Compiler Design Lab Record(ALL PROGRAMS)

SYED.ATHEEQ
AP18110010299
CSE-E

Week 1: Implementation of Language recognizer

1. Implementation of Language recognizer for set of all strings over input alphabet ∑={a,b} containing an even number of a's and even number of b's.

Aim: Implement a Language recognizer for set of all strings over input alphabet ∑={a,b} containing an even number of a's and even number of b's.

Procedure:

Input:

input //input string

Output:

Algorithm prints a message

"String accepted": If the input is acceptable by the language, "String not accepted" otherwise,

"Invalid token": If the input string contains symbols other than input alphabet.

Method:

```
state=0 //initial state
while((current=input[i++])!='\0')
{
    switch(state)
      case 0: if(current=='a')
                                        state=1
            ; else if(current=='b')
                                        state=2
            ;
                  else
                      Print "Invalid token" ;
      exit; case 1: if(current=='a')    state=0;
```

```
                else if(current=='b')
                                        state=3
                    ; else
                        Print "Invalid token" ;
        exit; case 2: if(current=='a')    state=3;
                else if(current=='b')
                                        state=0
                    ; else
                        Print "Invalid token" ;
        exit; case 3: if(current=='a')     state=2;
                else if(current=='b')
                                            state=1
                    ; else
                        Print "Invalid token" ; exit;
    end
switch end
while
//Print
output
if(state==0)
    Print ”String accepted”
else
    Print ”String not accepted
```

C Program:

```
#include<stdio.h
> void main(){
   int state=0,i=0;
   char current,input[20];
   printf("Enter input string \t :");
   scanf("%s",input);
   while((current=input[i++])!='\0')
   {
      switch(state)
      {
         case 0:
               if(current=='a'
               ) state=1;
            else
               if(current=='b')
               state=2;
```

```
        else
        {
           printf("Invalid
           token"); exit(0);
        }
        break;
case 1:
           if(current=='a'
           ) state=0;
        else
           if(current=='b')
           state=3;
        else
        {
           printf("Invalid
           token"); exit(0);
        }
        break;
case 2:
           if(current=='a'
           ) state=3;
        else
           if(current=='b')
           state=0;
        else
        {
           printf("Invalid
           token"); exit(0);
        }
        break;
case 3:
           if(current=='a'
           ) state=2;
        else
           if(current=='b')
           state=1;
        else
        {
           printf("Invalid token");
```

```
                exit(0);
            }
            break;
        }
    }
    if(state==0)
        printf("\n\nString accepted\n\n");
    else
        printf("\n\nString not accepted\n\n");
}
```

Input/Output:

Test case 1:
Input: ab
Output: String Accepted

Test Case 2:
Input: aaab
Output: String not Accepted

Test Case 3:
Input: abbaabba
Output: String Accepted

Test Case 4:
Input: ababac
Output: Invalid token

2. Implementation of Language recognizer for set of all strings ending with two symbols of the same type.

Aim: Implement a Language recognizer for a set of all strings ending with two symbols of the same type.

Procedure:

Input:
input //input string

Output:
Algorithm prints a message
"String accepted": If the input is acceptable by the
language, "String not accepted" otherwise,
"Invalid token": If the input string contains symbols other than the input

alphabet.

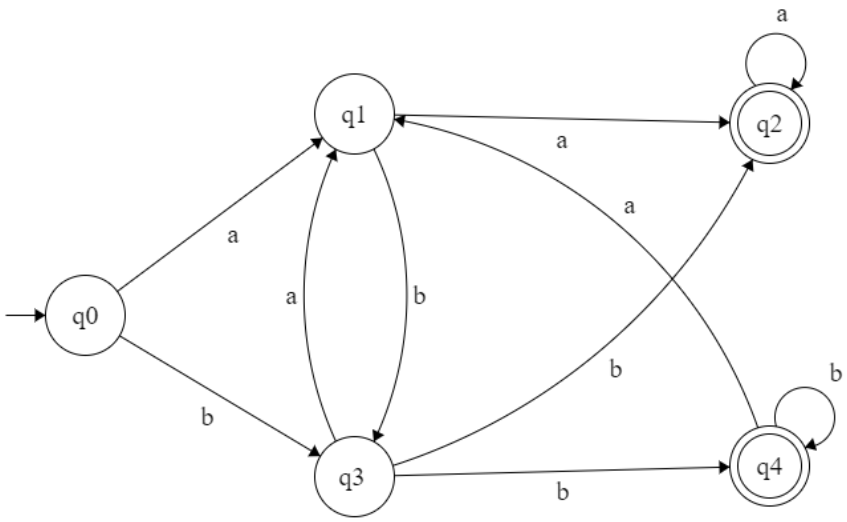
`

Pseudo Code:

```
state=0 //initial state
while((current=input[i++])!='\0')
{
    switch(state)
case 0:
        if (current == 'a')
          state = 1;

        else if (current == 'b')
          state = 3;

        else

          {

printf ("Invalid token");
printf(" : '%c'", current);
            exit (0);

}

break;

case 1:
        if (current == 'a')
          state = 2;

        else if (current == 'b')
          state = 3;

        else

          {

printf ("Invalid token");
printf(" : '%c'", current);
            exit (0);

}

break;

case 2:
        if (current == 'a')
          state = 2;
```

```
        else if (current == 'b')
          state = 3;

        else

          {

printf ("Invalid token");
printf(" : '%c'", current);
            exit (0);

}

break;

case 3:
        if (current == 'a')
          state = 1;

        else if (current == 'b')
          state = 4;

        else

          {

printf ("Invalid token");
printf(" : '%c'", current);



exit (0);

}

break;


case 4:
        if (current == 'a')
          state = 1;

        else if (current == 'b')
          state = 4;

        else

          {

printf ("Invalid token");
printf(" : '%c'", current);
```

```
exit (0);

}

break;

}

}


if (state == 2 || state == 4)

printf ("\n\nString accepted\n\n");
else

printf ("\n\nString not accepted\n\n");

}
```



C Program:

```
#include<stdio.h
> void main(){
    int state=0,i=0;
    char current,input[20];
    printf("Enter input string \t :");
    scanf("%s",input);
    while((current=input[i++])!='\0')
    {
        switch(state)
        {
case 0:
 if (current == 'a')
   state = 1;

 else if (current == 'b')
   state = 3;

 else
   {
printf ("Invalid token");
printf(" : '%c'", current);
exit (0);
    }

break;



case 1:
 if (current == 'a')
   state = 2;

 else if (current == 'b')
   state = 3;

   else
   {
printf ("Invalid token");
printf(" : '%c'", current);
```

```
    exit (0);
}

break;

case 2:
 if (current == 'a')
   state = 2;

 else if (current == 'b')
   state = 3;

 else
   {
printf ("Invalid token");
printf(" : '%c'", current);
    exit (0);
}

break;



case 3:
 if (current == 'a')
   state = 1;

 else if (current == 'b')
   state = 4;

 else
   {
printf ("Invalid token");
printf(" : '%c'", current);
exit (0);
}
break;


case 4:
 if (current == 'a')
   state = 1;
```

```
 else if (current == 'b')
   state = 4;

 else
   {
printf ("Invalid token");
printf(" : '%c'", current);
exit (0);
}

break;

}

}


if (state == 2 || state == 4)

printf ("\n\nString accepted\n\n");
else

printf ("\n\nString not accepted\n\n");

}
```

