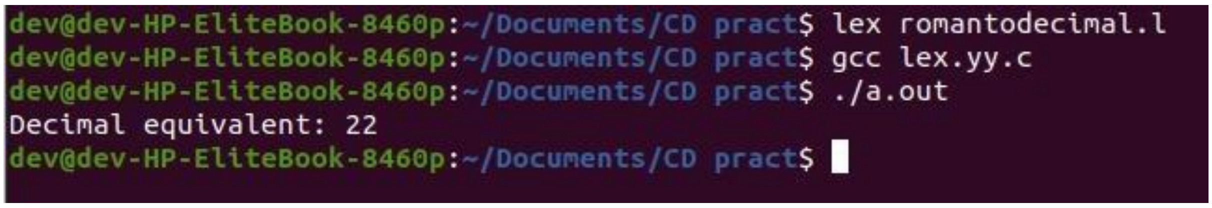


```
C:\Users\DELL\Desktop>lex c.l

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>
```

d. Convert Roman to Decimal

```
%{
    #include <stdio.h>
    int romanToDecimal(char *roman);
%}

%%
[IVXLCDM]+
{
    int decimal = romanToDecimal(yytext);
    printf("%s => %d\n", yytext, decimal);
}
.|\n;
%%
```

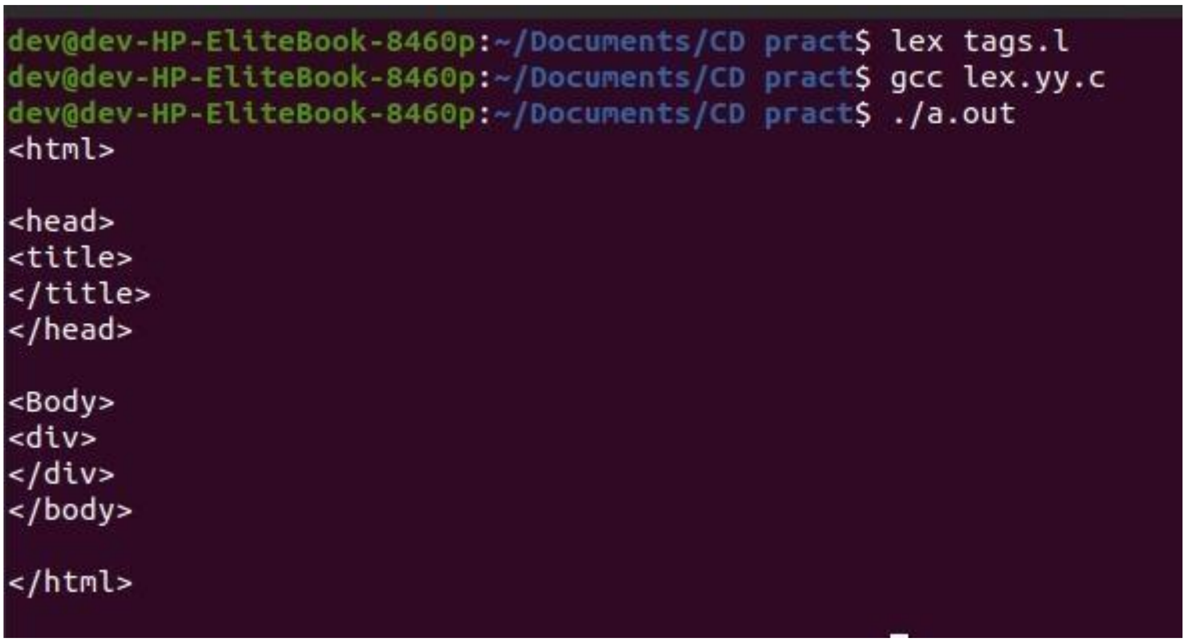
```
int main(int argc, char* argv[])
{
    yylex();
    return 0;
}

int romanToDecimal(char *roman) {
    int values [] = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
    char* numerals [] = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV",
"I"};

    int result = 0;
    int i = 0;
    while (*roman) {
        for (int j = 0; j < 13; j++) {
            if (strncmp(roman, numerals[j], strlen(numerals[j])) == 0) {
                result += values[j];
                roman += strlen(numerals[j]);
                break;
            }
        }
    }
    return result;
}
```



```
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ lex romantodecimal.l
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ gcc lex.yy.c
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ ./a.out
Decimal equivalent: 22
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ 
```

e. Print all HTML tags from the given file.

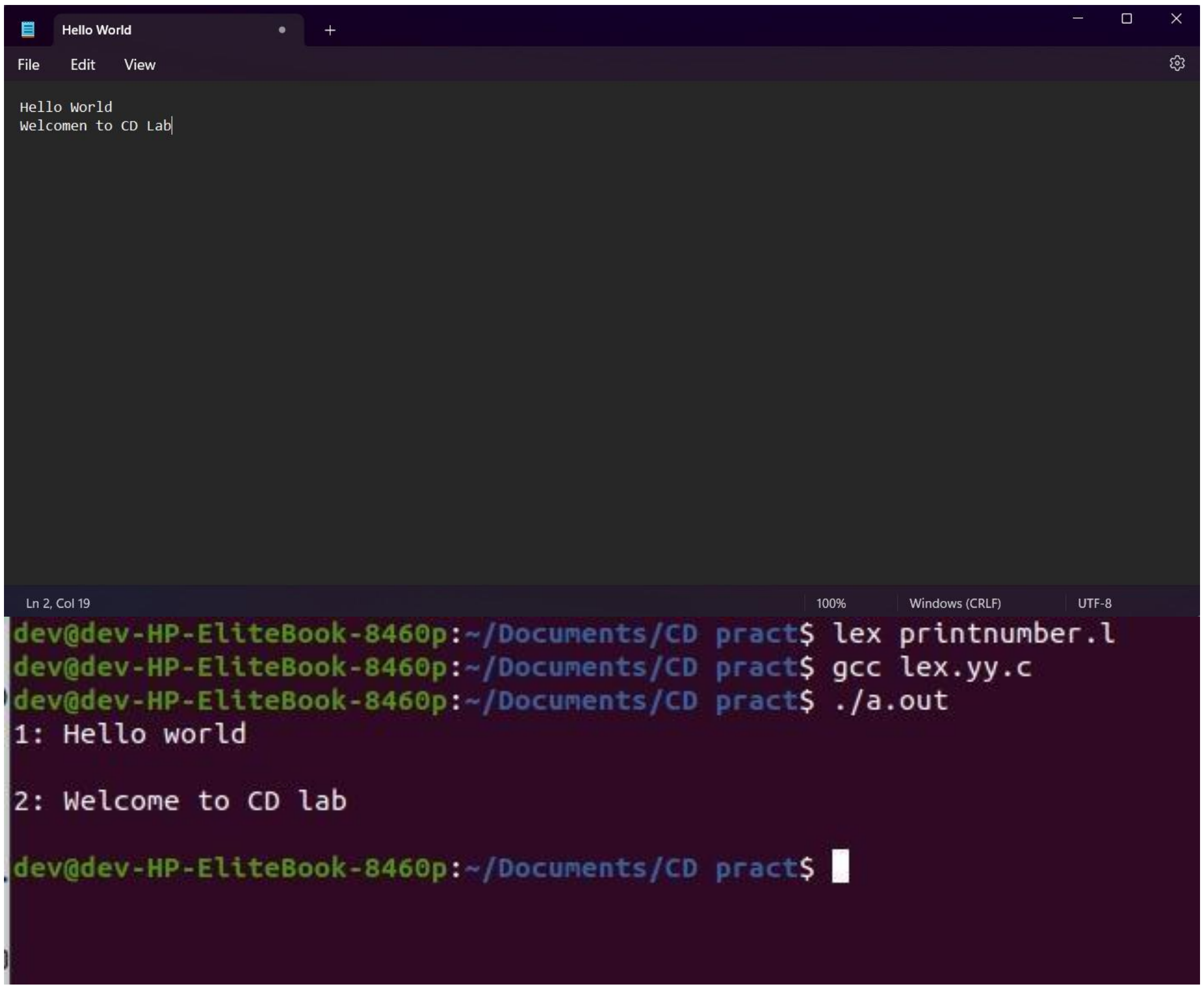
```
%{
%}

%%
"<"[^>]*> {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;
}
```



```
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ lex tags.l
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ gcc lex.yy.c
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ ./a.out
<html>

<head>
<title>
</title>
</head>

<Body>
<div>
</div>
</body>

</html>
```

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

```
%{
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();
}
```

