NATIONAL INSTITUTE OF TECHNOLOGY AGARTALA

COMPILER DESIGN LABORATORY

(UCS06B25)

Submission by:-

NAME : MD WASIF

Enrollment no: 19UICS002

Reg : 1910910

Sem & sec : 6TH & C

Submitted to:-

Dr. Diptendu Bhatacharya

Lab asst. Ardhendu Gupta

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

National institute of technology, Agartala

Jirania, Tripura-799046

ACKNOWLEDGEMENT

I would like to express my special thanks to my faculty members Dr. ***Diptendu Bhatacharya*** *(Assistant Professor) &* ***Ardhendu Gupta****(LAB/Technical Assistant) ,who gave me the golden opportunity to do this wonderful CD lab which also helped me in doing a lot of Research and I came to know about so many new things and valuable guidance and feedback has helped me in completing and learning these a lot.*
I would again like to thank my teacher for putting efforts to help us grasp this subject and for generating interest in it.
My thanks and appreciation also goes to all the teaching staff of the computer science and engineering department for their kind and timely support.

MD WASIF
19UICS002

INDEX

Experiment number	Name of experiment	Date	Signature
1	Write a C program to identify a given line is comment or not	19/01/2022	
2	Write a C program to recognize string under a*, a*b+, abb	25/01/2022	
3	Write a C program to check whether a given identifier is valid or not	25/01/2022	
4	Write a C program to simulate lexical analyzer for validating operators	01/02/2022	
5	Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and newlines	01/02/2022	
6	Write a C program for implementing the functionalities of predictive parser.	09/02/2022	
7	Implement the lexical analyzer using JLex, Flex or other lexical analyzer generating tools	15/02/2022	
8	Construction of a recursive descent parser	04/04/2022	
9	Write a c program to implement operator precedence parsing	04/04/2022	
10	C program to design LALR bottom up parser	11/04/2022	

EXPERIMENT - 1

Date : 19/01/2022

Write a C program to identify whether a given line is a comment or not.

Objective :-

To identify whether a given line is a comment or not.

Resource :-

VS CODE

Program Logic :-

- Check if at the first Index(i.e. index 0) the value is '/' then follow below steps else print "It is not a comment".
- If line[0] == '/':
    - If line[1] == '/', then print "It is a single line comment".
    - If line[1] == '*', then traverse the string and if any adjacent pair of '*' & '/' is found then prints "It is a multi-line comment".
- Otherwise, print "It is not a comment".

Procedure :-

- Go to debug-> run or press CTRL+F5 to run the program.

Program :-

```
#include <bits/stdc++.h>
using namespace std;

// Function to check if the given
// string is a comment or not
void isComment(string line)
{

    if (line[0] == '/' && line[1] == '/'
        && line[2] != '/')
         {

        cout << "It is a single-line comment";
        return;
    }

    if (line[line.size() - 2] == '*'
        && line[line.size() - 1] == '/' && line[0] == '/' && line[1] == '*')
         {

        cout << "It is a multi-line comment";
        return;
    }

    cout << "It is not a comment";
}

int main()
{
    // Given string
    string line = "/*Hello compiler Design*/";


    isComment(line);

    return 0;
```

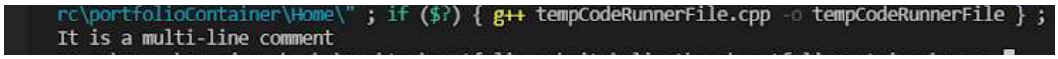
```
}
```

Input :-

“/*Hello compiler Design*/”

Output :-

It is a multi-line comment.



Conclusion :-

The C++ program to identify whether a given line is a comment or not is executed successfully.

EXPERIMENT - 2

Date : 25/01/2022

OBJECTIVE:-

Write a C program to recognize strings under 'a*', 'a*b+', 'abb'.

RESOURCE:-

Dev C++

PROGRAM LOGIC

By using a transition diagram we verify input of the state. If the state recognizes the given pattern rule.

Then print string is accepted under a*/ a*b+/ abb. Else print string not accepted.

PROCEDURE:-

Go to debug -> run or press CTRL + F9 to run the program.

