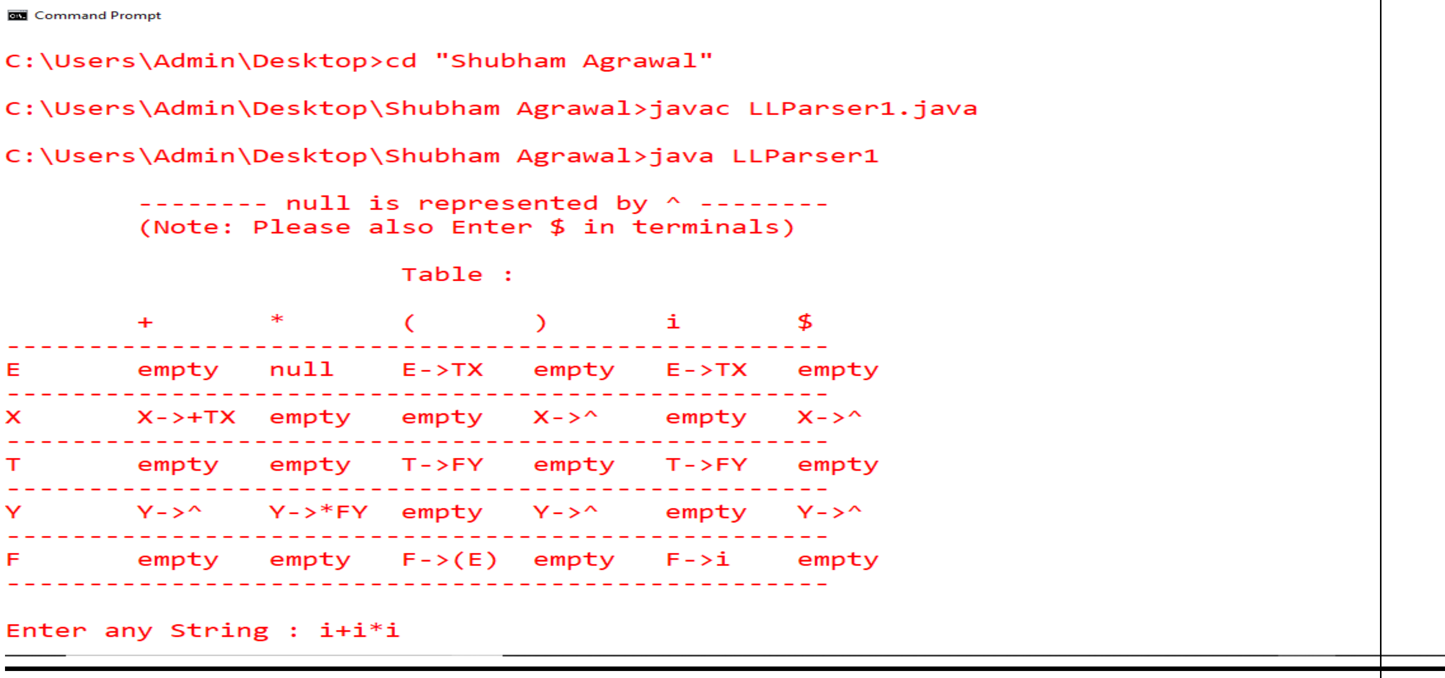
EXPERIMENT-5

Aim: To design LL parser for Context Free Grammar.