Input/Output:

aabb	String accepted
abab	String not accepted
aaabc	Invalid token
bbab	String not accepted
aaa	String accepted
aabababa	String not accepted
caab	Invalid token

Week 2 : Programs using Lex tool

1. Programs using Lex tool

a. Identification of Vowels and Consonants

Aim: Write a program using LEX tool for Identification of Vowels and Consonants

Procedure:

1. Define a string.
2. Convert the character to lowercase so that comparisons can be reduced. Else we need to compare with capital **(A, E, I, O, U)**.
3. If the input character in string matches with vowels **(a, e, i, o, u)** then display the output that the given character is a vowel.
4. If any character lies between 'a' and 'z' except vowels, then then display the output that the given character is NOT a vowel.

C Program:

```
%{
        /*To find whether given letter is a vowel or not*/
#undef yywrap
#define yywrap() 1
        void display(int);
%}

%%

[a|e|i|o|u|] {
                int flag=1;
                display(flag);
                return;
                }

.+ {
        int flag=0;
        display(flag);
        return;
        }
```

```
%%

void display(int flag)
{
        if(flag==1)
                printf("The given letter [%s] is a vowel",yytext);
        else
                printf("The given letter [%s] is NOT a vowel",yytext);
}

int main()
{
        printf("Enter a letter to check if it is a vowel or not:   ");
        yylex();
}
```

Input/Output:

Testcase1:

```
Enter a letter to check if it is a vowel or not:    a
"The given letter a  is a vowel
```

Testcase2:

```
Enter a letter to check if it is a vowel or not:    c
"The given letter c  is NOT a vowel
```

b. count number of vowels and consonants

Aim: Write a program using LEX tool to count number of vowels and consonants

Procedure:

1. Define a string.
2. Convert the string to lowercase so that comparisons can be reduced. Else we need to compare with capital **(A, E, I, O, U)**.

3. If any character in string matches with vowels **(a, e, i, o, u)** then increment the vcount by 1.
4. If any character lies between 'a' and 'z' except vowels, then increment the count for ccount by 1.
5. Print both the counts.

C Program:

```
%{

#include<stdio.h>

int vcount=0,ccount=0;

%}

%%

[a|i|e|o|u|E|A|I|O|U] {vcount++;}

[a-z A-Z(^a|i|e|o|u|E|A|I|O|U) ] {ccount++;}

%%

int main()

{

yylex();

printf("No. of vowels:%d\n No.of Consonants :%d\n",vcount,ccount);

return 1;

}

int yywrap()

{

}
```

Input/Output:

```
C:\Users\hp\Desktop\flex>lex vowel1.l

C:\Users\hp\Desktop\flex>gcc lex.yy.c

C:\Users\hp\Desktop\flex>a.exe
Enter the string of vowels and consonents:djfkd

hai

Number of vowels are:  2
Number of consonants are:  6
```

c. Count the number of Lines in given input

Aim: Write a program using LEX tool to Count the number of Lines in given input

Procedure:

begin
num_lines=0
num_chars=0
if new line
then num_lines++
end

LOGIC:

Read each character from the text file :

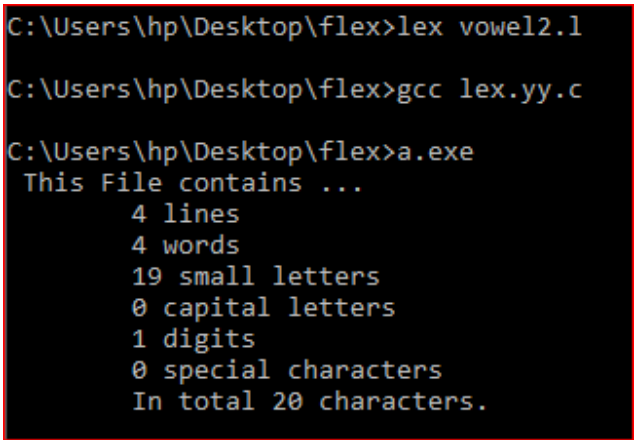
- Is it a capital letter in English? [A-Z] : increment capital letter count by 1.
- Is it a small letter in English? [a-z] : increment small letter count by 1
- Is it [0-9]? increment digit count by 1.
- All other characters (like '!', '@','&') are counted as special characters
- How to count the number of lines? we simply count the encounters of '\n' <newline> character.that's all!!
- To count the number of words we count white spaces and tab character(of course, newline characters too..)

C Program:

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

```
\n { lines++; words++;}
[\t ' '] words++;
[A-Z] c_letters++;
[a-z] s_letters++;
[0-9] num++;
. spl_char++;
%%
main(void)
{
yyin= fopen("myfile.txt","r");
yylex();
total=s_letters+c_letters+num+spl_char;
printf(" This File contains ...");
printf("\n\t%d lines", lines);
printf("\n\t%d words",words);
printf("\n\t%d small letters", s_letters);
printf("\n\t%d capital letters",c_letters);
printf("\n\t%d digits", num);
printf("\n\t%d special characters",spl_char);
printf("\n\tIn total %d characters.\n",total);
}
int yywrap()
{
return(1);
}
```

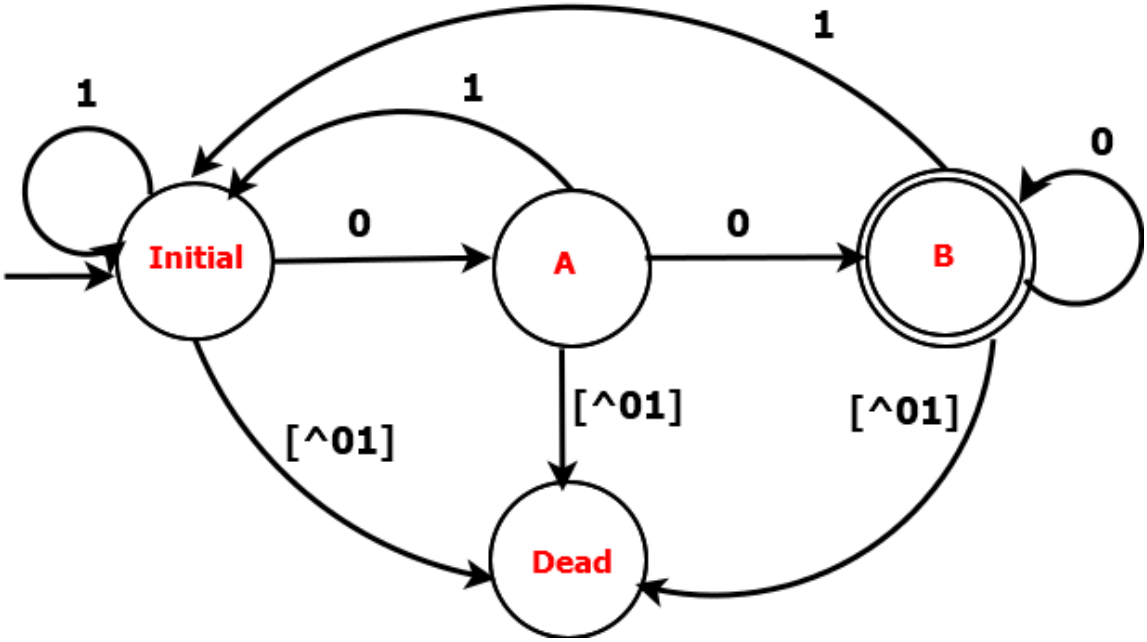
Input/Output:



d. Recognize strings ending with 00

Aim: Write a program using LEX tool to recognize strings ending with 00

Procedure: LEX provides us with an INITIAL state by default. So in order to make a DFA, use this as the initial state of the DFA. Now we define three more states A, B and DEAD where DEAD state would be used if we encounter a wrong or invalid input. When the user input invalid characters, move to DEAD state and print message “INVALID” and if input string ends at state B then display a message “Accepted”. If input string ends at state INITIAL and A then display a message “Not Accepted”.

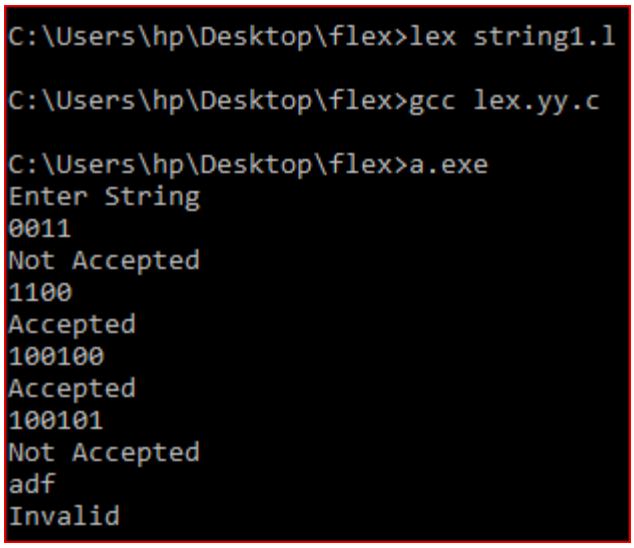


C Program:

```
%{
%}
  %s A B DEAD
  %%
<INITIAL>0 BEGIN A;
<INITIAL>1 BEGIN INITIAL;
<INITIAL>[^01\n] BEGIN DEAD;
<INITIAL>\n BEGIN INITIAL; {printf("Not Accepted\n");}
  <A>0 BEGIN B;
```

```
<A>1 BEGIN INITIAL;
<A>[^01\n] BEGIN DEAD;
<A>\n BEGIN INITIAL; {printf("Not Accepted\n");}
  <B>0 BEGIN B;
<B>1 BEGIN INITIAL;
<B>[^01\n] BEGIN DEAD;
<B>\n BEGIN INITIAL; {printf("Accepted\n");}
  <DEAD>[^\n] BEGIN DEAD;
<DEAD>\n BEGIN INITIAL; {printf("Invalid\n");}
 %%
int main()
{
    printf("Enter String\n");
    yylex();
}
int yywrap()
{
return 1;
}
```

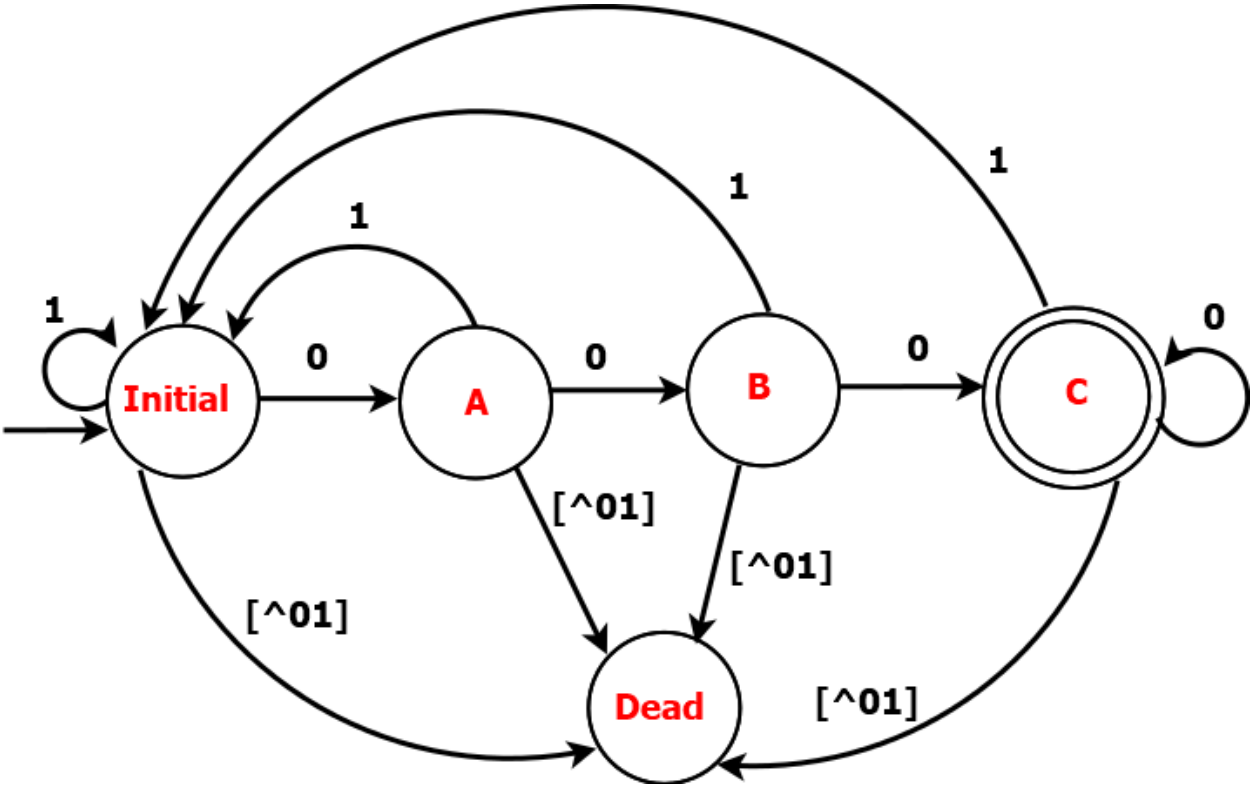
Input/Output:



e. Recognize a string with three consecutive 0's

Aim: Write a program using LEX tool to recognize a string with three consecutive 0's

Procedure:LEX provides us with an INITIAL state by default. So in order to make a DFA, use this as the initial state of the DFA. Now we define four more states A, B, C and DEAD were DEAD state would be used if we encounter a wrong or invalid input. When the user input invalid characters, move to DEAD state and print message "INVALID" and if input string ends at state C then display a message "Accepted". If input string ends at state INITIAL and A then display a message "Not Accepted".