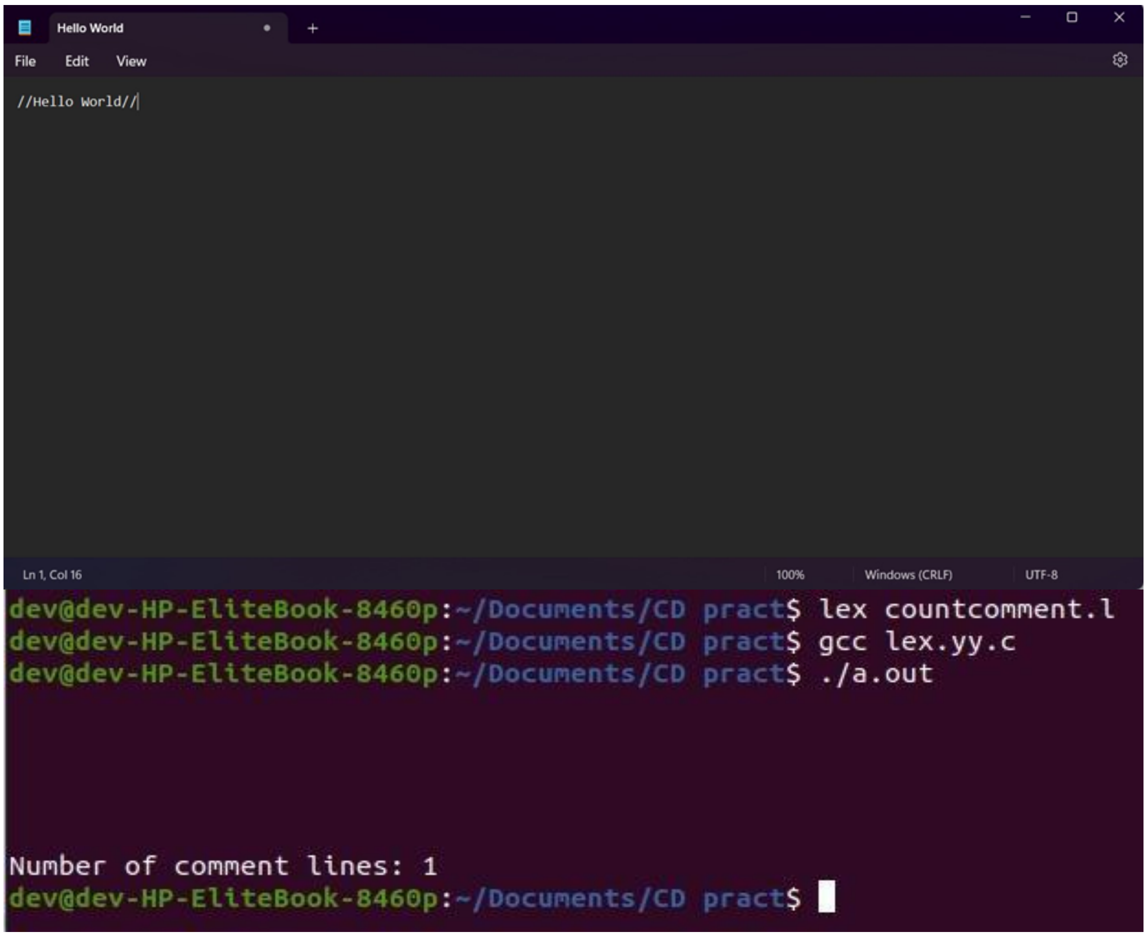


```
Hello World
Welcomen to CD Lab
```

```
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ lex printnumber.l
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ gcc lex.yy.c
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ ./a.out
1: Hello world

2: Welcome to CD lab

dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$
```

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

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

%%
"/*"[.]*"*/" {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;
}
```

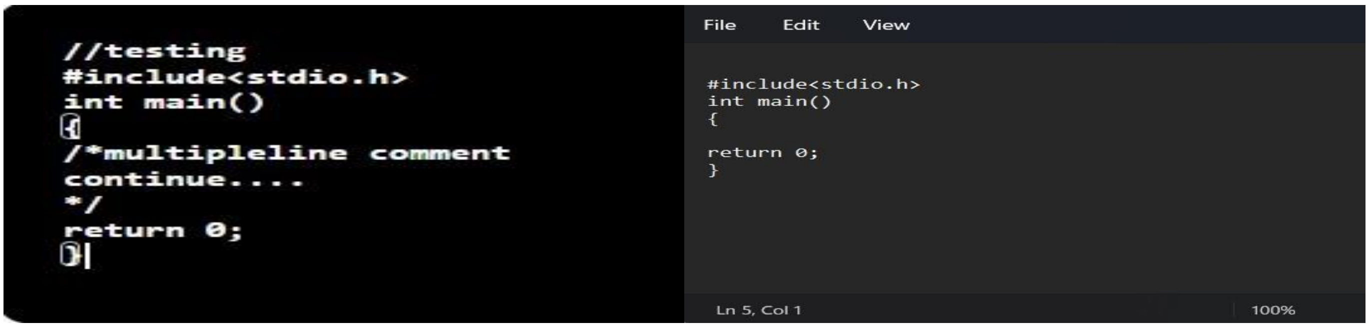


```
//Hello World//
```

```
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ lex countcomment.l
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ gcc lex.yy.c
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ ./a.out



Number of comment lines: 1
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ 
```

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

```
%
{
    #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;
}
```

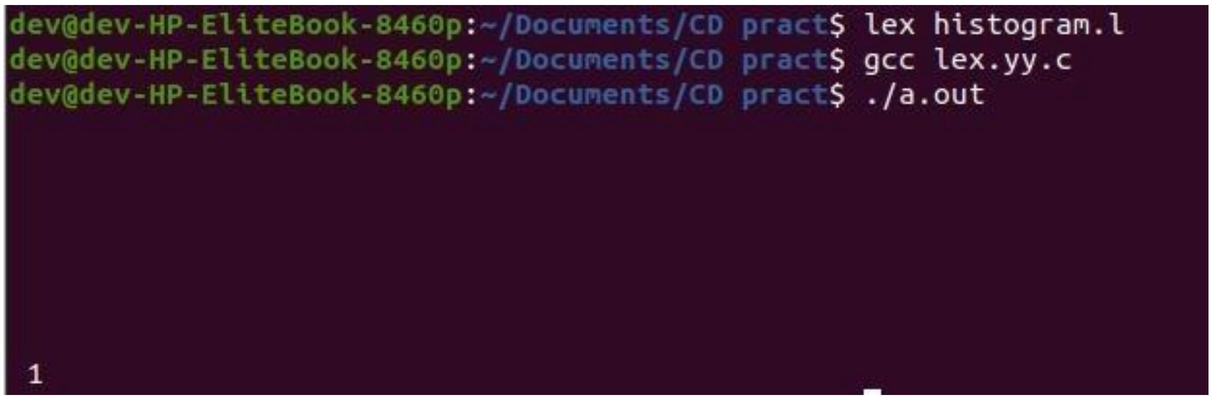


```
//testing
#include<stdio.h>
int main()
{
/*multipleline comment
continue....
*/
return 0;
}
```

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

return 0;
}
```

i. Generate Histogram of Words

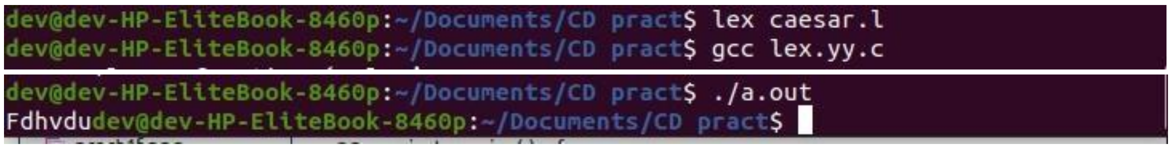
```
%
{
    #include<stdio.h>
    #include<string.h>
    char word [] = "geeks";
    int count = 0;
%}

%%
[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);
}
```



```
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ lex histogram.l
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ gcc lex.yy.c
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ ./a.out







 1
```

j. Caesar Cipher

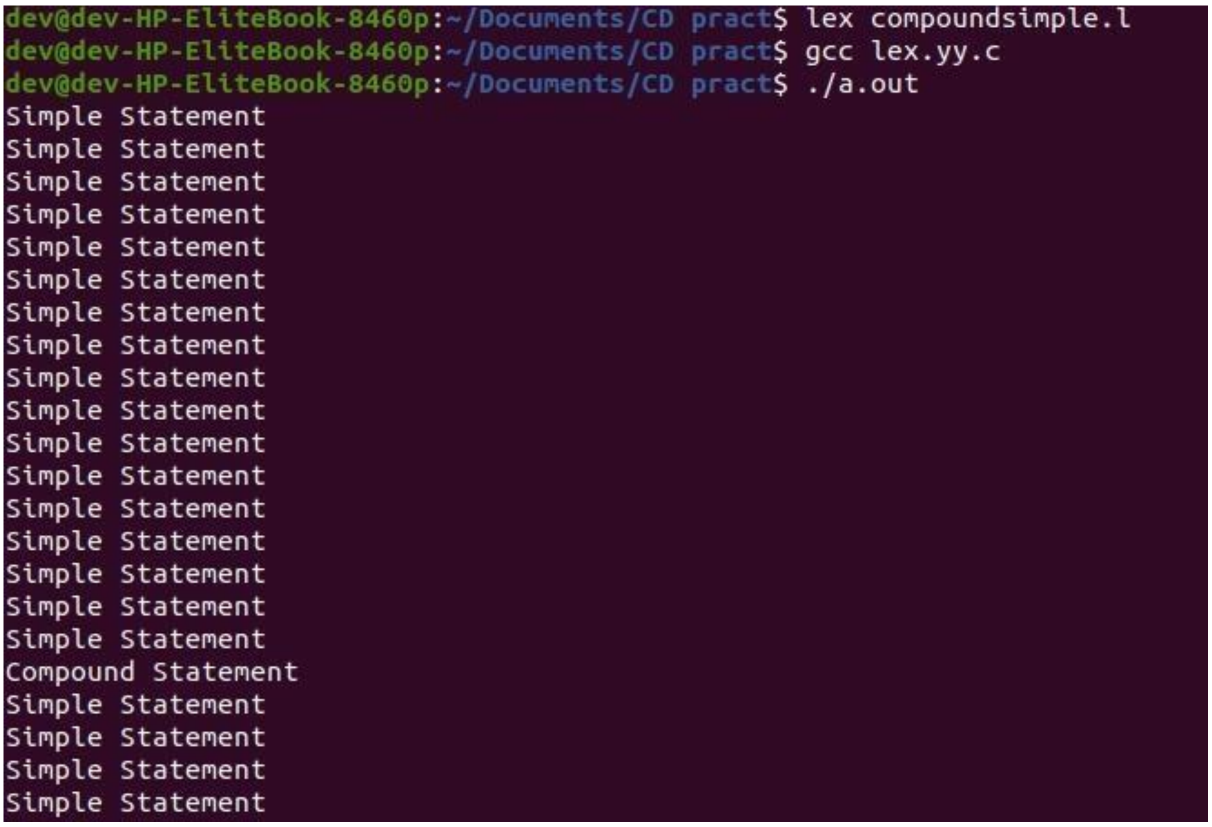
```
%%
[a-z]
{
    char ch = yytext[0];
    ch += 3;
    if (ch> 'z')
        ch -= ('z'+1- 'a');
        printf ("%c" ,ch );
}
[A-Z]
{
    charch = yytext[0];
    ch += 3;
    if (ch> 'Z')
        ch -= ('Z'+1- 'A');
        printf("%c",ch);
}
%%
```



```
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ lex caesar.l
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ gcc lex.yy.c
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ ./a.out
Fdhvdudev@dev-HP-EliteBook-8460p:~/Documents/CD pract$
```

k. Check weather given statement is compound or simple

```
%{
    #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;
}
```