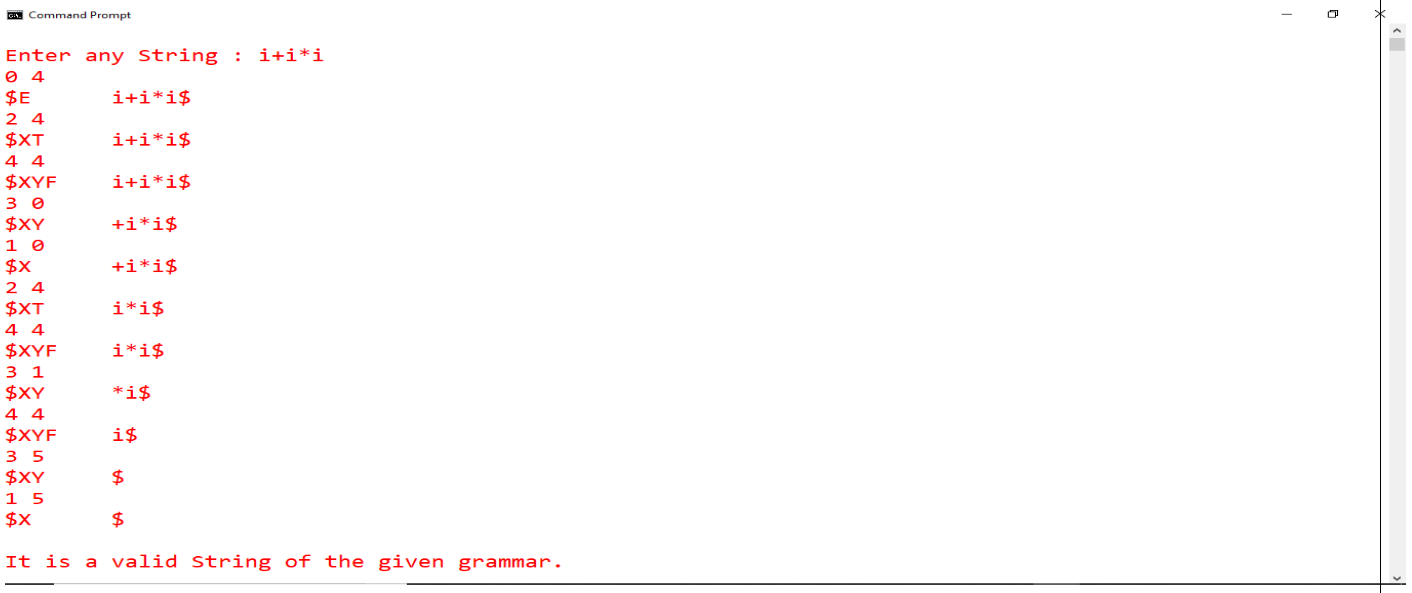
```java
import java.util.Scanner;
import java.util.Stack;
import java.util.Iterator;

class LLParser1
{
    static String[] nonTerminals;
    static String[] terminals;
    static String[][] table;
    static int n;
    static Stack<String> stack = new Stack<String>();

    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("\n\t-------- null is represented by ^ --------\n\t(Note: Please also Enter $ in terminals)");
        String ter = "+ * ( ) i $";
        terminals = ter.split(" ");
        String nonTer = "E X T Y F";
        nonTerminals = nonTer.split(" ");

        table = new String[nonTerminals.length][terminals.length];

        table[0][0] = "empty";
        table[0][0] = "empty";
        table[0][2] = "E->TX";
        table[0][3] = "empty";
        table[0][4] = "E->TX";
        table[0][5] = "empty";

        table[1][0] = "X->+TX";
        table[1][1] = "empty";
        table[1][2] = "empty";
        table[1][3] = "X->^";
        table[1][4] = "empty";
        table[1][5] = "X->^";

        table[2][0] = "empty";
        table[2][1] = "empty";
        table[2][2] = "T->FY";
        table[2][3] = "empty";
        table[2][4] = "T->FY";
        table[2][5] = "empty";

        table[3][0] = "Y->^";
        table[3][1] = "Y->*FY";
        table[3][2] = "empty";
        table[3][3] = "Y->^";
        table[3][4] = "empty";
        table[3][5] = "Y->^";

        table[4][0] = "empty";
        table[4][1] = "empty";
        table[4][2] = "F->(E)";
        table[4][3] = "empty";
        table[4][4] = "F->i";
        table[4][5] = "empty";


        System.out.println("\n\t\t\tTable : \n");
        System.out.print("\t");
        for(int j=0;j<terminals.length;j++)
        {
            System.out.print(terminals[j]+"\t");
        }
        System.out.println("\n--------------------------------------------------------");

        for(int i=0;i<nonTerminals.length;i++)
        {
            System.out.print(nonTerminals[i]+"\t");
            for(int j=0;j<terminals.length;j++)
            {
                System.out.print(table[i][j]+"\t");
            }
            System.out.println("\n------------------------------------------------------");
        }


        System.out.print("\nEnter any String : ");
        String str = sc.next();
        str=str+"$";

        stack.push("$");
        stack.push(nonTerminals[0]);

        if(checkString(str))
        {
            System.out.println("\nIt is a valid String of the given grammar.");
        }
        else
        {
            System.out.println("\nIt is not a valid String of the given 
grammar.");
        }
    }

    public static boolean checkString(String str)
    {
        for(int i=0;i<str.length();i++)
        {
                while(true)
                {
                    String tempp=stack.peek();
                    char t=str.charAt(i);
                    String tt=t+"";
                    int j = findIndexNT(tempp);
                    int k = findIndexT(tt);
                    System.out.println(j+" "+k);

                    Iterator<String> itr = stack.iterator();
                    while(itr.hasNext())
                    {
                        System.out.print(itr.next());
                    }
                    System.out.print("\t");
                    for(int b=i;b<str.length();b++)
                    {
                        System.out.print(str.charAt(b));
                    }
                    System.out.println();

                    if(stack.peek().equals(tt))
                    {
                        stack.pop();
                        break;
                    }


                    String temp = table[j][k];

                    if(temp.equals("empty"))
                    {
                        return false;
                    }

                    String[] temp1 = temp.split("->");

                    if(temp1[1].equals("^"))
                    {
                        stack.pop();
                    }
                    else
                    {
                        stack.pop();
                        for(int c=temp1[1].length()-1;c>=0;c--)
                        {
                            stack.push(temp1[1].charAt(c)+"");
                        }
                    }
                    if(stack.peek().equals(tt))
                    {
                        stack.pop();
                        break;
                    }
                }

            }
            if(stack.empty())
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        public static int findIndexNT(String s)
        {
            int i=0;
            for (;i<nonTerminals.length;i++)
            {
                if(s.equals(nonTerminals[i]))
                {
                    break;
                }
            }
            return i;
        }

        public static int findIndexT(String s)
        {
            int i=0;
            for (;i<terminals.length;i++)
            {
                if(s.equals(terminals[i]))
                {
                    break;
                }
            }
```

```
                return i;
            }
        }
```

Output:



```
C:\Users\Admin\Desktop>cd "Shubham Agrawal"

C:\Users\Admin\Desktop\Shubham Agrawal>javac LLParser1.java

C:\Users\Admin\Desktop\Shubham Agrawal>java LLParser1

          -------- null is represented by ^ --------
          (Note: Please also Enter $ in terminals)

                              Table :

          +         *         (         )         i         $
--------------------------------------------------------------
E         empty     null      E->TX     empty     E->TX     empty
--------------------------------------------------------------
X         X->+TX    empty     empty     X->^      empty     X->^
--------------------------------------------------------------
T         empty     empty     T->FY     empty     T->FY     empty
--------------------------------------------------------------
Y         Y->^      Y->*FY    empty     Y->^      empty     Y->^
--------------------------------------------------------------
F         empty     empty     F->(E)    empty     F->i      empty
--------------------------------------------------------------

Enter any String : i+i*i
```



```
Enter any String : i+i*i
0 4
$E      i+i*i$
2 4
$XT     i+i*i$
4 4
$XYF    i+i*i$
3 0
$XY     +i*i$
1 0
$X      +i*i$
2 4
$XT     i*i$
4 4
$XYF    i*i$
3 1
$XY     *i$
4 4
$XYF    i$
3 5
$XY     $
1 5
$X      $

It is a valid String of the given grammar.
```