PROGRAM:-

```
 #include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
        void main()
       {
       char s[20],c; int state=0,i=0; clrscr();
       printf("\n Enter a string:"); gets(s);
```

```
while(s[i]!='\0')
{
switch(state
```

```
{
case 0: c=s[i++]; if(c=='a')
 state=1;
else if(c=='b') state=2;
else state=6; break;
case 1: c=s[i++]; if(c=='a')
state=3;
else if(c=='b')
state=4; else state=6; break;
case 2: c=s[i++]; if(c=='a') state=6;
else if(c=='b')
state=2; else state=6;
break;
case 3: c=s[i++]; if(c=='a') state=3;
else if(c=='b') state=2;
else state=6; break;
case 4: c=s[i++]; if(c=='a') state=4;
```

```
else if(c=='b') state=5;
else state=6; break;
case 5: c=s[i++]; if(c=='a') state=6;
else if(c=='b') state=2;
else state=6; break;
case 6: printf("\n %s is not recognised.",s); exit(0);
}
}
if(state==1)
printf("\n %s is accepted under rule 'a'",s); else if((state==2)||(state==4))
printf("\n %s is accepted under rule 'a*b+'",s); else if(state==5)
printf("\n %s is accepted under rule 'abb'",s); getch();
}
```

INPUT & OUTPUT:

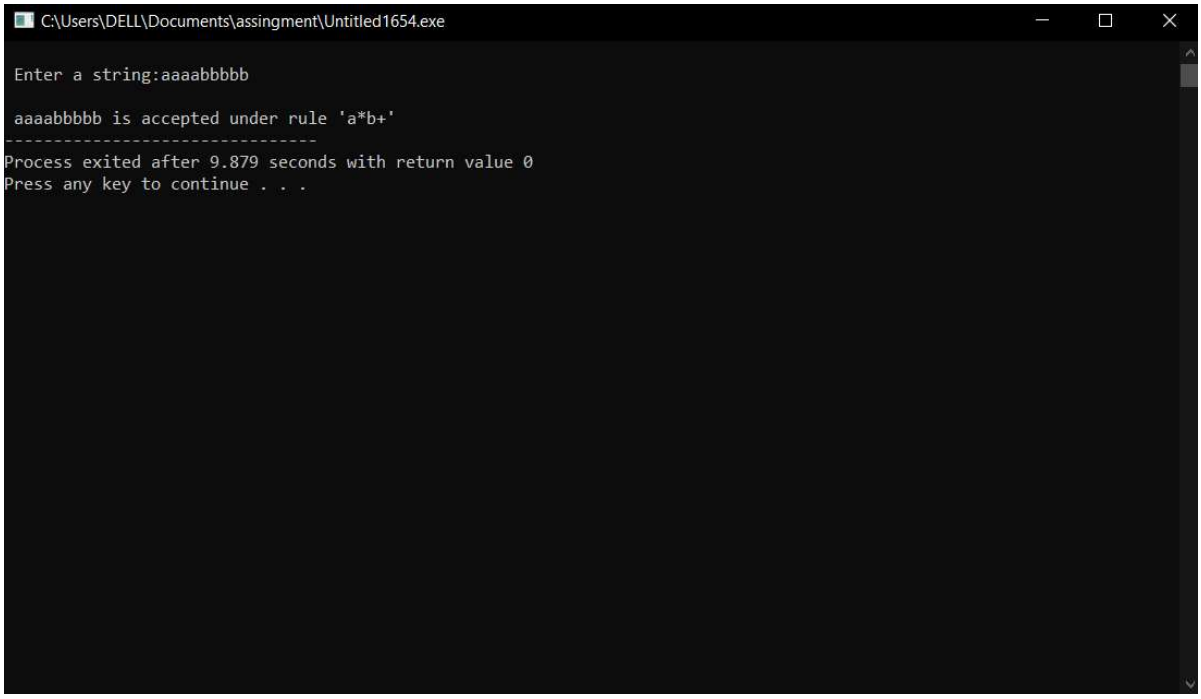
Input :-

Enter a String: aaaabbbbb

Output:-

aaaabbbbb is accepted under rule 'a*b+'

Enter a string: cdgs cdgs is not recognized



Conclusion:-

Thus, Program was implemented successfully and verified

EXPERIMENT - 3

Date : 25/01/2022

OBJECTIVE:-

Write a C program to recognize a identifier

RESOURCE:-

Dev C++

Program LOGIC:-

Read the given input string. Check the initial character of the string is numerical or any special character except '_' then print it is not a valid identifier. Otherwise print it as a valid identifier if the remaining characters of the string don't contain any special characters except '_'.