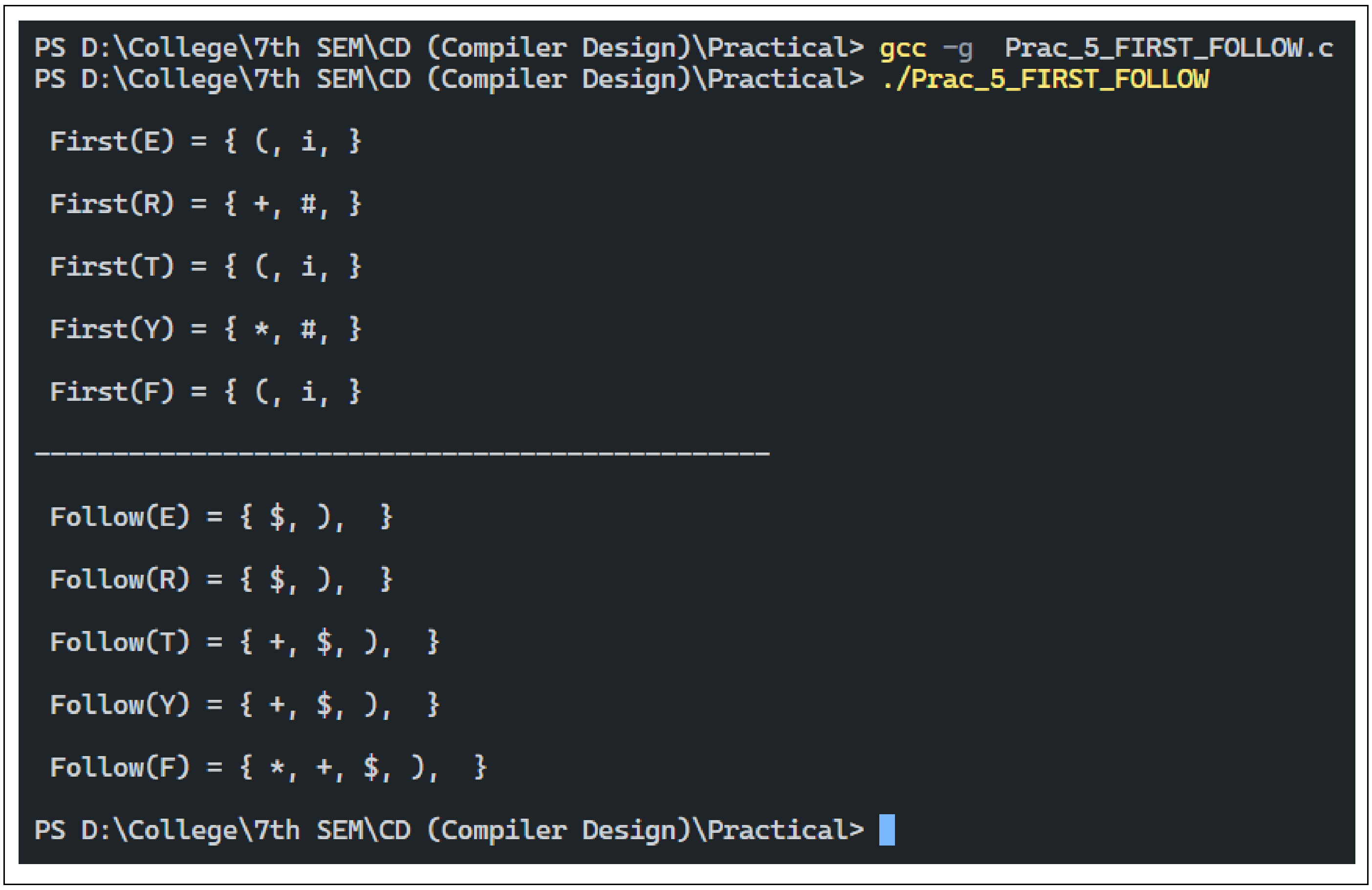
```
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ lex compoundsimple.l
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ gcc lex.yy.c
dev@dev-HP-EliteBook-8460p:~/Documents/CD pract$ ./a.out
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
Compound Statement
Simple Statement
Simple Statement
Simple Statement
Simple Statement
```

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.

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

// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);

// Function to calculate First
void findfirst(char, int, int);

int count, n = 0;

// Stores the final result
// of the First Sets
char calc_first[10][100];

// Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;

// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char **argv)
{
	int jm = 0;
	int km = 0;
	int i, choice;
	char c, ch;
	count = 8;

	strcpy(production[0], "E=TR");
	strcpy(production[1], "R=+TR");
	strcpy(production[2], "R=#");
	strcpy(production[3], "T=FY");
	strcpy(production[4], "Y=*FY");
	strcpy(production[5], "Y=#");
	strcpy(production[6], "F=(E)");
	strcpy(production[7], "F=i");

	int kay;
	char done[count];
	int ptr = -1;

	// Initializing the calc_first array
```

```
    for(k = 0; k < count; k++) {
        for(kay = 0; kay < 100; kay++) {
            calc_first[k][kay] = '!';
        }
    }
    int point1 = 0, point2, x;
    for(k = 0; k < count; k++)
    {
        c = production[k][0];
        point2 = 0;
        x = 0;

        // Checking if First of c has
        // already been calculated
        for(kay = 0; kay <= ptr; kay++)
            if(c == done[kay])
                x = 1;

        if (x == 1)
            continue;

        // Function call
        findfirst(c, 0, 0);
        ptr += 1;

        // Adding c to the calculated list
        done[ptr] = c;
        printf("\n First(%c) = { ", c);
        calc_first[point1][point2++] = c;

        // Printing the First Sets of the grammar
        for(i = 0 + jm; i < n; i++) {
            int lark = 0, chk = 0;
            for(lark = 0; lark < point2; lark++) {

                if (first[i] == calc_first[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if(chk == 0)
            {
                printf("%c, ", first[i]);
                calc_first[point1][point2++] = first[i];
            }
        }
        printf("}\n");
        jm = n;
        point1++;
    }
    printf("\n");
    printf("-----------------------------------------------\n\n");
    char donee[count];
    ptr = -1;

    // Initializing the calc_follow array
    for(k = 0; k < count; k++) {
            for(kay = 0; kay < 100; kay++) {
                calc_follow[k][kay] = '!';
            }
        }
        point1 = 0;
        int land = 0;
        for(e = 0; e < count; e++)
        {
            ck = production[e][0];
            point2 = 0;
            x = 0;

            // Checking if Follow of ck
            // has alredy been calculated
            for(kay = 0; kay <= ptr; kay++)
                if(ck == donee[kay])
                    x = 1;

            if (x == 1)
                continue;
            land += 1;

            // Function call
            follow(ck);
            ptr += 1;

            // Adding ck to the calculated list
            donee[ptr] = ck;
            printf(" Follow(%c) = { ", ck);
            calc_follow[point1][point2++] = ck;

            // Printing the Follow Sets of the grammar
            for(i = 0 + km; i < m; i++) {
                int lark = 0, chk = 0;
                for(lark = 0; lark < point2; lark++)
                {
                    if (f[i] == calc_follow[point1][lark])
                    {
                        chk = 1;
                        break;
                    }
                }
                if(chk == 0)
                {
                    printf("%c, ", f[i]);
                    calc_follow[point1][point2++] = f[i];
                }
            }
            printf(" }\n\n");
            km = m;
            point1++;
        }
}

void follow(char c)
{
        int i, j;
        // Adding "$" to the follow
        // set of the start symbol
        if(production[0][0] == c) {
                f[m++] = '$';
        }
        for(i = 0; i < 10; i++)
        {
                for(j = 2;j < 10; j++)
                {
                        if(production[i][j] == c)
                        {
                                if(production[i][j+1] != '\0')
                                {
                                        // Calculate the first of the next
                                        // Non-Terminal in the production
                                        followfirst(production[i][j+1], i, (j+2));
                                }

                                if(production[i][j+1]=='\0' && c!=production[i][0])
                                {
                                        // Calculate the follow of the Non-Terminal
                                        // in the L.H.S. of the production
                                        follow(production[i][0]);
                                }
                        }
                }
        }
}

void findfirst(char c, int q1, int q2)
{
        int j;