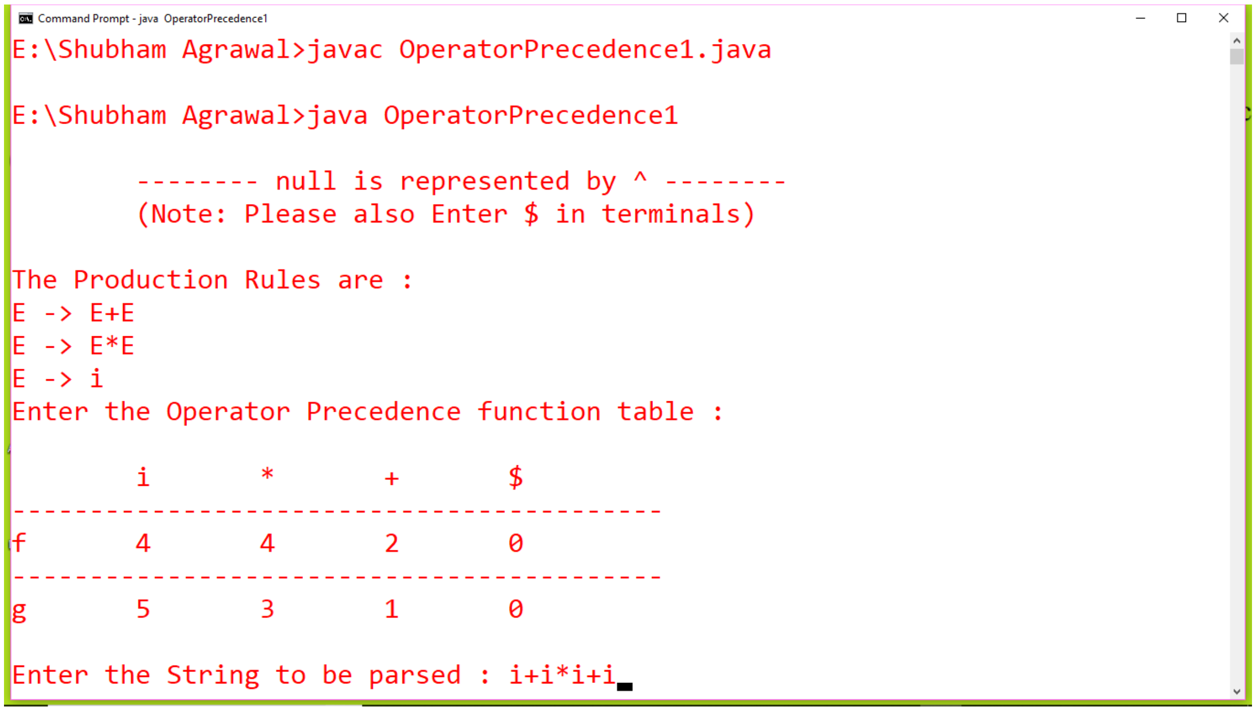
EXPERIMENT-6

Aim: Program to implement Operator Precedence Parser.