C Program:

```
%{
%}

%s A B C DEAD

%%
<INITIAL>0 BEGIN A;
<INITIAL>1 BEGIN INITIAL;
<INITIAL>[^01\n] BEGIN DEAD;
<INITIAL>\n BEGIN INITIAL; {printf("Not Accepted\n");}

<A>0 BEGIN B;
<A>1 BEGIN INITIAL;
<A>[^01\n] BEGIN DEAD;
<A>\n BEGIN INITIAL; {printf("Not Accepted\n");}
```

```
<B>0 BEGIN C;
<B>1 BEGIN INITIAL;
<B>[^01\n] BEGIN DEAD;
<B>\n BEGIN INITIAL; {printf("Not Accepted\n");}

<C>0 BEGIN C;
<C>1 BEGIN INITIAL;
<C>[^01\n] BEGIN DEAD;
<C>\n BEGIN INITIAL; {printf("Accepted\n");}

<DEAD>[^\n] BEGIN DEAD;
<DEAD>\n BEGIN INITIAL; {printf("Invalid\n");}

%%
int main()
{
   printf("Enter String\n");
   yylex();
}
int yywrap()

{

return 1;

}
```

Input/Output:

```
C:\Users\hp\Desktop\flex>lex string2.l

C:\Users\hp\Desktop\flex>gcc lex.yy.c

C:\Users\hp\Desktop\flex>a.exe
Enter String
01000
Accepted
0111
Not Accepted
adcd
Invalid
000
Accepted
```

Week 3 Implement lexical analyzer using LEX

1. LEX Program for identifying the below and print the identified token along with information.

a. Keywords: int, char, double, void, main

b. Identifier: letter(letter|digit)*

c. Integer, Float and Relational operators*/

Aim: Write a Program to Design Lexical Analyzer by using LEX Tool.

Procedure:

Here, to write this program we have to follow some structure i.e, we need 3 sections to write the program, those are: I. Declaration Section

ii. Transition rules Section

iii. Auxiliary Function Section

Each Section ends with the symbol **"%%"** (A pair of percentages)

Declaration Section:

The declarations section consists of two parts, regular definitions and auxiliary declarations. LEX allows the use of short-hands and extensions to regular expressions for the regular definitions. The auxiliary declarations are copied as such by LEX to the output lex.yy.c file.

Example

```
%{
#include int global_variable;     //Auxiliary declarations
%}
number [0-9]+                    //Regular definitions
op [-|+|*|/|^|=]
%%
```

```
/* Rules */
%%
/* Auxiliary functions */
```

A regular definition in LEX is of the form : D R where D is the symbol representing the regular expression R. The auxiliary declarations (which are optional) are written in C language and are enclosed within ' %{ ' and ' %} ' . It is generally used to declare functions, include header files, or define global variables and constants.

Transition rules Section:

Rules in a LEX program consists of two parts:

i. The pattern to be matched
ii. The corresponding action to be executed

Example:

```
/* Declarations*/
%%
{number} {printf(" number");}
{op} {printf(" operator");}
%%
/* Auxiliary functions */
```

The pattern to be matched is specified as a regular expression

LEX obtains the regular expressions of the symbols number and op from the declarations section and generates code into a function yylex() in the lex.yy.c file. This function checks the input stream for the first match to one of the patterns specified and executes code in the action part corresponding to the pattern.

Auxiliary functions:

LEX generates C code for the rules specified in the Rules section and places this code into a single function called yylex(). (To be discussed in detail later). In addition to this LEX generated code, the programmer may wish to add his own code to the lex.yy.c file. The auxiliary functions section allows the programmer to achieve this.

Example:

```
/* Declarations */
%%
/* Rules */
%%
int main()
{
yylex();
return 1;
}
```

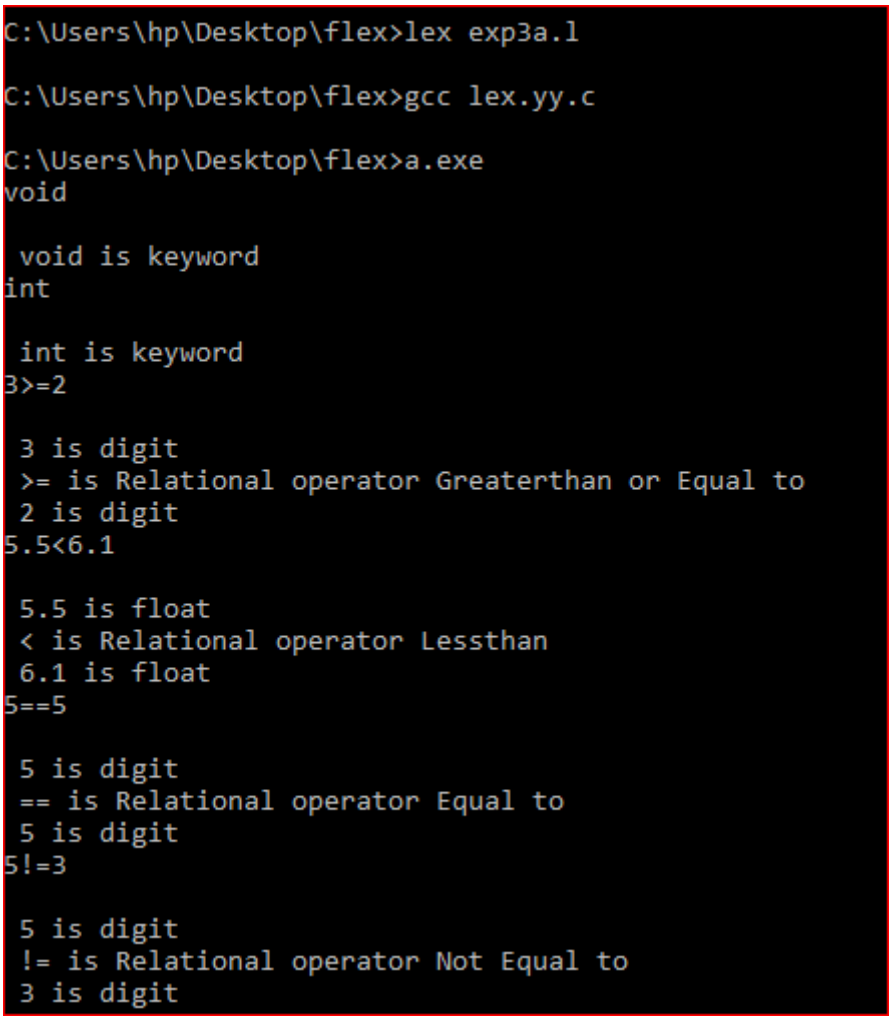
The C code in the auxiliary section and the declarations in the declaration section are copied as such to the lex.yy.c file.