        // The case where we
        // encounter a Terminal
        if(!(isupper(c))) {
                first[n++] = c;
        }
        for(j = 0; j < count; j++)
        {
                if(production[j][0] == c)
                {
                        if(production[j][2] == '#')
                        {
                                if(production[q1][q2] == '\0')
                                        first[n++] = '#';
                                else if(production[q1][q2] != '\0'
                                                && (q1 != 0 || q2 != 0))
                                {
                                        // Recursion to calculate First of New
                                        // Non-Terminal we encounter after epsilon
                                        findfirst(production[q1][q2], q1, (q2+1));
                                }
                                else
                                        first[n++] = '#';
                        }
                        else if(!isupper(production[j][2]))
                        {
```

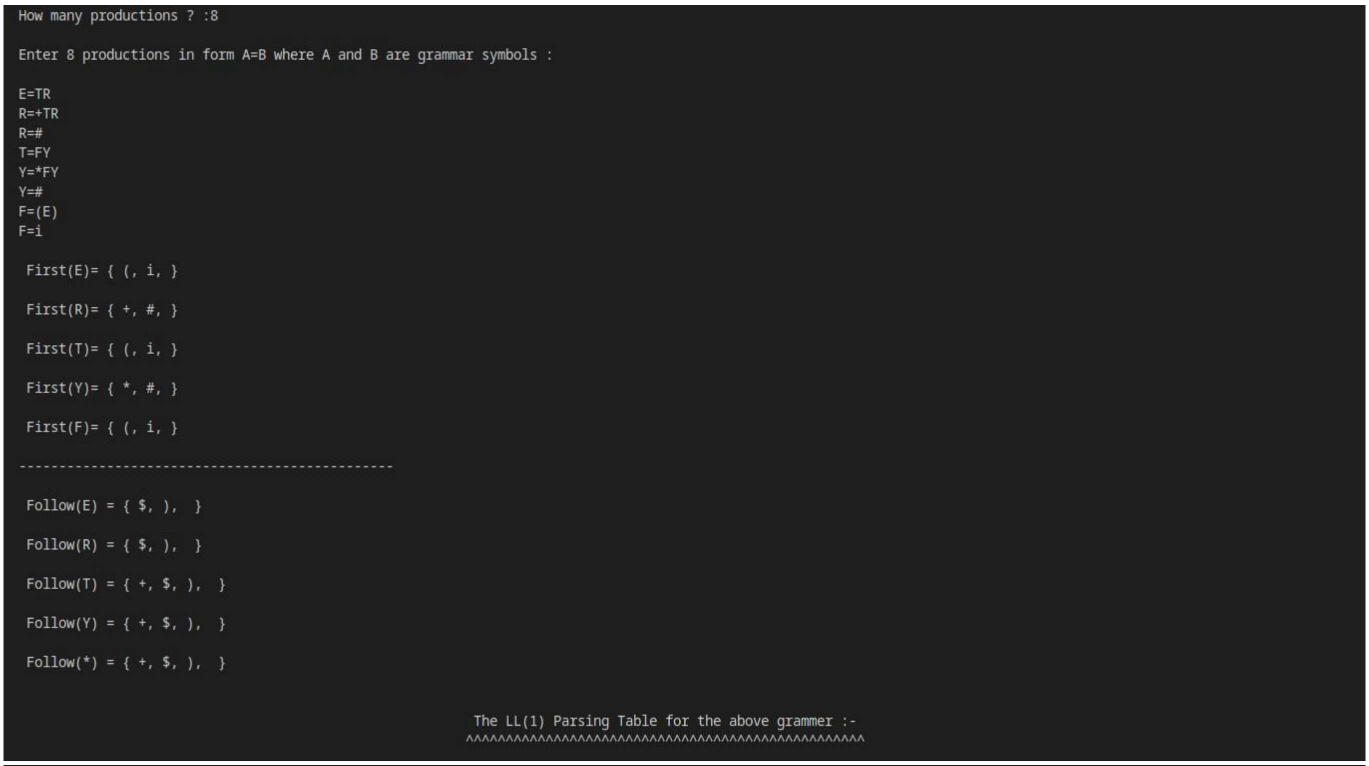
```
                        first[n++] = production[j][2];
                    }
                    else
                    {
                        // Recursion to calculate First of
                        // New Non-Terminal we encounter
                        // at the beginning
                        findfirst(production[j][2], j, 3);
                    }
                }
        }
}

void followfirst(char c, int c1, int c2)
{
        int k;

        // The case where we encounter
        // a Terminal
        if(!(isupper(c)))
                f[m++] = c;
        else
        {
                int i = 0, j = 1;
                for(i = 0; i < count; i++)
                {
                        if(calc_first[i][0] == c)
                                break;
                }

                //Including the First set of the
                // Non-Terminal in the Follow of
                // the original query
                while(calc_first[i][j] != '!')
                {
                        if(calc_first[i][j] != '#')
                        {
                                f[m++] = calc_first[i][j];
                        }
                        else
                        {
                                if(production[c1][c2] == '\0')
                                {
                                        // Case where we reach the
                                        // end of a production
                                        follow(production[c1][0]);
                                }
                                else
                                {
                                        // Recursion to the next symbol
                                        // in case we encounter a "#"
                                        followfirst(production[c1][c2], c1, c2+1);
                                }
                        }
                        j++;
                }
        }
}
```



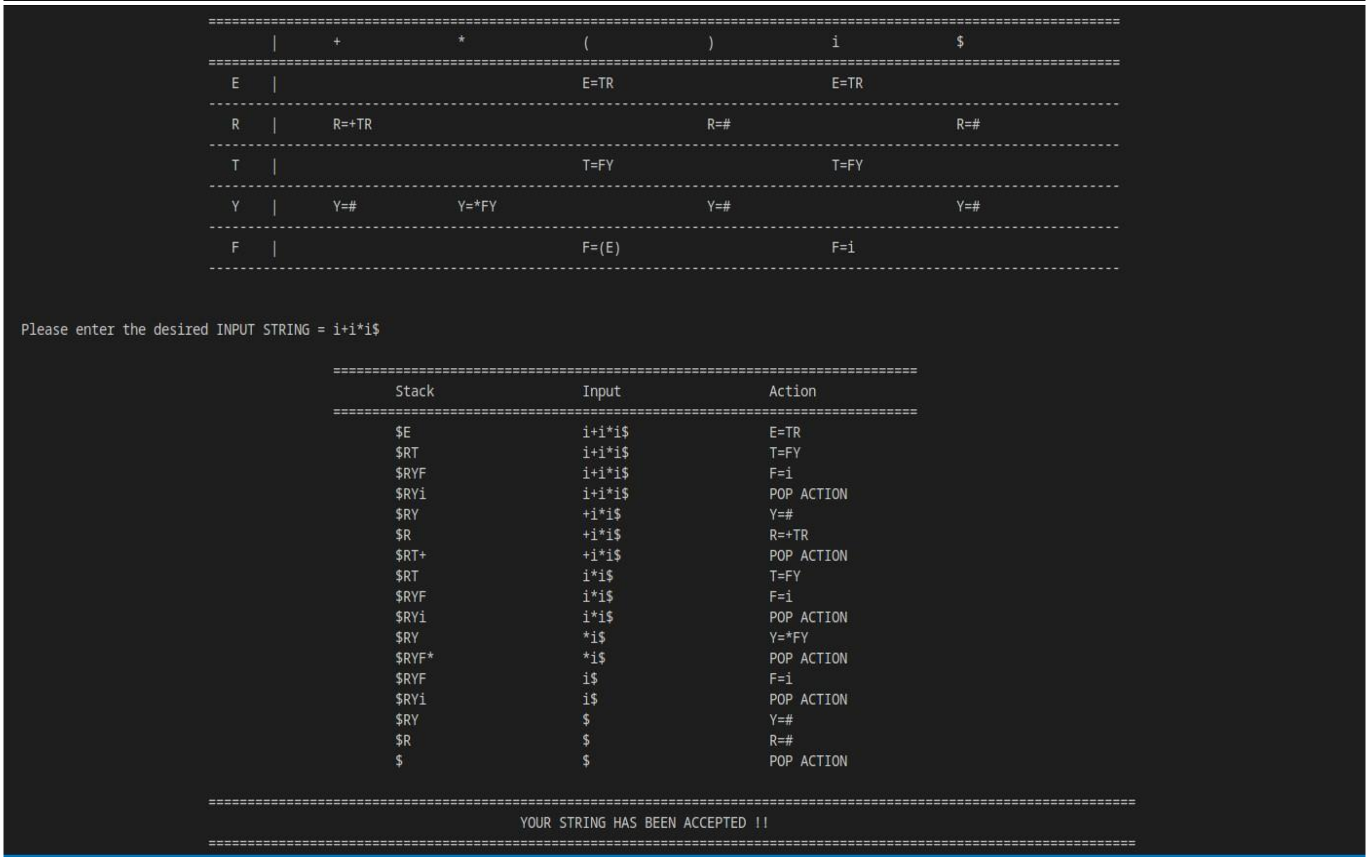
```
PS D:\College\7th SEM\CD (Compiler Design)\Practical> gcc -g  Prac_5_FIRST_FOLLOW.c
PS D:\College\7th SEM\CD (Compiler Design)\Practical> ./Prac_5_FIRST_FOLLOW