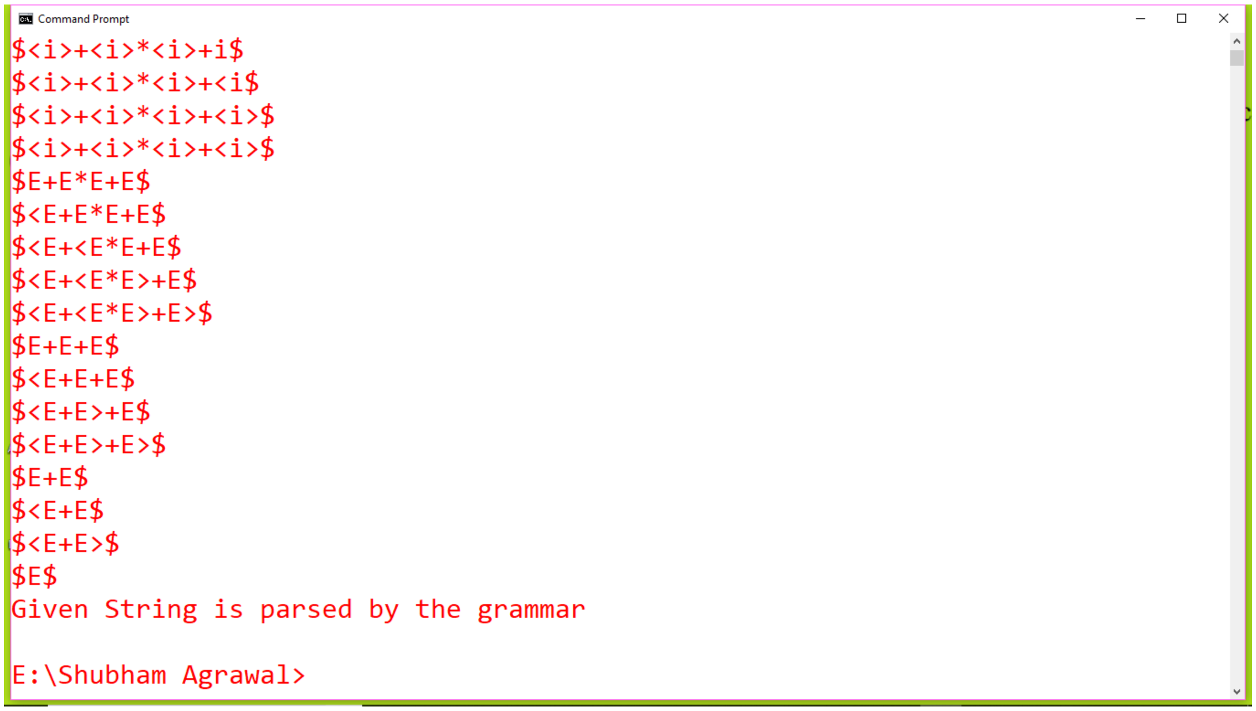
```
import java.util.Scanner;
import java.util.Stack;
import java.util.Map;
import java.util.HashMap;

class OperatorPrecedence1
{
    static String[] nonTerminals;
    static String[] terminals;
    static String[][] productionRules;
    static Map<String,Integer> f;
    static Map<String,Integer> g;
    static int n;
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("\n\t-------- null is represented by ^ --------
\n\t(Note: Please also Enter $ in terminals)");
        terminals = "i * + $".split(" ");
        nonTerminals = "E".split(" ");
        n = 3;
        productionRules = new String[n][2];


        String[] temp1 = "E->E+E".split("->");
        productionRules[0] =  temp1;

        temp1 = "E->E*E".split("->");
        productionRules[1] =  temp1;

        temp1 = "E->i".split("->");
        productionRules[2] =  temp1;

        System.out.println("\nThe Production Rules are : ");
        for(int i=0;i<n;i++)
        {
            System.out.println(productionRules[i][0]+" -
> "+productionRules[i][1]);
        }

        f = new HashMap<String,Integer>();
        g = new HashMap<String,Integer>();

     System.out.print("Enter the Operator Precedence function table : \n\n");
        f.put(terminals[0],4);
        f.put(terminals[1],4);
        f.put(terminals[2],2);
        f.put(terminals[3],0);

        g.put(terminals[0],5);
        g.put(terminals[1],3);
        g.put(terminals[2],1);
        g.put(terminals[3],0);
        for(int k=0;k<terminals.length;k++)
        {
            System.out.print("\t"+terminals[k]);
        }
        System.out.println();
        System.out.println("-----------------------------------------");
        System.out.print("f\t");
        for(int k=0;k<terminals.length;k++)
        {
            System.out.print(f.get(terminals[k])+"\t");
        }
        System.out.println();
        System.out.println("-----------------------------------------");
        System.out.print("g\t");
        for(int k=0;k<terminals.length;k++)
        {
            System.out.print(g.get(terminals[k])+"\t");
        }
        System.out.println("\n");

        System.out.print("Enter the String to be parsed : ");
        String str = sc.next();
        str = "$"+str+"$";
        str=check(str);
        if("$E$".equals(str))
        {
            System.out.println("Given String is parsed by the grammar");
        }
        else
        {
            System.out.println("Given String is not parsed by the grammar");
        }
    }

    public static String check(String str)
    {
        int l=0,r=0;

        for(r=1;r<str.length();)
        {

          if(str.charAt(r)!='+' || str.charAt(r)!='*' || str.charAt(r)!='i' ||
str.charAt(r)!='$')
            {
                if(f.get(""+str.charAt(l)) > g.get(""+str.charAt(r)))
                {
                    str = new StringBuffer(str).insert(l+1,">").toString();
```

```java
             System.out.println(str);
         }
                else if(f.get(""+str.charAt(l)) < g.get(""+str.charAt(r)))
                {
                    str = new StringBuffer(str).insert(r,"<").toString();
                    System.out.println(str);
                }
                else
                {
                    return str;
                }
            }
            else
            {
                continue;
            }
            l=r+1;
            r=r+2;

        }

        System.out.println(str);

        int lessThan = -1;
        int greaterThan = -1;

        lessThan = str.indexOf('<');
        greaterThan = str.indexOf('>');

        String sub = str.substring(lessThan+1,greaterThan);

        for(int i=0;i<n;i++)
        {
            if(productionRules[i][1].equals(sub))
            {
                str=str.replace("<"+sub+">",productionRules[i][0]);
            }
        }
        System.out.println(str);
        return check1(str);
    }

    public static String check1(String str)
    {
        Stack<Integer> stack = new Stack<Integer>();
        int l=0,r=0;

        for(r=1;r<str.length();)
        {

            if(str.charAt(r)=='+' || str.charAt(r)=='*' || str.charAt(r)=='$')
            {
                if(f.get(""+str.charAt(l)) > g.get(""+str.charAt(r)))
                {
                    str = new StringBuffer(str).insert(r,">").toString();
                    System.out.println(str);
                    l=++r;
                    r++;
                }
            else if(f.get(""+str.charAt(l)) < g.get(""+str.charAt(r)))
            {
               str = new StringBuffer(str).insert(l+1,"<").toString();
               System.out.println(str);
               l=++r;
               r++;
            }
            else
            {  return str;
            }
            }
            else
            {
                r++;
                continue;
            }
        }
        for(int i=0;i<str.length();i++)
        {
            if(str.charAt(i) == '<')
            {
                stack.push(i);
            }
            if(str.charAt(i) == '>')
            {
                stack.push(i);
                int greaterThan = stack.pop();
                int lessThan = stack.pop();

                String sub = str.substring(lessThan+1,greaterThan);
                boolean flag = false;
                for(int j=0;j<n;j++)
                {
                    if(productionRules[j][1].equals(sub))
                    {
                        flag=true;
                  str=str.replace("<"+sub+">",productionRules[j][0]);
                    }
                }
                if(!flag)
                    return null;
                str=str.replace("<","");
                str=str.replace(">","");

                System.out.println(str);
                break;
            }
            }
            return check1(str);
        }
    }
```

Output:



```
E:\Shubham Agrawal>javac OperatorPrecedence1.java

E:\Shubham Agrawal>java OperatorPrecedence1

        -------- null is represented by ^ --------
        (Note: Please also Enter $ in terminals)

The Production Rules are :
E -> E+E
E -> E*E
E -> i
Enter the Operator Precedence function table :

        i       *       +       $
------------------------------------------
f       4       4       2       0
------------------------------------------
g       5       3       1       0

Enter the String to be parsed : i+i*i+i
```



```
$<i>+<i>*<i>+i$
$<i>+<i>*<i>+<i$
$<i>+<i>*<i>+<i>$
$<i>+<i>*<i>+<i>$
$E+E*E+E$
$<E+E*E+E$
$<E+<E*E+E$
$<E+<E*E>+E$
$<E+<E*E>+E>$
$E+E+E$
$<E+E+E$
$<E+E>+E$
$<E+E>+E>$
$E+E$
$<E+E$
$<E+E>$
$E$
Given String is parsed by the grammar

E:\Shubham Agrawal>
```