C Program:

```
digit [0-9]*
        id [a-zA-Z][a-zA-Z0-9]*
        num [0-9]*\.[0-9]*

        %{
        %}
        %%
        int |
        float |
        char |
        double |
        void |
        main { printf(" \n %s is keyword",yytext);}
        "<=" {printf("\n %s is Relational operator Lessthan or Equal to",yytext);}
        "<" {printf("\n %s is Relational operator Lessthan",yytext);}
        ">=" {printf("\n %s is Relational operator Greaterthan or Equal to",yytext);}
        ">" {printf("\n %s is Relational operator Greaterthan",yytext);}
        "==" {printf("\n %s is Relational operator Equal to",yytext);}
        "!=" {printf("\n %s is Relational operator Not Equal to",yytext);}

        {id} {  printf("\n %s is identifier",yytext); }
        {num} { printf("\n %s is float",yytext);}
        {digit} {printf("\n %s is digit",yytext);}

        %%

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

Input/Output:

I: 234 O: number
I: int O: keyword
I: 2>=1 number relation operator number



2. Dealing with comments

Aim: Write a Program to Design Lexical Analyzer by using LEX Tool.

Procedure:

Here, to write this program we have to follow some structure i.e, we need 3 sections to write the program, those are: I. Declaration Section

ii. Transition rules Section

iii. Auxiliary Function Section

Each Section ends with the symbol **"%%"** (A pair of percentages)

Declaration Section:

The declarations section consists of two parts, regular definitions and auxiliary declarations. LEX allows the use of short-hands and extensions to regular expressions for the regular definitions. The auxiliary declarations are copied as such by LEX to the output lex.yy.c file.

A regular definition in LEX is of the form : D R where D is the symbol representing the regular expression R. The auxiliary declarations (which are optional) are written in C language and are enclosed within ' %{ ' and ' %} ' . It is generally used to declare functions, include header files, or define global variables and constants.

Transition rules Section:

Rules in a LEX program consists of two parts:

iii. The pattern to be matched

iv. The corresponding action to be executed

LEX obtains the regular expressions of the symbols number and op from the declarations section and generates code into a function yylex() in the lex.yy.c file. This function checks the input stream for the first match to one of the patterns specified and executes code in the action part corresponding to the pattern.

Auxiliary functions:

LEX generates C code for the rules specified in the Rules section and places this code into a single function called yylex(). (To be discussed in detail later). In addition to this LEX generated code, the programmer may wish to add his own code to the lex.yy.c file. The auxiliary functions section allows the programmer to achieve this.

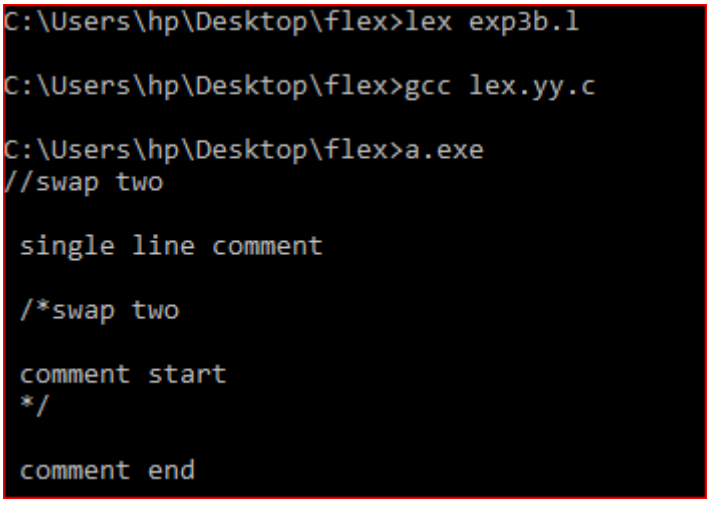
The C code in the auxiliary section and the declarations in the declaration section are copied as such to the lex.yy.c file.

C Program:

```
digit [0-9]*
id [a-zA-Z][a-zA-Z0-9]*
num [0-9]*\.[0-9]*
%{
int cnt=0,n=0,com=0,scom=0;
%}

%%
\n {scom=0;n++;}
"//" {scom=1;printf("\n single line comment\n\n");}
"/*" {com=1;printf("\n comment start\n");}
"*/" {com=0;printf("\n comment end\n");}
int |
float |
char |
double |
void |
main { if(!com&&!scom) printf(" \n %s is keyword",yytext);}
```

```
"<=" {if (!com&&!scom) printf("\n %s is Relational operator Lessthan or Equal to",yytext);}
        "<" {if(!com&&!scom) printf("\n %s is Relational operator Lessthan",yytext);}
        ">=" {if(!com) printf("\n %s is Relational operator Greaterthan or Equal to",yytext);}
        ">" {if(!com&&!scom) printf("\n %s is Relational operator Greaterthan",yytext);}
"==" {if(!com&&!scom) printf("\n %s is Relational operator Equal to",yytext);}
"!=" {if (!com&&!scom) printf("\n %s is Relational operator Not Equal to",yytext);}
{id} {  if(!com&&!scom) printf("\n %s is identifier",yytext); }
{num} { if(!com&&!scom) printf("\n %s is float",yytext);}
{digit} {if (!com&&!scom) printf("\n %s is digit",yytext);}
%%
int main()
{
 yylex();
 printf(" \n no of lines = %d\n",n);
return 0;
}
int yywrap()
{
return 1;
}
```

Input/Output:



Week 4 DESIGN OF LEXICAL ANALYZER FOR C LANGUAGE

Design a Lexical analyzer for identifying different types of tokens used in C language. Make a note of the following constraints.

- Comment line and whitespace must be eliminated
- The reserved keywords such as if, else, for, int, return, etc., must be reported as invalid identifiers.
- C allows identifier names to begin with underscore characters too.
- Print the number of tokens present in the input C program.

Aim: Design a Lexical analyzer for identifying different types of tokens used in C language

Procedure:

1. Get the space separated input C program
2. Read the input string left to right character by character for performing Tokenization .i.e. Dividing the program into valid tokens.
3. Check for identifiers by finding a string starts with an alphabet by using isalpha(), followed by alphabet or number or underscore.
4. Check for literal by finding a string constant enclosed within double quotes.
5. Check for operators such as +, -, *, /
6. Check for delimiters by identifying special symbols such as {:, ;, {, }, (,), ,}
7. Check for constants by identifying numbers by using isdigit() function
8. Remove white space characters.
9. Remove comments.
10. Print the various tokens by removing duplicates.

C Program:

```
%{ int n = 0 ;
%} %% [\n\t ' ']
{}; \/\/.* ;
\/\*(.*\n)*.*\*\/ ;
[a-zA-Z_][a-zA-Z0-9_]* {n++;printf("\t identifier : %s", yytext);}
[0-9]+ {n++;printf("\t integer : %s", yytext);}

[0-9]*"."[0-9]+ {n++;printf("\t float : %s", yytext);}
[(){}|, ;] {n++;printf("\t delimiters : %s", yytext);}
"<="|"=="|"="|"++"|"-"|"*"|"+" {n++;printf("\t operator : %s", yytext);}
"while"|"if"|"else" {n++;printf("\t keywords : %s", yytext);}
"int"|"float" {n++;printf("\t keywords : %s", yytext);}
```

```
. ;

%% int
yywrap() {
return 1; }
int main() {
yylex();

printf("\n total no. of token = %d\n", n);
}
```

Input/Output:

Week 5 Implementation of Recursive Descent Parser

Implementation of Recursive Descent Parser

Aim: Write a Program to Design Recursive Descent Parser

Procedure:

C Program:

```
#include&lt;stdio.h&gt;
#include&lt;string.h&gt;
int E(),Edash(),T(),Tdash(),F();
char *ip;
char string[50];
main()
{
   printf(&quot;Enter the string\n&quot;);
   scanf(&quot;%s&quot;,string);

   ip=string;
   printf(&quot;\n\nInput\tAction\n--------------------------------\n&quot;);
   if(E()){
      printf(&quot;\n--------------------------------\n&quot;);
      printf(&quot;\n String is successfully parsed\n&quot;);
   }
   else{
```

```
        printf(&quot;\n--------------------------------\n&quot;);
        printf(&quot;Error in parsing String\n&quot;);
    }
}
int E()
{
   printf(&quot;%s\tE-&gt;TE&#39; \n&quot;,ip);
    if(T())
    {
        if(Edash())
        {
            return 1;
        }
        else
            return 0;
    }
    else
        return 0;
}
int Edash()
{
    if(*ip==&#39;+&#39;)
    {
        printf(&quot;%s\tE&#39;-&gt;+TE&#39; \n&quot;,ip);
        ip++;
        if(T())
        {
            if(Edash())
            {
                return 1;
            }
            else
                return 0;

        }
        else
            return 0;
    }
    else
    {
        printf(&quot;%s\tE&#39;-&gt;^ \n&quot;,ip);
        return 1;
    }
}
```

```
int T()
{
   printf(&quot;%s\tT-&gt;FT&#39; \n&quot;,ip);
   if(F())
   {
      if(Tdash())
      {
         return 1;
      }
      else
         return 0;
   }
   else
      return 0;
}
int Tdash()
{
   if(*ip==&#39;*&#39;)
   {
      printf(&quot;%s\tT&#39;-&gt;*FT&#39; \n&quot;,ip);
      ip++;
      if(F())
      {
         if(Tdash())
         {
            return 1;
         }
         else
            return 0;
      }

      else
         return 0;
   }
   else
   {
     printf(&quot;%s\tT&#39;-&gt;^ \n&quot;,ip);
      return 1;
   }
}
int F()
{
   if(*ip==&#39;(&#39;)
   {
```

```
        printf(&quot;%s\tF-&gt;(E) \n&quot;,ip);
        ip++;
        if(E())
        {
            if(*ip==&#39;)&#39;)
            {
                ip++;
               return 0;
            }
            else
                return 0;
        }
        else
            return 0;
    }
    else if(*ip==&#39;i&#39;)
    {
        ip++;
          printf(&quot;%s\tF-&gt;id \n&quot;,ip);
        return 1;
    }
    else
        return 0;
}
```

Input/Output:

Week6 computation of FIRST and FOLLOW

Write a C program for the computation of FIRST and FOLLOW for a given CFG

Aim: Write a C program for the computation of FIRST and FOLLOW for a given CFG

Procedure:

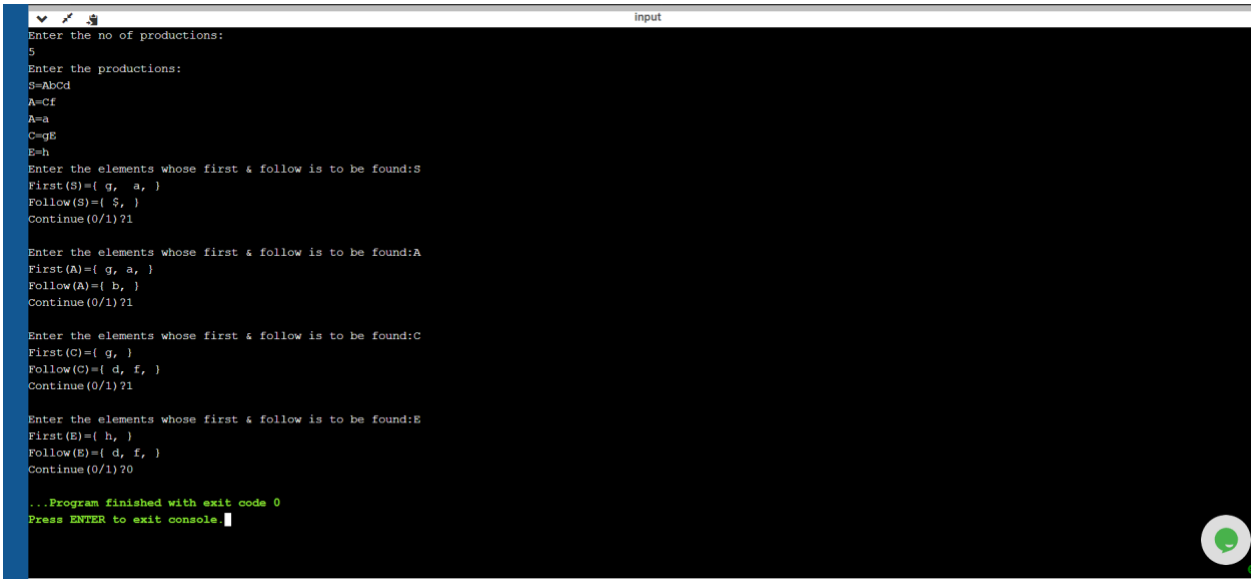
C Program:

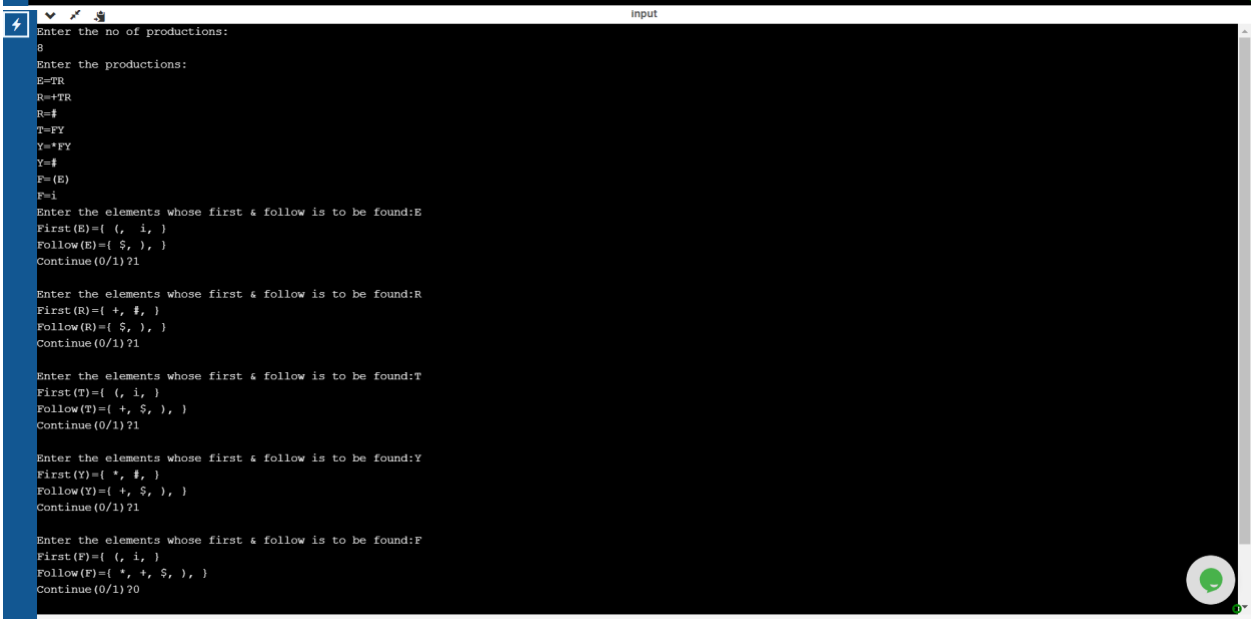
```
#include<stdio.h>
#include<math.h>
#include<string.h>
```