 First(E) = { (, i, }

 First(R) = { +, #, }

 First(T) = { (, i, }

 First(Y) = { *, #, }

 First(F) = { (, i, }

-----------------------------------------------

 Follow(E) = { $, ),  }

 Follow(R) = { $, ),  }

 Follow(T) = { +, $, ),  }

 Follow(Y) = { +, $, ),  }

 Follow(F) = { *, +, $, ),  }

PS D:\College\7th SEM\CD (Compiler Design)\Practical>
```

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

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

void followfirst(char, int, int);
void findfirst(char, int, int);
void follow(char c);

int count, n = 0;
char calc_first[10][100];
char calc_follow[10][100];
int m = 0;
char production[10][10], first[10];
char f[10];
int k;
char ck;
int e;

int main(int argc, char **argv)
{
    int jm = 0;
    int km = 0;
    int i, choice;
    char c, ch;
    printf("How many productions ? :");
    scanf("%d", &count);
    printf("\nEnter %d productions in form A=B where A and B are grammar symbols :\n\n",
count);
    for (i = 0; i < count; i++)
    {
        scanf("%s%c", production[i], &ch);
    }
    int kay;
    char done[count];
    int ptr = -1;
    for (k = 0; k < count; k++)
    {
        for (kay = 0; kay < 100; kay++)
        {
            calc_first[k][kay] = '!';
        }
    }
    int point1 = 0, point2, xxx;
    for (k = 0; k < count; k++)
    {
        c = production[k][0];
        point2 = 0;
        xxx = 0;
        for (kay = 0; kay <= ptr; kay++)
            if (c == done[kay])
                xxx = 1;
        if (xxx == 1)
            continue;
        findfirst(c, 0, 0);
        ptr += 1;
        done[ptr] = c;
        printf("\n First(%c)= { ", c);
        calc_first[point1][point2++] = c;
        for (i = 0 + jm; i < n; i++)
        {
            int lark = 0, chk = 0;
            for (lark = 0; lark < point2; lark++)
            {
                if (first[i] == calc_first[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0)
            {
                printf("%c, ", first[i]);
                calc_first[point1][point2++] = first[i];
            }
        }
        printf("}\n");
        jm = n;
        point1++;
    }
    printf("\n");
    printf("-----------------------------------------------\n\n");
    char donee[count];
    ptr = -1;
    for (k = 0; k < count; k++)
    {
        for (kay = 0; kay < 100; kay++)
        {
            calc_follow[k][kay] = '!';
        }
    }
    point1 = 0;
    int land = 0;
    for (e = 0; e < count; e++)
    {
        ck = production[e][0];
        point2 = 0;
        xxx = 0;
        for (kay = 0; kay <= ptr; kay++)
            if (ck == donee[kay])
                xxx = 1;
        if (xxx == 1)
            continue;
        land += 1;
        follow(ck);
        ptr += 1;
        donee[ptr] = ck;
        printf(" Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;
        for (i = 0 + km; i < m; i++)
        {
            int lark = 0, chk = 0;
            for (lark = 0; lark < point2; lark++)
            {
                if (f[i] == calc_follow[point1][lark])
                {
                    chk = 1;
                    break;
                }
            }
            if (chk == 0)
            {
                printf("%c, ", f[i]);
                calc_follow[point1][point2++] = f[i];
            }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }
    char ter[10];
    for (k = 0; k < 10; k++)
    {
        ter[k] = '!';
    }
    int ap, vp, sid = 0;
    for (k = 0; k < count; k++)
    {
        for (kay = 0; kay < count; kay++)
        {
            if (!isupper(production[k][kay]) && production[k][kay] != '#' &&
production[k][kay] != '=' && production[k][kay] != '\0')
            {
                vp = 0;
                for (ap = 0; ap < sid; ap++)
                {
                    if (production[k][kay] == ter[ap])
                    {
                        vp = 1;
                        break;
                    }
                }
                if (vp == 0)
                {
                    ter[sid] = production[k][kay];
                    sid++;
                }
            }
        }
    }
    ter[sid] = '$';
    sid++;
    printf("\n\t\t\t\t\t\t\t The LL(1) Parsing Table for the above grammer :-");
    printf("\n\t\t\t\t\t\t\t^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n");

printf("\n\t\t\t===============================================================================
========================================\n");
    printf("\t\t\t\t|\t");
    for (ap = 0; ap < sid; ap++)
    {
        printf("%c\t\t", ter[ap]);
    }

printf("\n\t\t\t==================================================================================
=======================================\n");
    char first_prod[count][sid];
    for (ap = 0; ap < count; ap++)
    {
        int destiny = 0;
        k = 2;
        int ct = 0;
        char tem[100];
        while (production[ap][k] != '\0')
        {
            if (!isupper(production[ap][k]))
            {
                tem[ct++] = production[ap][k];
                tem[ct++] = '_';
                tem[ct++] = '\0';
                k++;
                break;
            }
            else
            {
                int zap = 0;
                int tuna = 0;
                for (zap = 0; zap < count; zap++)
                {
                    if (calc_first[zap][0] == production[ap][k])
                    {
                        for (tuna = 1; tuna < 100; tuna++)
                        {
                            if (calc_first[zap][tuna] != '!')
                            {
                                tem[ct++] = calc_first[zap][tuna];
                            }
                            else
                                break;
                        }
                        break;
                    }
                }
                tem[ct++] = '_';
            }
            k++;
        }
        int zap = 0, tuna;
        for (tuna = 0; tuna < ct; tuna++)
        {
            if (tem[tuna] == '#')
            {
                zap = 1;
            }
            else if (tem[tuna] == '_')
            {
                if (zap == 1)
                {
                    zap = 0;
                }
                else
                    break;
            }
            else
            {
                first_prod[ap][destiny++] = tem[tuna];
            }
        }
    }
    char table[land][sid + 1];
    ptr = -1;
    for (ap = 0; ap < land; ap++)
    {
        for (kay = 0; kay < (sid + 1); kay++)
        {
            table[ap][kay] = '!';
        }
    }
    for (ap = 0; ap < count; ap++)
    {
        ck = production[ap][0];
        xxx = 0;
        for (kay = 0; kay <= ptr; kay++)
            if (ck == table[kay][0])
                xxx = 1;
        if (xxx == 1)
            continue;
        else
        {
            ptr = ptr + 1;
            table[ptr][0] = ck;
        }
    }
    for (ap = 0; ap < count; ap++)
    {
        int tuna = 0;
        while (first_prod[ap][tuna] != '\0')
        {
            int to, ni = 0;
            for (to = 0; to < sid; to++)
            {
                if (first_prod[ap][tuna] == ter[to])
                {
                    ni = 1;
                }
            }
            if (ni == 1)
            {
                char xz = production[ap][0];
                int cz = 0;
                while (table[cz][0] != xz)
                {
                    cz = cz + 1;
                }
                int vz = 0;
                while (ter[vz] != first_prod[ap][tuna])
                {
                    vz = vz + 1;
                }
                table[cz][vz + 1] = (char)(ap + 65);
            }
            tuna++;
        }
    }
    for (k = 0; k < sid; k++)
    {
        for (kay = 0; kay < 100; kay++)
        {
            if (calc_first[k][kay] == '!')
            {
                break;
            }
            else if (calc_first[k][kay] == '#')
            {
                int fz = 1;
                while (calc_follow[k][fz] != '!')
                {
                    char xz = production[k][0];
                    int cz = 0;
                    while (table[cz][0] != xz)
                    {
                        cz = cz + 1;
                    }
                    int vz = 0;
                    while (ter[vz] != calc_follow[k][fz])
                    {
                        vz = vz + 1;
                    }
                    table[k][vz + 1] = '#';
                    fz++;
                }
                break;
            }
        }
    }
    for (ap = 0; ap < land; ap++)
    {
        printf("\t\t\t   %c\t|\t", table[ap][0]);
        for (kay = 1; kay < (sid + 1); kay++)
        {
            if (table[ap][kay] == '!')
                printf("\t\t");
            else if (table[ap][kay] == '#')
                printf("%c=#\t\t", table[ap][0]);
            else
            {
                int mum = (int)(table[ap][kay]);
                mum -= 65;
                printf("%s\t\t", production[mum]);
            }
        }
        printf("\n");
        printf("\t\t\t-----------------------------------------------------------------
----------------------------------------------");
        printf("\n");
    }
    int j;
    printf("\n\nPlease enter the desired INPUT STRING = ");
    char input[100];
    scanf("%s%c", input, &ch);

printf("\n\t\t\t\t\t===============================================================
==\n");
    printf("\t\t\t\t\t\tStack\t\t\tInput\t\t\tAction");

printf("\n\t\t\t\t\t===============================================================
==\n");
    int i_ptr = 0, s_ptr = 1;
    char stack[100];
    stack[0] = '$';
    stack[1] = table[0][0];
    while (s_ptr != -1)
    {
        printf("\t\t\t\t\t\t");
        int vamp = 0;
        for (vamp = 0; vamp <= s_ptr; vamp++)
        {
            printf("%c", stack[vamp]);
        }
        printf("\t\t\t");
        vamp = i_ptr;
        while (input[vamp] != '\0')
        {
            printf("%c", input[vamp]);
            vamp++;
        }
        printf("\t\t\t");
        char her = input[i_ptr];
        char him = stack[s_ptr];
        s_ptr--;
        if (!isupper(him))
        {
            if (her == him)
            {
                i_ptr++;
                printf("POP ACTION\n");
            }
            else
            {
                printf("\nString Not Accepted by LL(1) Parser !!\n");
                exit(0);
            }
        }
        else
        {
            for (i = 0; i < sid; i++)
            {
                if (ter[i] == her)
                    break;
            }
            char produ[100];
            for (j = 0; j < land; j++)
            {
                if (him == table[j][0])
                {
                    if (table[j][i + 1] == '#')
                    {
                        printf("%c=#\n", table[j][0]);
                        produ[0] = '#';
                        produ[1] = '\0';
                    }
                    else if (table[j][i + 1] != '!')
                    {
                        int mum = (int)(table[j][i + 1]);
                        mum -= 65;
                        strcpy(produ, production[mum]);
                        printf("%s\n", produ);
                    }
                    else
                    {
                        printf("\nString Not Accepted by LL(1) Parser !!\n");
                        exit(0);
                    }
                }
            }
            int le = strlen(produ);
            le = le - 1;
            if (le == 0)
            {
                continue;
            }
            for (j = le; j >= 2; j--)
            {
                s_ptr++;
                stack[s_ptr] = produ[j];
            }
        }
    }

printf("\n\t\t\t=============================================================================
==========================================\n");
    if (input[i_ptr] == '\0')
    {
        printf("\t\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN ACCEPTED !!\n");
    }
    else
        printf("\n\t\t\t\t\t\t\t\t\tYOUR STRING HAS BEEN REJECTED !!\n");

printf("\t\t\t===============================================================================
========================================\n");
}

void follow(char c)
{
    int i, j;
    if (production[0][0] == c)
    {
        f[m++] = '$';
    }
    for (i = 0; i < 10; i++)
    {
        for (j = 2; j < 10; j++)
        {
            if (production[i][j] == c)
            {
                if (production[i][j + 1] != '\0')
                {
                    followfirst(production[i][j + 1], i, (j + 2));
                }
                if (production[i][j + 1] == '\0' && c != production[i][0])
                {
                    follow(production[i][0]);
                }
            }
        }
    }
}

void findfirst(char c, int q1, int q2)
{
    int j;
    if (!(isupper(c)))
    {
        first[n++] = c;
    }
    for (j = 0; j < count; j++)
    {
        if (production[j][0] == c)
        {
            if (production[j][2] == '#')
            {
                if (production[q1][q2] == '\0')
                    first[n++] = '#';
                else if (production[q1][q2] != '\0' && (q1 != 0 || q2 != 0))
                {
                    findfirst(production[q1][q2], q1, (q2 + 1));
                }
                else
                    first[n++] = '#';
            }
            else if (!isupper(production[j][2]))
            {
                first[n++] = production[j][2];
            }
            else
            {
                findfirst(production[j][2], j, 3);
            }
        }
    }
}

void followfirst(char c, int c1, int c2)
{
    int k;
    if (!(isupper(c)))
        f[m++] = c;
    else
    {
        int i = 0, j = 1;
        for (i = 0; i < count; i++)
        {
            if (calc_first[i][0] == c)
                break;
        }
        while (calc_first[i][j] != '!')
        {
            if (calc_first[i][j] != '#')
            {
                f[m++] = calc_first[i][j];
            }
            else
            {
                if (production[c1][c2] == '\0')
                {
                    follow(production[c1][0]);
                }
                else
                {
                    followfirst(production[c1][c2], c1, c2 + 1);
                }
            }
            j++;
        }
    }
}
```



```
How many productions ? :8

Enter 8 productions in form A=B where A and B are grammar symbols :

E=TR
R=+TR
R=#
T=FY
Y=*FY
Y=#
F=(E)
F=i

 First(E)= { (, i, }

 First(R)= { +, #, }

 First(T)= { (, i, }

 First(Y)= { *, #, }

 First(F)= { (, i, }

-----------------------------------------------

 Follow(E) = { $, ),  }

 Follow(R) = { $, ),  }

 Follow(T) = { +, $, ),  }

 Follow(Y) = { +, $, ),  }

 Follow(*) = { +, $, ),  }


                                                          The LL(1) Parsing Table for the above grammer :-
                                                         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

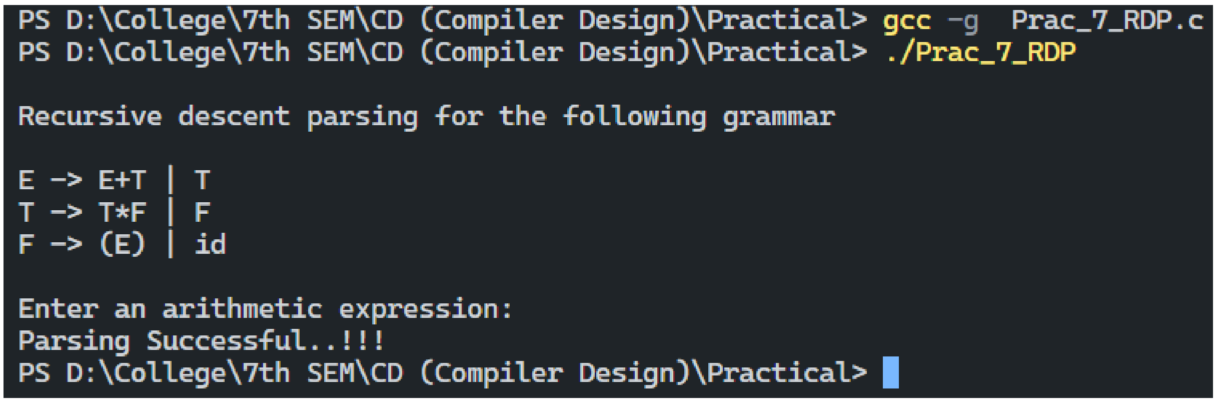