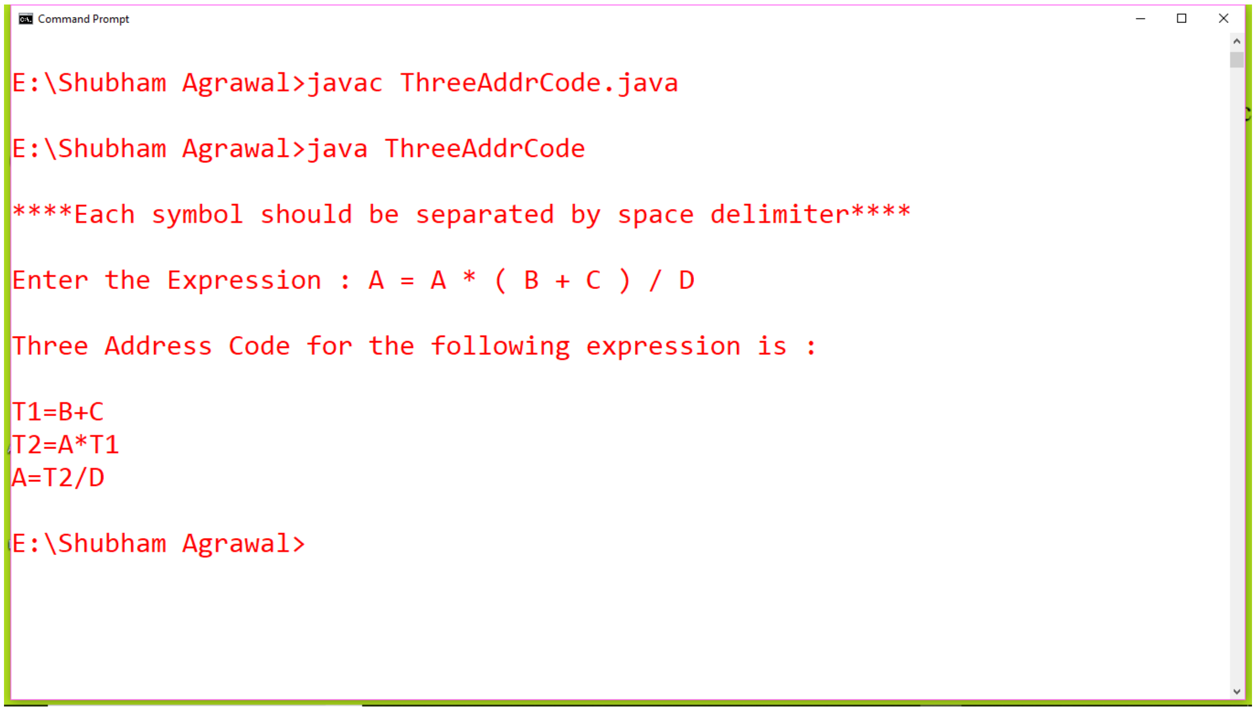
EXPERIMENT-7

Aim: Program to generate Valid 3 Address Code for a given Expression.

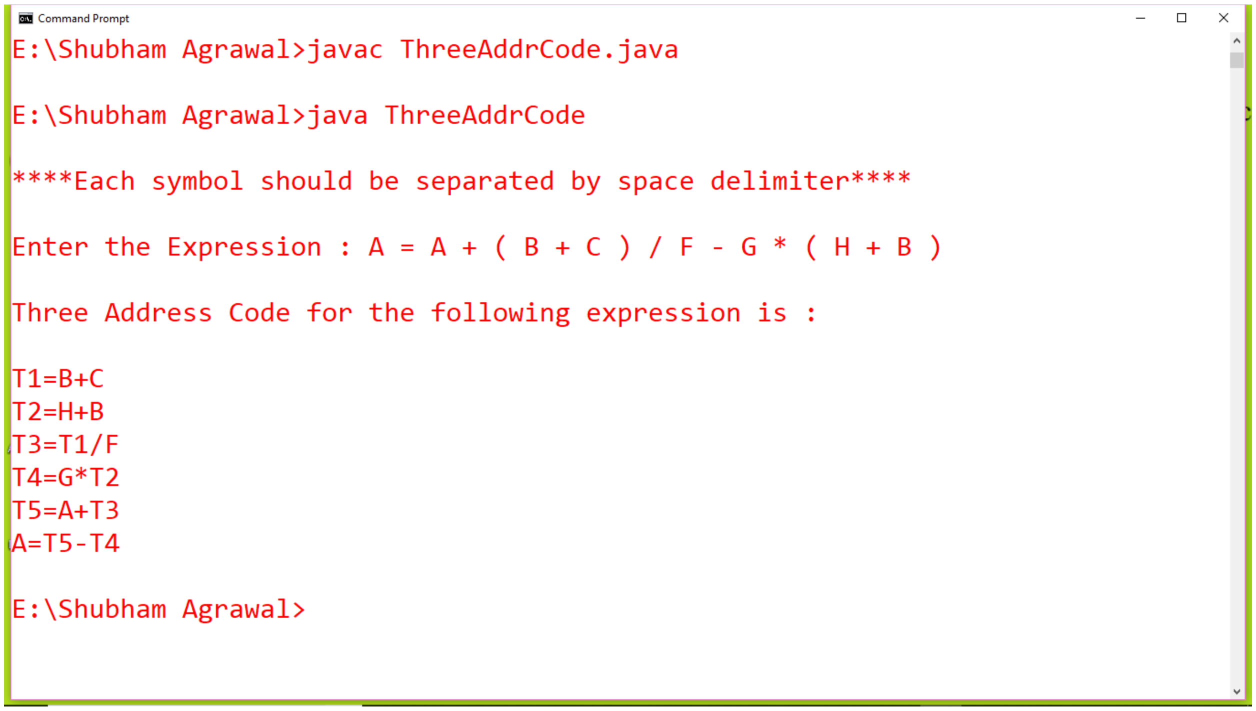
```java
import java.util.Scanner;
import java.util.Map;
import java.util.Set;
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedHashMap;

class ThreeAddrCode
{
    static int count = 0;
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("\n****Each symbol should be separated by space delimiter****\n");
        System.out.print("Enter the Expression : ");
        String expr1 = sc.nextLine();
        String[] expr2 = expr1.split(" ");
        List<String> expr = new ArrayList<String>();
        for(int i=0;i<expr2.length;i++)
        {
            expr.add(expr2[i]);
        }
        System.out.println("\nThree Address Code for the following expression is : \n");

        convertToThreeAddrCode(expr);

    }
    public static void convertToThreeAddrCode(List<String> expr)
    {
        Map<String,String> map = new LinkedHashMap<String,String>();

        int pLeft = -1;
        int pRight = -1;
        int plus = -1;
        int minus = -1;
        int multiply = -1;
        int divide = -1;
        int assignmentOp = -1;


        while(expr.size() > 5)
        {
                pLeft = expr.indexOf("(");
                pRight = expr.indexOf(")");
                plus = expr.indexOf("+");
                minus = expr.indexOf("-");
                multiply = expr.indexOf("*");
                divide = expr.indexOf("/");
                int op=-1;
                if(pLeft != -1 && pRight != -1 && pLeft < pRight)
                {
                    List<String> l = sublist(expr,pLeft,pRight);
                    convertToThreeAddrCode(l);
                }
                else if(divide!=-1 || multiply!=-1)
                {
                    if(divide!=-1 && multiply!=-1)
                    {
                        op = divide<multiply?divide:multiply;
                    }
                    else if(divide!=-1)
                    {
                        op = divide;
                    }
                    else
                    {
                        op = multiply;
                    }
                }
                else if(plus!=-1 || minus!=-1)
                {
                    if(plus!=-1 && minus!=-1)
                    {
                        op = plus<minus?plus:minus;
                    }
                    else if(plus!=-1)
                    {
                        op = plus;
                    }
                    else
                    {
                        op = minus;
                    }
                }
                if(op!=-1)
                {
                    ++count;
                    String str = subString(expr,op-1,op+2);
```

```java
                    map.put("T"+count,str);
                    replace(expr,op-1,op+2,"T"+count);
                }
        }

        Set<Map.Entry<String,String>> set= map.entrySet();
        for(Map.Entry<String,String> entry : set)
        {
            System.out.println(entry.getKey()+"="+entry.getValue());
        }
        System.out.println(subString(expr,0,expr.size()));
    }

        public static String subString(List<String> expr,int l,int r)
        {
            String str="";
            for(int i=l;i<r;i++)
            {
                str=str+expr.get(i);
            }
            return str;
        }
    public static void replace(List<String> expr,int l,int r,String temp)
        {
            int t = r-l;
            for(int i=0;i<t;i++)
            {
                expr.remove(l);
            }
            expr.add(l,temp);
        }
    public static List<String> sublist(List<String> expr,int l,int r)
        {
            List<String> l1 = new ArrayList<String>();
            String str = "T"+(++count);
            l1.add(str);
            l1.add("=");
            for(int i=l+1;i<r;i++)
            {
                l1.add(expr.get(i));
            }
            int t = r-l+1;
            for(int i=0;i<t;i++)
            {
                expr.remove(l);
            }
            expr.add(l,str);
            return l1;
        }
        }
```