PROCEDURE:-

Go to debug -> run or press CTRL + F9 to run the program.

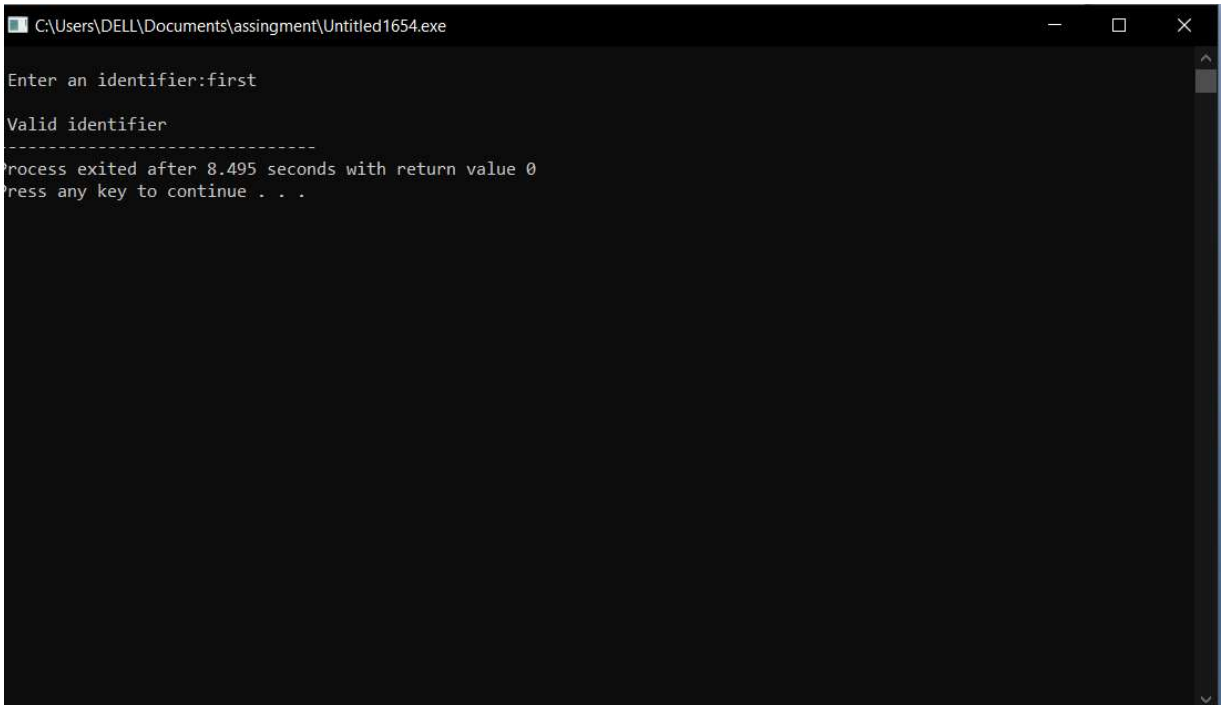
PROGRAM:-

```
#include<stdio.h>

#include<conio.h>

#include<ctype.h>

void main()

{

char a[10];

int flag, i=1;

clrscr();

printf("\n Enter an identifier:");

gets(a);
```

```
if(isalpha(a[0]))
flag=1;
else
printf("\n Not a valid identifier");
while(a[i]!='\0'){
if(!isdigit(a[i])&&!isalpha(a[i]))
{flag=0;
break;
}
i++;
}if(flag==1)
printf("\n Valid identifier");
getch();
}
```

Input & Output

```
Enter an identifier: first
Valid identifier
Enter an identifier:1aqw
Not a valid identifier
```



Conclusion:-

Program had been tested and verified

EXPERIMENT - 4

Date : 01/01/2022

Write a C program to simulate lexical analyzer for validating operators

Objective :-

To simulate lexical analyzer for validating operators

Resource :-

Atom editor with GCC Compiler

Program Logic :-

1. Read the given input.
2. If the given input matches with any operator symbol.
3. Then display in terms of words of the particular symbol.
4. Else print not a operator