```
                       ==================================================================================================================
                               |       +              *              (               )              i              $
                       ==================================================================================================================
                          E    |                                      E=TR                           E=TR
                       ------------------------------------------------------------------------------------------------------------------
                          R    |       R=+TR                                          R=#                            R=#
                       ------------------------------------------------------------------------------------------------------------------
                          T    |                                      T=FY                           T=FY
                       ------------------------------------------------------------------------------------------------------------------
                          Y    |       Y=#             Y=*FY                          Y=#                            Y=#
                       ------------------------------------------------------------------------------------------------------------------
                          F    |                                      F=(E)                          F=i
                       ------------------------------------------------------------------------------------------------------------------


Please enter the desired INPUT STRING = i+i*i$

                                       ==========================================================================
                                              Stack                   Input                  Action
                                       ==========================================================================
                                              $E                      i+i*i$                 E=TR
                                              $RT                     i+i*i$                 T=FY
                                              $RYF                    i+i*i$                 F=i
                                              $RYi                    i+i*i$                 POP ACTION
                                              $RY                     +i*i$                  Y=#
                                              $R                      +i*i$                  R=+TR
                                              $RT+                    +i*i$                  POP ACTION
                                              $RT                     i*i$                   T=FY
                                              $RYF                    i*i$                   F=i
                                              $RYi                    i*i$                   POP ACTION
                                              $RY                     *i$                    Y=*FY
                                              $RYF*                   *i$                    POP ACTION
                                              $RYF                    i$                     F=i
                                              $RYi                    i$                     POP ACTION
                                              $RY                     $                      Y=#
                                              $R                      $                      R=#
                                              $                       $                      POP ACTION

                       ====================================================================================================================
                                                              YOUR STRING HAS BEEN ACCEPTED !!
                       ====================================================================================================================
```

7. Implementation of Recursive Descent Parser without backtracking
Input: The string to be parsed.
Output: Whether string parsed successfully or not.
(NOTE: Students have to implement the recursive procedure for RDP for a typical grammar. The production numbers are displayed as they are used to derive the string.)

```
#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();

int 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);     scanf("%s",
input);     E();
    if (strlen(input) == i && error == 0)
        printf("\nParsing Successful..!!!\n");
    else
        printf("\nError..!!!\n");
    return 0;
}

void E()
{
    T();
    Eprime();
}

void Eprime()
{
    if (input[i] == '+')
    {
        i++;
        T();
        Eprime();
    }
}

void T()
{
    F();
    Tprime();
}

void Tprime()
```

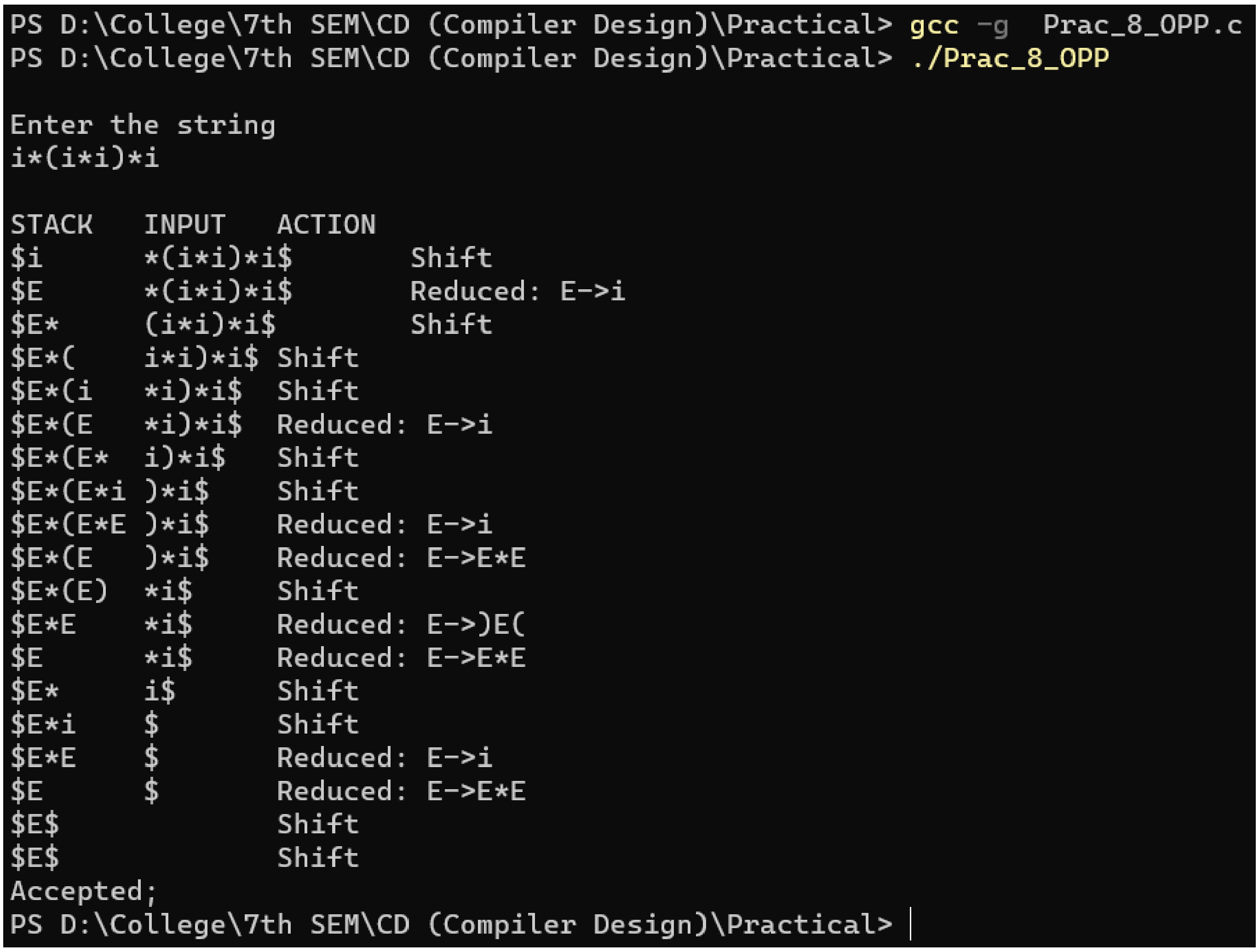
```
{
    if (input[i] == '*')
    {
        i++;
        F();
        Tprime();
    }
}