Output:



```
Command Prompt

E:\Shubham Agrawal>javac ThreeAddrCode.java

E:\Shubham Agrawal>java ThreeAddrCode

****Each symbol should be separated by space delimiter****

Enter the Expression : A = A * ( B + C ) / D

Three Address Code for the following expression is :

T1=B+C
T2=A*T1
A=T2/D

E:\Shubham Agrawal>
```



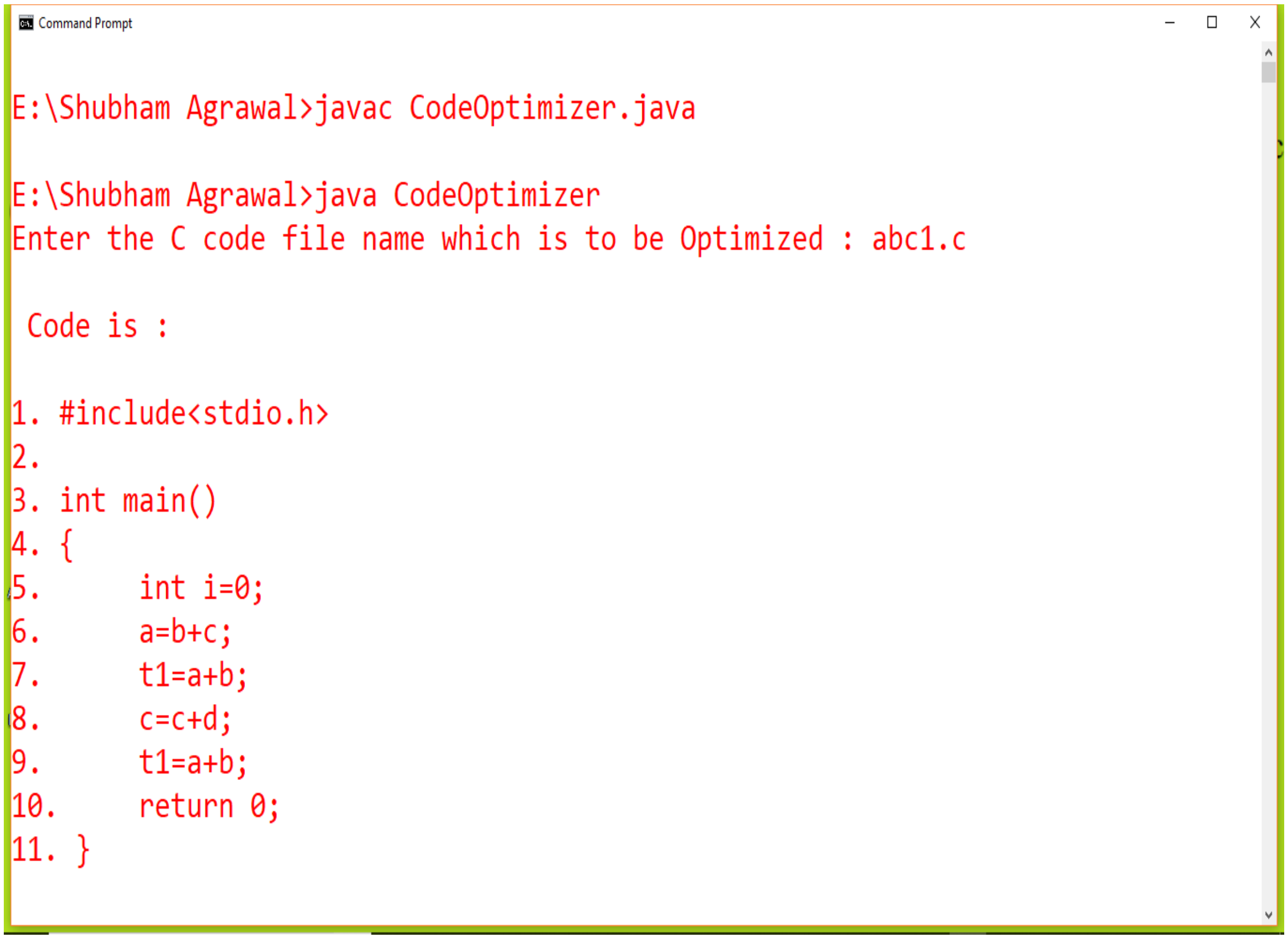
```
Command Prompt
E:\Shubham Agrawal>javac ThreeAddrCode.java

E:\Shubham Agrawal>java ThreeAddrCode

****Each symbol should be separated by space delimiter****

Enter the Expression : A = A + ( B + C ) / F - G * ( H + B )

Three Address Code for the following expression is :

T1=B+C
T2=H+B
T3=T1/F
T4=G*T2
T5=A+T3
A=T5-T4

E:\Shubham Agrawal>
```