Program :-

lex.c

```c
#include<stdio.h>
#include<conio.h>
void main()
{
char s[5];
printf("\nEnter any operator:");
gets(s);
switch(s[0])
{
case'>': if(s[1]=='=')
printf("\n Greater than or equal");
else
printf("\n Greater than");
break;
case'<': if(s[1]=='=')
printf("\n Less than or equal");
else
printf("\nLess than");
break;
```

```
case'=': if(s[1]=='=')
printf("\nEqual to");
else
printf("\nAssignment");
break;
case'!': if(s[1]=='=')
printf("\nNot Equal");
else
printf("\n Bit Not");
break;
case'&': if(s[1]=='&')
printf("\nLogical AND");
else
printf("\n Bitwise AND");
break;
case'|': if(s[1]=='|')
printf("\nLogical OR");
else
printf("\nBitwise OR");
break;
case'+': printf("\n Addition");
break;
case'-': printf("\nSubstraction");
break;
case'*': printf("\nMultiplication");
break;
case'/': printf("\nDivision");
break;
case'%': printf("Modulus");
break;
default: printf("\n Not a operator");
}
getch();
}
```

PROCEDURE:

1. Right-click on the program
2. Click on the Run Code
3. Or press F5 to run the program

Output:-

Conclusion:-

Thus , C program to Simulate lexical analyzer for validating operators successfully implemented .

EXPERIMENT - 5

Date : 01/01/2022

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.

Objective :-

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.

Resource :-

Atom editor with GCC Compiler

Program Logic :-

1. Read the input Expression
2. Check whether input is alphabet or digits then store it as identifier
3. If the input is is operator store it as symbol
4. Check the input for keywords

Program :-

lex.c

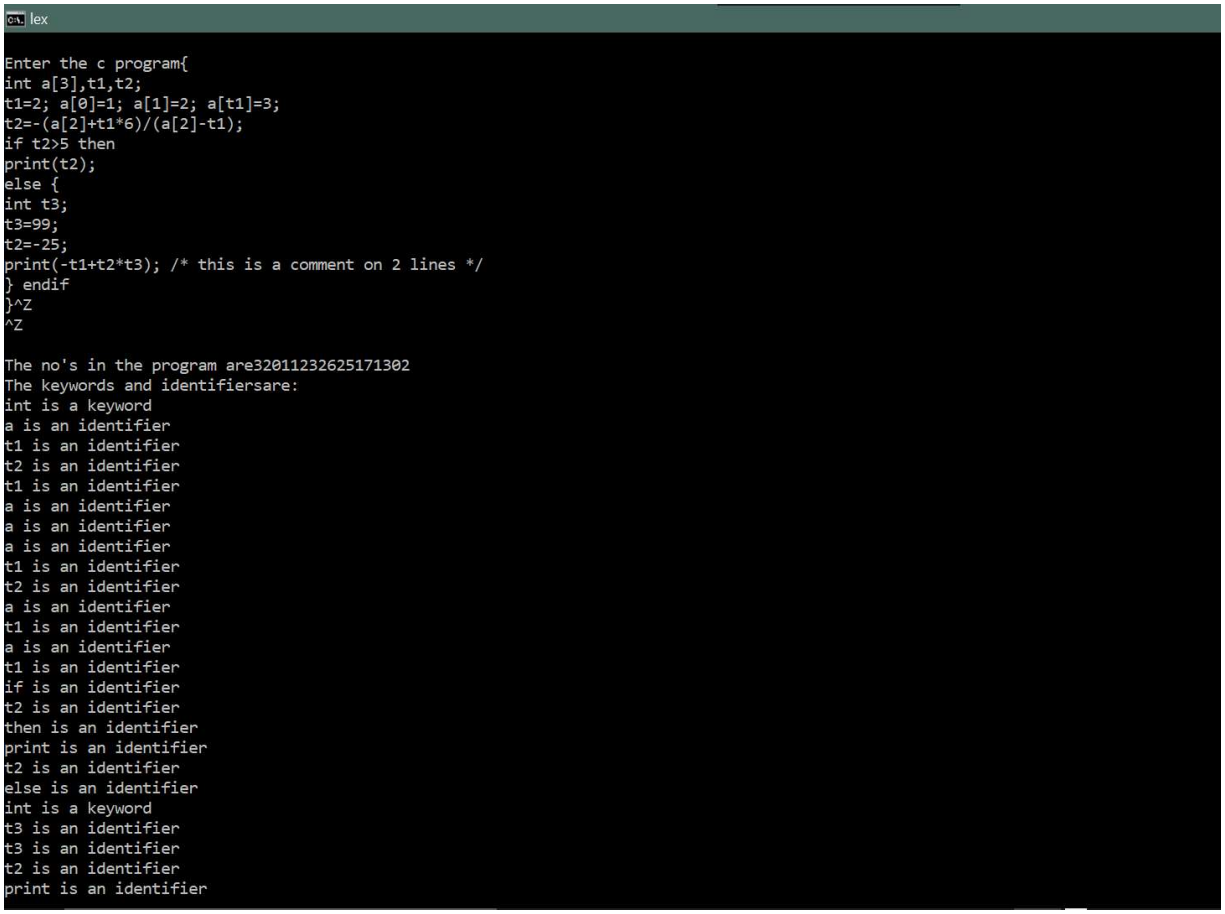