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



```
PS D:\College\7th SEM\CD (Compiler Design)\Practical> gcc -g  Prac_7_RDP.c
PS D:\College\7th SEM\CD (Compiler Design)\Practical> ./Prac_7_RDP

Recursive descent parsing for the following grammar

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

Enter an arithmetic expression:
Parsing Successful..!!!
PS D:\College\7th SEM\CD (Compiler Design)\Practical>
```

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

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

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

int shift()
{
    stack[++top] = *(input + i++);
    stack[top + 1] = '\0';
}

int reduce()
{
    int i, len, found, t;
    for (i = 0; i < 5; i++)
    {
        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));
}

int main()
{
    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, "$E$") == 0)
        printf("\nAccepted;");
    else
        printf("\nNot Accepted;");

    return 0;
}
```



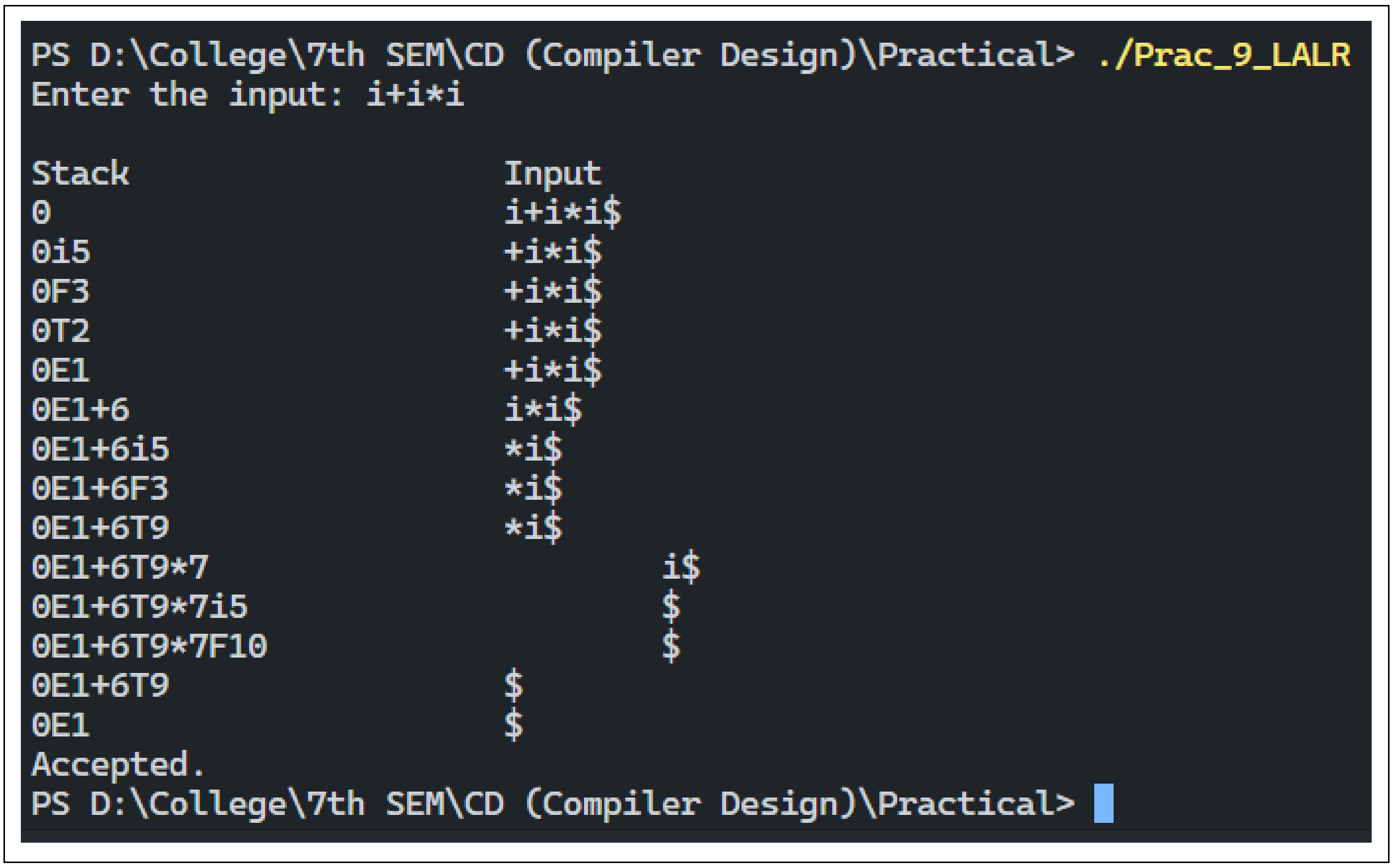
```
PS D:\College\7th SEM\CD (Compiler Design)\Practical> gcc -g  Prac_8_OPP.c
PS D:\College\7th SEM\CD (Compiler Design)\Practical> ./Prac_8_OPP

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;
PS D:\College\7th SEM\CD (Compiler Design)\Practical>
```

9. Implement a C program to implement LALR parsing.

```
#include <stdio.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, len;
    printf("Enter the input: ");
    scanf("%s", inp);
    len = strlen(inp);
    inp[len] = '$';
    inp[len + 1] = '\0';
    push(stack, &top, bl);
    printf("\nStack\t\t\tInput");
    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("\nAccepted.");
    else
        printf("\nNot Accepted.");

    getchar();
}

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)
{
```

```c
    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;
    }
}
```


```
PS D:\College\7th SEM\CD (Compiler Design)\Practical> ./Prac_9_LALR
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                     $
Accepted.
PS D:\College\7th SEM\CD (Compiler Design)\Practical>
```

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

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 program and the output is a C program.

Input File:

```
/*definitions*/
%%
/*rules*/
…..
%%
/*auxiliaryroutines*/
….
```

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 definitions 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 name dy.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 headerfiley.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.y
2. Open Terminal and Navigate to the Directory where you have saved the files.
3. Typelexfile.l
4. Typeyaccfile.y
5. Typecclex.yy.cy.tab.h-ll
6. type./a.out

Example:

YaccFile(.y):

```
%{
    #include <ctype.h>
    #include <stdio.h>
    #defineYYSTYP Edouble
}%

%%
Lines:LinesS'\n'{printf("OK\n");}
|S'\n' |error'\n'{yyerror("Error:reenterlastline:");
yyerrok;};
S :'('S')'
|'['S']' |
%%

#include "lex.yy.c"
Void yyerror(c har*s){
    Fprintf(stderr,"%s\n",s);
}
intmain(void){
   Returnyyparse();
}
```

LexFile(.l):

```
%{
%}
%%
[\t] {}
\n]. {returnyytext[0];}
%%
```

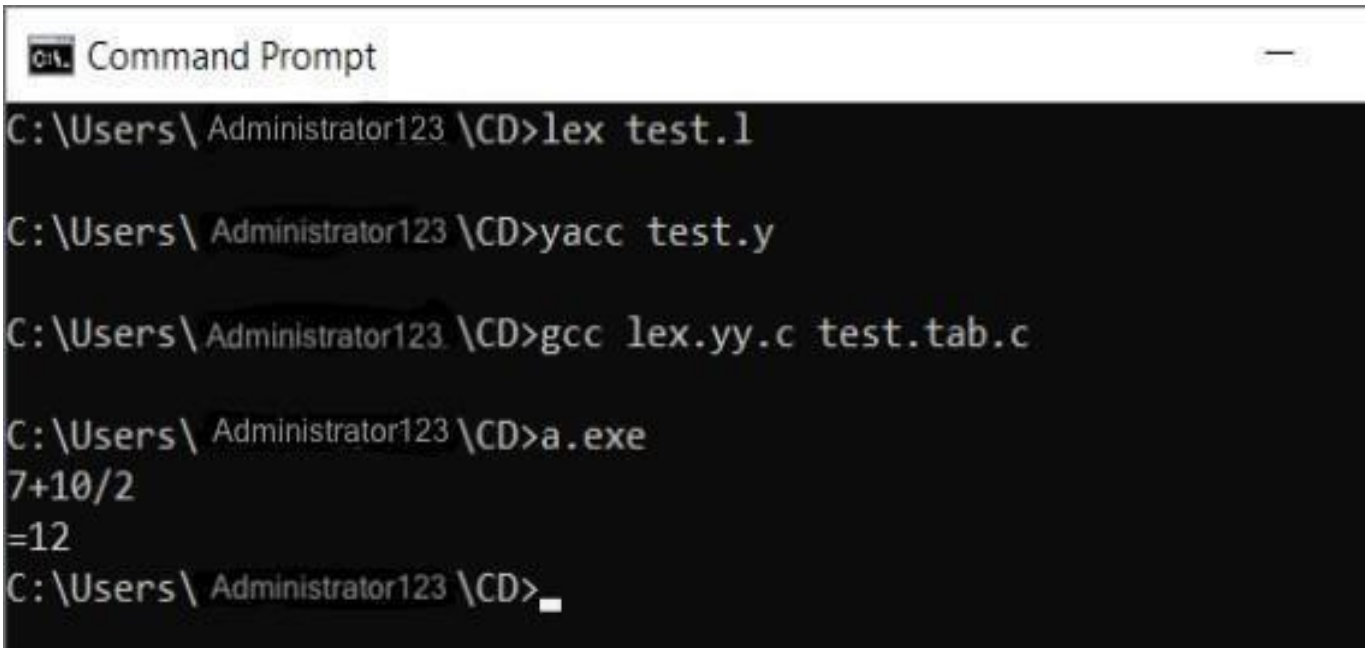
11. Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, * and /.

Text.l