EXPERIMENT-8

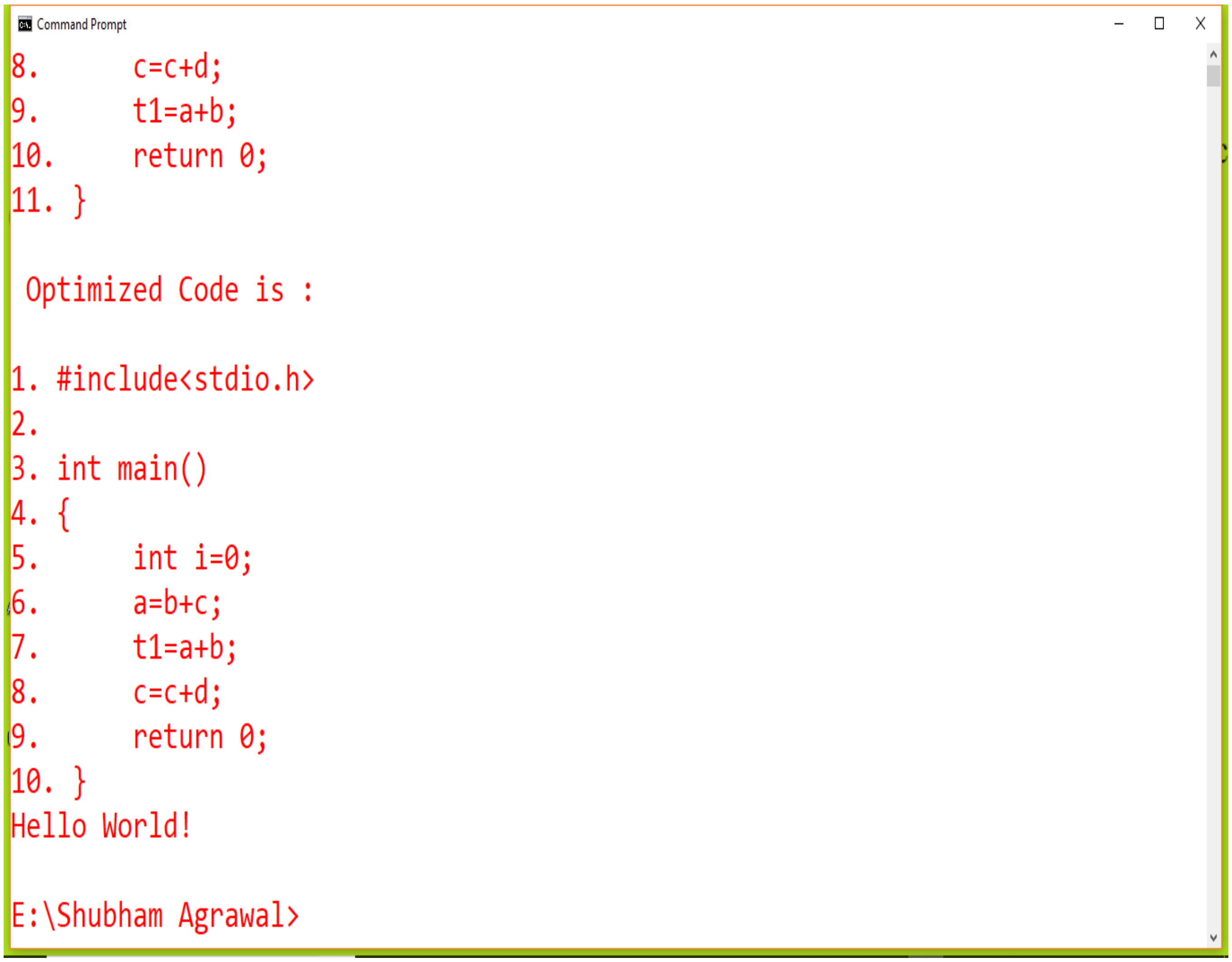
Aim: Program to design Code Optimizer.

```java
import java.util.*;
import java.io.*;

class CodeOptimizer
{
    public static void main(String[] args)
    {
        System.out.print("Enter the C code file name which is to be Optimized : ");
        Scanner sc = new Scanner(System.in);
        String path = sc.next();

        ArrayList<String> list = new ArrayList<String>();

        File file = null;
        FileReader fir = null;
        BufferedReader br = null;

        System.out.println("\n Code is : \n");
        try{
            file = new File(path);
            fir = new FileReader(file);
            br = new BufferedReader(fir);
            String line="";

            int count=0;
            while((line=br.readLine())!=null)
            {
                System.out.println((++count)+". "+line);
                if(!list.contains(line))
                {
                    list.add(line);
                }
            }


            System.out.println("\n Optimized Code is : \n");
            Iterator itr = list.iterator();
            count=0;
            while(itr.hasNext())
            {
                System.out.println((++count)+". "+itr.next());
            }

        }
        catch(Exception e )
        {
            System.out.println(e);
        }
        System.out.println("Hello World!");
    }
}
```

Output:



```
E:\Shubham Agrawal>javac CodeOptimizer.java

E:\Shubham Agrawal>java CodeOptimizer
Enter the C code file name which is to be Optimized : abc1.c

 Code is :

1. #include<stdio.h>
2.
3. int main()
4. {
5.      int i=0;
6.      a=b+c;
7.      t1=a+b;
8.      c=c+d;
9.      t1=a+b;
10.     return 0;
11. }
```



```
8.      c=c+d;
9.      t1=a+b;
10.     return 0;
11. }

 Optimized Code is :

1. #include<stdio.h>
2.
3. int main()
4. {
5.      int i=0;
6.      a=b+c;
7.      t1=a+b;
8.      c=c+d;
9.      return 0;
10. }
Hello World!

E:\Shubham Agrawal>
```

EXPERIMENT – 9

Aim: Case study on Lex compiler

1. Introduction.

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a
deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream. The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system. *Lex* generates programs to be used in simple lexical analysis of text. The input *files* (standard input default) contain regular expressions to be searched for and actions written in C to be executed when expressions are found.