```c
#include<string.h>
#include<ctype.h>
#include<stdio.h>
void keyword(char str[10])
{
if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0|| strcmp("int",str)==0||strcmp("float",str)==0||strcmp("char",

strcmp("static",str)==0||strcmp("switch",str)==0||strcmp("case",str)==0)
printf("\n%s is a keyword",str);
else
printf("\n%s is an identifier",str);
}
```

```c
main()
{
FILE *f1,*f2,*f3;
char c,str[10],st1[10];
int num[100],lineno=0,tokenvalue=0,i=0,j=0,k=0;
printf("\nEnter the c program");/*gets(st1);*/
f1=fopen("input","w");
while((c=getchar())!=EOF)
putc(c,f1);
fclose(f1);
f1=fopen("input","r");
f2=fopen("identifier","w");
f3=fopen("specialchar","w");
while((c=getc(f1))!=EOF){
if(isdigit(c))
{
tokenvalue=c-'0';
c=getc(f1);
while(isdigit(c)){
tokenvalue*=10+c-'0';
c=getc(f1);
}
num[i++]=tokenvalue;
ungetc(c,f1);
}
else if(isalpha(c))
{
putc(c,f2);
c=getc(f1);
while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
{
```

```
{
putc(c,f2);
c=getc(f1);
}
putc(' ',f2);
ungetc(c,f1);
}
else if(c==' '||c=='\t')
printf(" ");
else
if(c=='\n')
lineno++;
else
putc(c,f3);
}
fclose(f2);
fclose(f3);
fclose(f1);
printf("\nThe no's in the program are");
for(j=0;j<i;j++)
printf("%d",num[j]);
printf("\n");
f2=fopen("identifier","r");
k=0;
printf("The keywords and identifiersare:");
while((c=getc(f2))!=EOF){
if(c!=' ')
str[k++]=c;
else
{
str[k]='\0';
keyword(str);
```

```
}
}
fclose(f2);
f3=fopen("specialchar","r");
printf("\nSpecial characters are");
while((c=getc(f3))!=EOF)
printf("%c",c);
printf("\n");
fclose(f3);
printf("Total no. of lines are:%d",lineno);
}
```