```
#include<ctype.h>
#include<stdlib.h>
int n, m = 0, p, i = 0, j = 0;
char a[10][10], f[10];

void first (char c)
{
int k;
if (!isupper (c)) f[m++] = c;
for (k = 0; k < n; k++)
{
if (a[k][0] == c)
{
if (a[k][2] == '$') follow (a[k][0]);
else if (islower (a[k][2])) f[m++] = a[k][2];
else first (a[k][2]);
}}}


void follow (char c)
{
if (a[0][0] == c) f[m++] = '$';

for (i = 0; i < n; i++)
{
for (j = 2; j < strlen (a[i]); j++)
{
if (a[i][j] == c)
{
if (a[i][j + 1] != '\0') first (a[i][j + 1]);
if (a[i][j + 1] == '\0' && c != a[i][0]) follow (a[i][0]);
}}}}


int main ()
{
int i, z;
char c, ch;

printf ("Enter the no of productions:\n");
scanf ("%d", &n);
printf ("Enter the productions:\n");
for (i = 0; i < n; i++) scanf ("%s%c", a[i], &ch);
```

```
do
{
m = 0;
printf ("Enter the elements whose first & follow is to be found:");
scanf ("%c", &c);
first (c);
printf ("First(%c)={ ", c);

for (i = 0; i < m; i++) printf ("%c", f[i]);
printf (" }\n");
strcpy (f, " ");

m = 0;
follow (c);
printf ("Follow(%c)={ ", c);


for (i = 0; i < m; i++) printf ("%c", f[i]);
printf (" }\n");

printf ("Continue(0/1)?");
scanf ("%d%c", &z, &ch);
}
while (z == 1);

return (0);
}
```

Input/Output:





Week 7 : Predictive Parser for the Expression Grammar

Implement a Predictive Parser for the Expression Grammar:

Aim: Implement a Predictive Parser for the Expression Grammar:

E -> TE'
E'->+TE' | ε
T->FT'
T'->*FT' | ε

F->(E) | i

Procedure:

C Program:

```
#include<stdio.h>
#include<stdlib.h>
void pop(),push(char ),display();
char stack[100]="\0";
char input[100];
int top=-1;
char *ip;
void main()
{
printf(" enter the input string followed by $ \n");
scanf("%s",input);
ip=input;
push('$');
push('E');
printf("STACK\t INPUT \t ACTION\n");
printf("------\t ------- \t ------\n");
printf("\n%s\t%s\tSHIFT",stack,ip);
while(stack[top]!='$')
{
if(stack[top]=='+' || stack[top]=='*'|| stack[top]=='i'|| stack[top]=='('|| stack[top]==')'||
stack[top]=='$')
{
if(stack[top]==*ip)
{
pop();
ip++;
printf("\n%s\t%s\tSHIFT",stack,ip);
printf("\n");
}
else
{
printf("\n error ");
exit(0);
}
}
else if(stack[top]=='E' && (*ip=='(' || *ip=='i'))
{
pop();
push('E');
```

```
push('T');
printf("\n%s\t%s\t",stack,ip);
printf("REDUCE BY E->TE'\n");
}
else if(stack[top]=='E' && (*ip==')' || *ip=='$'))
{
pop();
printf("\n%s\t%s\t",stack,ip);
printf("REDUCE BY E'->^\n");
}
else if(stack[top]=='E' && *ip=='+' )
{
pop();
push('E');
push('T');
push('+');
printf("\n%s\t%s\t",stack,ip);
printf("REDUCE BY E'->+TE'\n");
}
else if(stack[top]=='T' && (*ip=='(' || *ip=='i'))
{
pop();
push('T');
push('F');
printf("\n%s\t%s\t",stack,ip);
printf("REDUCE BY T->FT'\n");
}
else if(stack[top]=='T' && (*ip==')' || *ip=='$' || *ip=='+'))
{
pop();
printf("\n%s\t%s\t",stack,ip);
printf("REDUCE BY E->TE'\n");
}
else if(stack[top]=='T' && *ip=='*')
{
pop();
push('T');
push('F');
push('*');
printf("\n%s\t%s\t",stack,ip);
printf("REDUCE BY T'->*FT'\n");
}
else if(stack[top]=='F' && *ip=='(' )
{
pop();
push(')');
```

```
push('E');
push('(');
printf("\n%s\t%s\t",stack,ip);
printf("REDUCE BY F->(E)\n");
}
else if(stack[top]=='F' && *ip=='i')
{
pop();
push('i');
printf("\n%s\t%s\t",stack,ip);
printf("REDUCE BY F->id\n");
}
else
{
printf("\n error");
exit(0);
}
}
printf("\n%s\t%s\t",stack,ip);
printf(" Accept\n\n\n");
}
void push(char c)
{
top++;
stack[top]=c;
}
void pop()
{
stack[top]='\0';
top--;
}
```

Input/Output:

```
enter the input string followed by $

i*i+i$

STACK          INPUT          ACTION
------    -------  ------

$E       i*i+i$  SHIFT
$ET     i*i+i$  REDUCE BY E->TE'