```
%option noyywrap
%{
    #include "stdio.h"
    #include "test.tab.h"
    extern int yylval;
%}

%%
[0-9]+ {yylval=atoi(yytext);return number;} \+ { return plus;}
\- { return minus;}
\* { return multiply;}
\/ {return divide;}
. {return yytext[0];} [\t]+;
\n return 0;
%%
```

Text.y

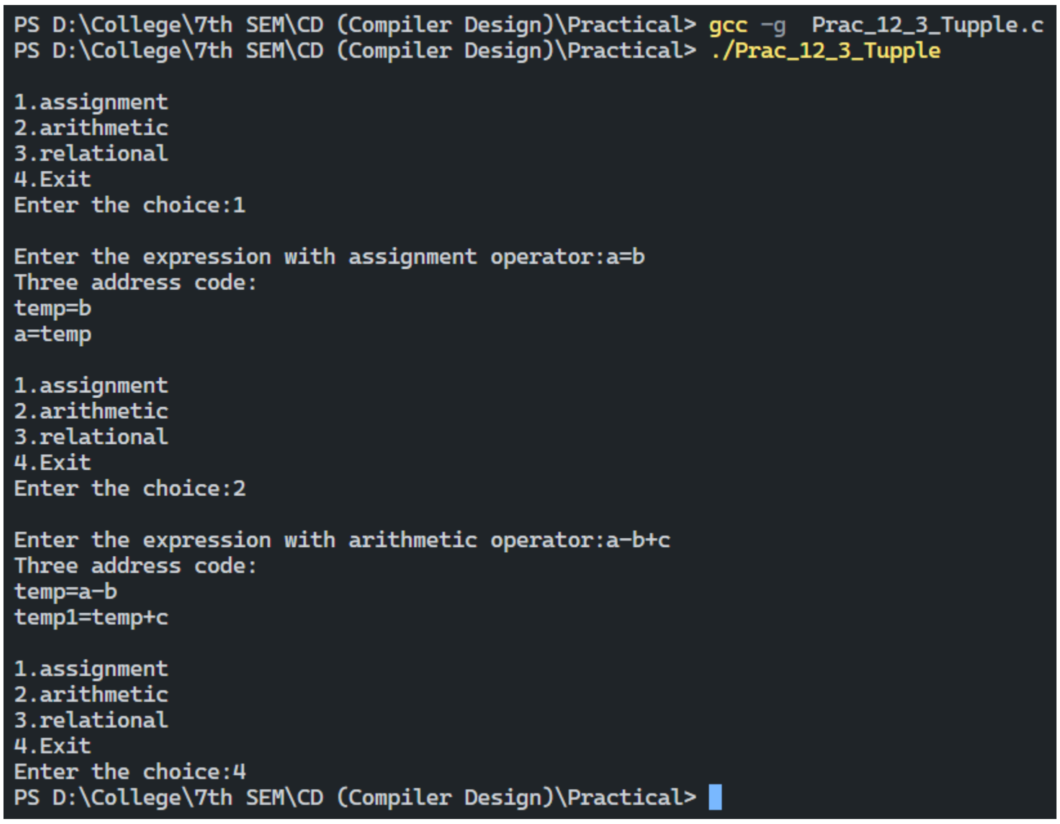
```
%{
    #include "stdio.h"
    int result=0;
    void yyerror(const char *str)
    {
        fprintf(stderr,"error: %s\n",str);
    }
    int yywrap(void) {
        return 1;
    }
%}

%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);
}
```



```
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>_
```

12. Generate 3-tuple intermediate code for given infix expression.

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

void pm();
void plus();
void divide();
void reverse(char *str);

int i, ch, j, l, addr = 100;
char ex[10], expr[10], expr1[10], expr2[10], id1[5], op[5], id2[5];

int main()
{
    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", expr);
            l = strlen(expr);
            expr2[0] = '\0';
            i = 0;
            while (expr[i] != '=')
            {
                i++;
            }
            strncat(expr2, expr, i);
            reverse(expr);
            expr1[0] = '\0';
            strncat(expr1, expr, l - (i + 1));
            reverse(expr1);
            printf("Three address code:\ntemp=%s\n%s=temp\n", expr1, expr2);
            break;
        case 2:
            printf("\nEnter the expression with arithmetic operator:");
            scanf("%s", ex);
            strcpy(expr, ex);
            l = strlen(expr);
            expr1[0] = '\0';
            for (i = 0; i < l; i++)
            {
                if (expr[i] == '+' || expr[i] == '-')
                {
                    if (expr[i + 2] == '/' || expr[i + 2] == '*')
                    {
                        pm();
                        break;
                    }
                    else
                    {
                        plus();
                        break;
                    }
                }
                else if (expr[i] == '/' || expr[i] == '*')
                {
                    divide();
                    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\t T:=0", addr);
                addr++;
                printf("\n%d\t goto %d", addr, addr + 2);
                addr++;
                printf("\n%d\t T:=1", addr);
            }
            break;
        case 4:
            exit(0);
        }
    }
    return 0;
}

void pm()
{
    reverse(expr);
    j = l - i - 1;
    strncat(expr1, expr, j);
    reverse(expr1);
    printf("Three address code:\ntemp=%s\ntemp1=%c%ctemp\n", expr1, expr[j + 1], expr[j]);
}

void divide()
{
    strncat(expr1, expr, i + 2);
    printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n", expr1, expr[i + 2], expr[i + 3]);
}

void plus()
{
    strncat(expr1, expr, i + 2);
    printf("Three address code:\ntemp=%s\ntemp1=temp%c%c\n", expr1, expr[i + 2], expr[i + 3]);
}

void reverse(char *str)
{
    int length = strlen(str);
    char *begin = str;
    char *end = str + length - 1;
    char temp;
    while (begin < end)
    {
        temp = *begin;
        *begin = *end;
        *end = temp;
        begin++;
        end--;
    }
}
```

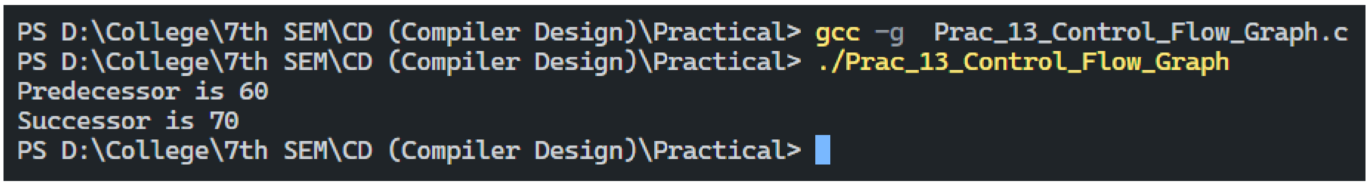


```
PS D:\College\7th SEM\CD (Compiler Design)\Practical> gcc -g  Prac_12_3_Tupple.c
PS D:\College\7th SEM\CD (Compiler Design)\Practical> ./Prac_12_3_Tupple

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:4
PS D:\College\7th SEM\CD (Compiler Design)\Practical>
```

13. Extract Predecessor and Successor from given Control Flow Graph.

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

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

// Utility function to create a new BST node
struct Node *newNode(int item)
{
    struct Node *temp = (struct Node *)malloc(sizeof(struct Node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

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

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

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

        // The minimum value in the right subtree is successor
        if (root->right != NULL)
        {
            struct 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);
    }
}

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

    // Let us create the BST
    struct 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);

    struct Node *pre = NULL, *suc = NULL;
    findPreSuc(root, &pre, &suc, key);

    if (pre != NULL)
        printf("Predecessor is %d\n", pre->key);
    else
        printf("No Predecessor\n");

    if (suc != NULL)
        printf("Successor is %d\n", suc->key);
    else
        printf("No Successor\n");

    return 0;
}
```



```
PS D:\College\7th SEM\CD (Compiler Design)\Practical> gcc -g  Prac_13_Control_Flow_Graph.c
PS D:\College\7th SEM\CD (Compiler Design)\Practical> ./Prac_13_Control_Flow_Graph
Predecessor is 60
Successor is 70
PS D:\College\7th SEM\CD (Compiler Design)\Practical>
```