Output:-



PROCEDURE:-

1. Right-click on the program
2. Click on the Run Code
3. Or press F5 to run the program

Conclusion:-

Thus , C program to Design a lexical analyzer for a given language and the lexical analyzer successfully implemented .

EXPERIMENT - 6

Date : 09/01/2022

Write a C program for implementing the functionalities of predictive parser.

Objective :-

Implement the functionalities of predictive parser

Resource :-

Atom editor with GCC Compiler

Program Logic :-

1. Read the input string.
2. By using the FIRST AND FOLLOW values.
3. Verify the FIRST of non terminal and insert the production in the FIRST value
4. If we have any @ terms in FIRST then insert the productions in FOLLOW values
5. Constructing the predictive parser table

PROCEDURE:

1. Right-click on the program
2. Click on the Run Code
3. Or press F5 to run the program

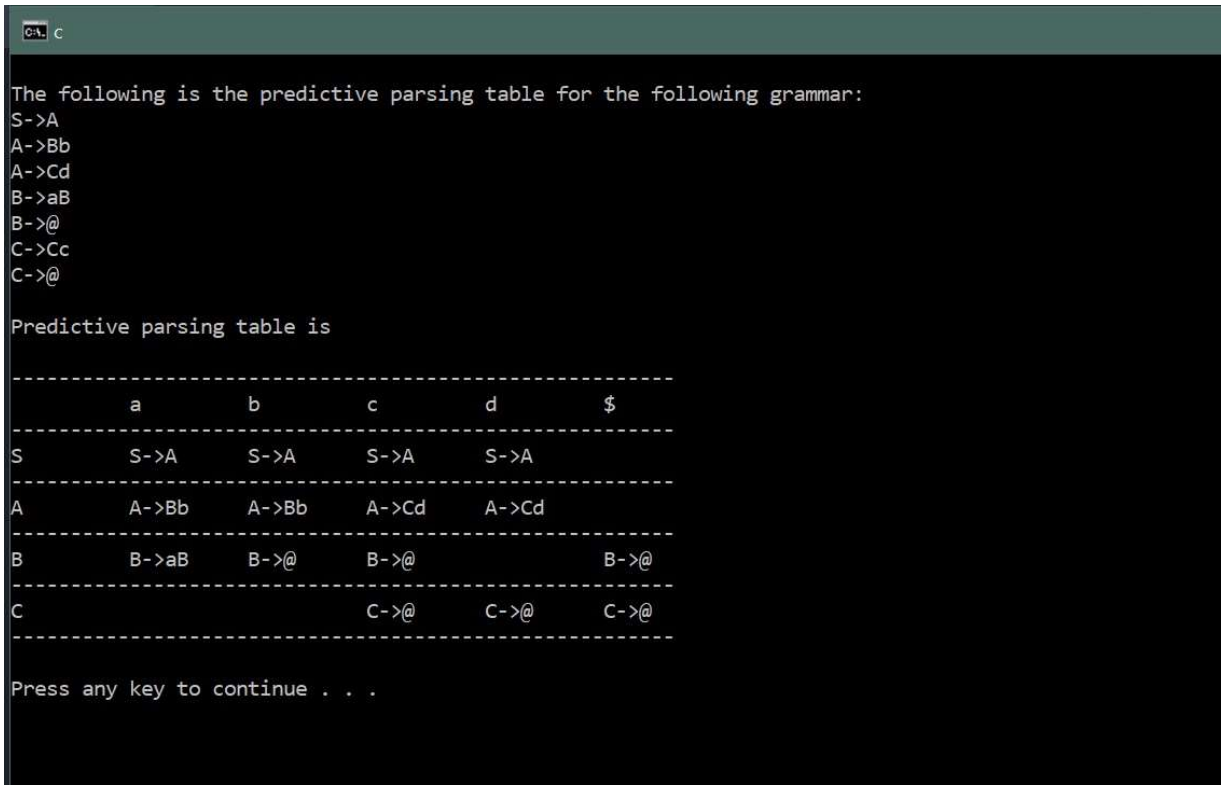
Program :-

```c
#include<stdio.h>
#include<conio.h>
#include<string.h>
char prol[7][10]={"S","A","A","B","B","C","C"};
char pror[7][10]={"A","Bb","Cd","aB","@","Cc","@"};
char prod[7][10]={"S->A","A->Bb","A->Cd","B->aB","B->@","C->Cc","C->@"};
char first[7][10]={"abcd","ab","cd","a@","@","c@","@"}; char follow[7][10]={"$","$","$","a$","b$","c$","d$"};
char table[5][6][10];
int numr(char c)
{
switch(c){
case 'S': return 0;
case 'A': return 1;
case 'B': return 2;
case 'C': return 3;
case 'a': return 0;
case 'b': return 1;
case 'c': return 2;
case 'd': return 3;
case '$': return 4;
}
return(2);
}
void main()
{
int i,j,k;
for(i=0;i<5;i++)
for(j=0;j<6;j++)
strcpy(table[i][j]," ");
printf("\nThe following is the predictive parsing table for the following grammar:\n"); for(i=0;i<7;i++)
printf("%s\n",prod[i]);
printf("\nPredictive parsing table is\n");
```

```
fflush(stdin);
for(i=0;i<7;i++){
k=strlen(first[i]);
for(j=0;j<10;j++)
if(first[i][j]!='@')
strcpy(table[numr(prol[i][0])+1][numr(first[i][j])+1],prod[i]);
}
for(i=0;i<7;i++){
if(strlen(pror[i])==1)
{
if(pror[i][0]=='@')
{
k=strlen(follow[i]);
for(j=0;j<k;j++)
strcpy(table[numr(prol[i][0])+1][numr(follow[i][j])+1],prod[i]);
}
}
}
strcpy(table[0][0]," ");
strcpy(table[0][1],"a");
strcpy(table[0][2],"b");
strcpy(table[0][3],"c");
strcpy(table[0][4],"d");
strcpy(table[0][5],"$");
strcpy(table[1][0],"S");
strcpy(table[2][0],"A");
strcpy(table[3][0],"B");
strcpy(table[4][0],"C");
printf("\n---------------------------------------------------------\n");
for(i=0;i<5;i++)
for(j=0;j<6;j++){
```

```
for(j=0;j<6;j++){
printf("%-10s",table[i][j]);
if(j==5)
printf("\n---------------------------------------------------------\n");
}
getch();
}
```

OUTPUT :-



Conclusion:-

Thus , the C program for implementing the functionalities of predictive parser was successful.