$ETF   i*i+i$  REDUCE BY T->FT'
```

$ETi	i*i+i$	REDUCE BY F->id
$ET	*i+i$	SHIFT
$ETF*	*i+i$	REDUCE BY T'->*FT'
$ETF	i+i$	SHIFT
$ETi	i+i$	REDUCE BY F->id
$ET	+i$	SHIFT
$E	+i$	REDUCE BY E->TE'
$ET+	+i$	REDUCE BY E'->+TE'
$ET	i$	SHIFT
$ETF	i$	REDUCE BY T->FT'
$ETi	i$	REDUCE BY F->id
$ET	$	SHIFT
$E	$	REDUCE BY E->TE'
$	$	REDUCE BY E'->^
$	$	Accept

Week 8 : Implementation of Shift reduce parser

Aim: Implement a Shift reduce parser

Procedure:

C Program:

```
#include<stdio.h>
#include<stdlib.h>
void pop(),push(char ),display();
char stack[100]="\0";
char inputbuffer[100];
int top=-1;
```

```
char *ip;
void main()
{
printf("E->E+E\n");
printf("E->E*E\n");
printf("E->(E)\n");
printf("E->d\n");
printf(" enter the input string followed by $ \n");
scanf("%s",inputbuffer);
ip=inputbuffer;
push('$');
printf("STACK\t BUFFER \t ACTION\n");
printf("-----\t ------- \t ------\n");
display();
do
{
if((stack[top]=='E' && stack[top-1]=='$') && (*(ip)=='$'))
break;
if(stack[top]=='$')
{
push(*ip);
ip++;
printf("Shift");
}
else if(stack[top]=='d')
{
display();
pop();
push('E');
printf("Reduce E->d\n");
}
else if(stack[top]=='E' && stack[top-1]=='+' && stack[top-2]=='E'&& *ip!='*')
{
display();
pop();
pop();
pop();
push('E');
printf("Reduce E->E+E");
}
else if(stack[top]=='E' && stack[top-1]=='*' && stack[top-2]=='E')
{
display();
pop();
pop();
pop();
```

```
push('E');
printf("Reduce E->E*E");
}
else if(stack[top]==')' && stack[top-1]=='E' && stack[top-2]=='(')
{
display();
pop();
pop();
pop();
push('E');
printf("Reduce E->(E)");
}
else
{
display();
push(*ip);
ip++;
printf("shift");
}
}while(1);
display();
printf(" Accept\n\n\n");
}
void push(char c)
{
top++;
stack[top]=c;
}
void pop()
{
stack[top]='\0';
top--;
}
void display()
{
printf("\n%s\t%s\t",stack,ip);
}
```

Input/Output:

E->E+E
E->E*E
E->(E)
E->d

Enter the input string followed by $

d+d*d$

STACK	BUFFER	ACTION
$	d+d*d$	Shift
$d	+d*d$	Reduce E->d
$E	+d*d$	shift
$E+	d*d$	shift
$E+d	*d$	Reduce E->d
$E+E	*d$	shift
$E+E*	d$	shift
$E+E*d	$	Reduce E->d
$E+E*E	$	Reduce E->E*E
$E+E	$	Reduce E->E+E

Week 9 : Implementation of Shift reduce parser

Aim: Implement LALR parser using LEX and YACC for the following Grammar:

E → E+T |T

T→ T*F | F

F→(E)|id

Procedure:

C Program:

parser.l

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {yylval=atoi(yytext);
return DIGIT;
}
[\t] ;
\n return 0;
. return yytext[0];
%%
```

Parser1.y

```
%{
#include<stdio.h>
%}
%token DIGIT
%%
S: E { printf("The result is =%d\n",$1);}
;
E: E'+'T { $$ = $1 + $3; }
| T {$$ = $1;}
;
T: T'*'F {$$ = $1 * $3; }
| F {$$ = $1;}
;
F: '('E')' {$$ = $2;}
| DIGIT {$$ = $1;}
;
%%
main()
{
yyparse();
}
yyerror(char *s)
{
printf("%s",s);
}
```

Input/Output:

Enter Any Arithmetic Expression which can have operations Addition,Subtraction, Multiplication, Division,Modulus and Round brackets:
2*3+4

Result=10

Entered arithmetic expression is Valid

Week 10 : Implementation of Intermediate code generator a. Quadruple Generation

Aim: Implement LALR parser using LEX and YACC for the following Grammar:

E → E+T |T

T→ T*F | F

F→(E)|id

Procedure:

C Program:

Qual.l

```
%{
#include<stdio.h>
#include "y.tab.h"
#include<string.h>
%}
%%
[a-z]([a-z]|[0-9])* { strcpy(yylval.exp,yytext);
return VAR;
}
\t ;
\n return 0;
. return yytext[0];
%%
```

Quad.y

```
%{
#include<stdio.h>
#include<string.h>
struct quad
{
char op[5];
char arg1[10];
char arg2[10];
char result[10];
}QUAD[30];
int i=0,j;
```

```
%}
%union
{
char exp[10];
}
%token <exp> VAR
%type <exp> S E T F
%%
S: E { printf("\n THere are %d quadrupls n",i);
printf("\n List of Quadruples are: \n");
for(j=0;j<i;j++)
printf("%s\t%s\t%s\t%s\n",QUAD[j].op,QUAD[j].arg1,QUAD[j].arg2,QUAD[j].result);
}
;
E: E'+'T { printf("\n E ->E+T, $1=%s, $3=%s, $$=%s\n",$1,$3,$$);
strcpy(QUAD[i].op,"+");
strcpy(QUAD[i].arg1,$1);
strcpy(QUAD[i].arg2,$3);
strcpy(QUAD[i].result,$$);i++;
i++;
}
| T { printf("\n E -> T, $1=%s, $$=%s\n",$1,$$);}
;
T: T'*'F { printf("\n T -> T*F, $1=%s, $3=%s, $$=%s\n",$1,$3,$$);
strcpy(QUAD[i].op,"*");
strcpy(QUAD[i].arg1,$1);
strcpy(QUAD[i].arg2,$3);
strcpy(QUAD[i].result,$$);
i++;
}
| F { printf("\n T -> F, $1=%s, $$=%s\n",$1,$$);}
;
F: VAR {printf("\n F ->VAR and $1=%s, $$=%s \n",$1,$$);}
;
%%
main()
{
yyparse();
}
int yywrap(){
return 1;
}
yyerror(char *s)
{
printf("%s",s);   }
```

Input/Output:

a*b+c

F ->VAR and $1=a, $$=a

T -> F, $1=a, $$=a

F ->VAR and $1=b, $$=b

T -> T*F, $1=a, $3=b, $$=a

E -> T, $1=a, $$=a

F ->VAR and $1=c, $$=c

T -> F, $1=c, $$=c

E ->E+T, $1=a, $3=c, $$=a

THere are 3 quadrupls n
List of Quadruples are:

*	a	b	a
+	a	c	a