A C source program, lex.yy.c is generated. This program, when run, copies unrecognized portions of the input to the output, and executes the associated C action for each regular expression that is recognized.

The options have the following meanings.

–t Place the result on the standard output instead of in file lex.yy.c.
–v Print a one–line summary of statistics of the generated analyzer.
–n Opposite of –v; –n is default.
–9 Adds code to be able to compile through the native C compilers.

2. Lex Source.

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

3. Lex Regular Expressions.

The definitions of regular expressions are very similar to those in QED [5]. A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

```
integer
```

matches the string integer wherever it appears and the expression

```
a57D
```

looks for the string a57D.

Operators. The operator characters are

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters.

Thus

```
xyz"++"
```

matches the string xyz++ when it appears.

4. Ambiguous Source Rules.

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1) The longest match is preferred.
2) Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules
integer keyword action ...;
[a-z]+ identifier action ...;

to be given in that order. If the input is integers, it is taken as an identifier, because [a-z]+ matches 8 characters while integer matches only 7. If the input is integer, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. int) will not match the expression integer and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example, '.*' might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

'first' quoted string here, 'second' here
the above expression will match
'first' quoted string here, 'second'

which is probably not what was wanted. A better rule is of the form
'[^'\n]*'
which, on the above input, will stop after 'first'. The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus expressions like .* stop on the current line. Don't try to defeat this with expressions like (.|\n)+ or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

EXAMPLE
This program converts upper case to lower, removes blanks at the end of lines, and replaces multiple blanks by single blanks.

```
%%
[A–Z]       putchar(yytext[0]+'a'–'A');
[ ]+$
[ ]+ putchar(' ')
```

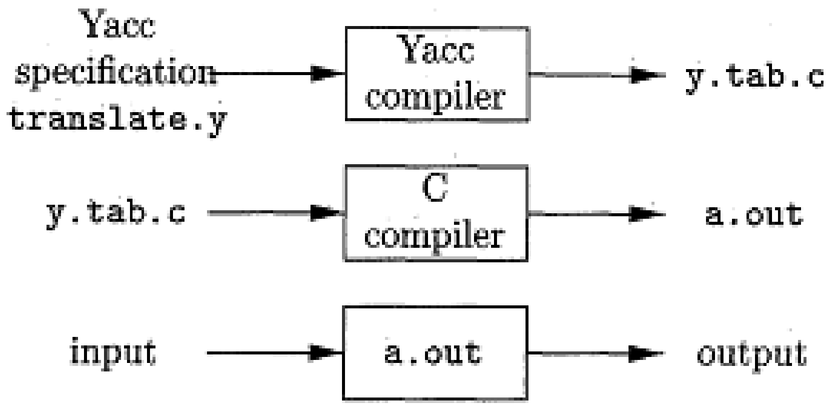
EXPERIMENT – 10

Aim: Case study on Yacc compiler

1. Introduction.

yacc is a computer program that serves as the standard parser generator on Unix systems. The name is an acronym for "Yet Another Compiler Compiler." It generates a parser (the part of a compiler that tries to make sense of the input) based on an analytic grammar written in BNF notation. Yacc generates the code for the parser in the C programming language.

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.



Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

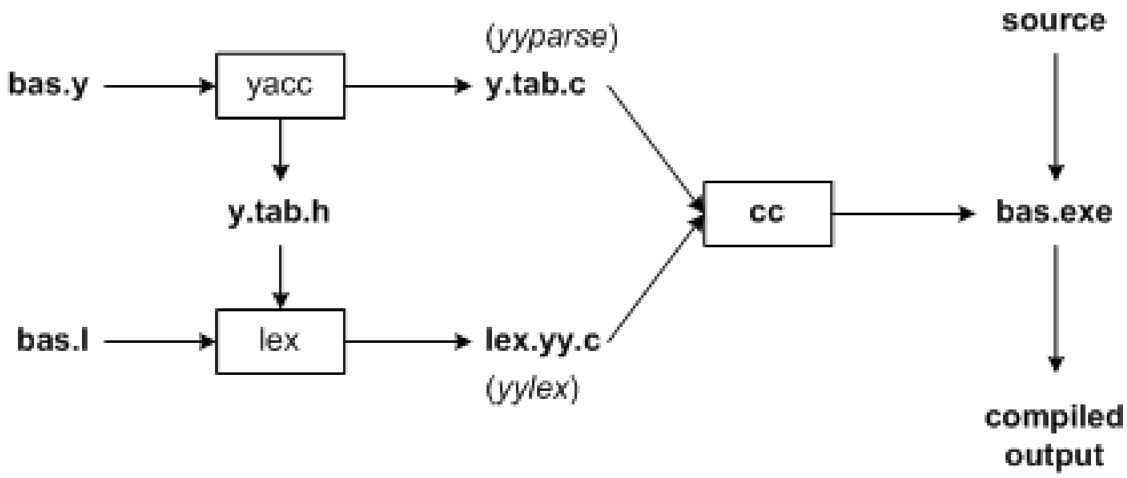
In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

Yacc converts a context–free grammar and translation code into a set of tables for an LR(1) parser and translator. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file, y.tab.c, must be compiled by the C compiler to produce a program yyparse. This program must be loaded with a lexical analyzer function, yylex(void) (often generated by *lex*(1)), with a main(int argc, char *argv[]) program, and with an error handling routine, yyerror(char*).

The options are:

–o *output* Direct output to the specified file instead of y.tab.c.
–D*n* Create file y.debug, containing diagnostic messages.
v Create file y.output, containing a description of the parsing tables and of conflicts arising from ambiguities in the grammar.
–d Create file y.tab.h, containing #define statements that associate *yacc*–assigned `token codes' with user–declared `token names'. Include it in source files other than y.tab.c to give access to the token codes.
–s *stem* Change the prefix y of the file names y.tab.c, y.tab.h, y.debug, and y.output to *stem*.
–S Write a parser that uses Stdio instead of the print routines in libc.



Lex and Yacc compiler