EXPERIMENT - 7

Date : 15/02/2022

Implement the lexical analyzer using JLex, flex or other lexical analyzer generating tools.

Objective :-

To implement the lexical analyzer

Resource :-

Linux using lex tool

Program Logic :-

1. Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%. The format is as follows: definitions %% rules %% user_subroutines
2. In the definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in %{..}%. Identifier is defined such that the first letter of an identifier is alphabet and remaining letters are alphanumeric.
3. In the rules section, the left column contains the pattern to be recognized in an input file to yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, new line character, number, string, identifier, beginning and end of block, comment statements, preprocessor directive statements etc.
4. Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.
5. When yylex() matches a string in the input stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.
6. In the user subroutine section, the main routine is called yylex(). yywrap() is used to get more input.

7. The lex command uses the rules and actions contained in the file to generate a program, lex.yy.c, which can be compiled with the cc command. That program can then receive input, break the input into the logical pieces defined by the rules in the file, and run program fragments contained in the actions in the file.

Procedure:

Go to terminal .Open vi editor ,Lex filename.l , cc lex.yy.c , ./a.out

Program:-

```
/* program name is abc.l */

Thunderbird Mail
/* program to recognize a c program */

int COMMENT=0;

%}

identifier [a-zA-Z][a-zA-Z0-9]*S

%%

#.* { printf("\n%s is a PREPROCESSOR DIRECTIVE",yytext);}

int |float |char |double |while |for |do |if |break |continue |void |switch |case |long |struct |const |typedef |return

|else |goto {printf("\n\t%s is a KEYWORD",yytext);}

"/*" {COMMENT = 1;}

/*{printf("\n\n\t%s is a COMMENT\n",yytext);}*/

"*/" {COMMENT = 0;}

/* printf("\n\n\t%s is a COMMENT\n",yytext);}*/

{identifier}\( {if(!COMMENT)printf("\n\nFUNCTION\n\t%s",yytext);}

{ {if(!COMMENT) printf("\n BLOCK BEGINS");} } {if(!COMMENT) printf("\n BLOCK ENDS");}

{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);} ".*\" {if(!COMMENT) printf("\n\t%s is a STRING",yytext);}

[0-9]+ {if(!COMMENT) printf("\n\t%s is a NUMBER",yytext);} {if(!COMMENT) printf("\n\t");ECHO;printf("\n");}

( ECHO;

{if(!COMMENT)printf("\n\t%s is an ASSIGNMENT OPERATOR",yytext);}

<= |>= |< |== |> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}

%%

int main(int argc,char **argv)

{

Lex  Tab Width: 8  Ln 11, Col 33  INS
```

```
[0-9]+ {if(!COMMENT) printf("\n\t%s is a NUMBER",yytext);} {if(!COMMENT) printf("\n\t");ECHO;printf("\n");}

( ECHO;

{if(!COMMENT)printf("\n\t%s is an ASSIGNMENT OPERATOR",yytext);}

<= |>= |< |== |> {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}

%%

int main(int argc,char **argv)

{

if (argc > 1)

{

FILE *file;

file = fopen(argv[1],"r"); if(!file)

{

printf("could not open %s \n",argv[1]); exit(0);

}

yyin = file;

}

yylex();

printf("\n\n");

return 0;

}

 int yywrap()

{

return 0;

}SS
```

Output:-

```
l2sys29@l2sys29-Veriton-M275:~/Desktop/syedvirus$ cc lex.yy.c
l2sys29@l2sys29-Veriton-M275:~/Desktop/syedvirus$ ./a.out

#include<stdio.h> is a preprocessor directive

#include<conio.h> is a preprocessor directive

        void is a keyword
FUNCTION
        main(
        )


 BLOCK BEGINS

        int is a keyword
 a IDENTIFIER,
 b IDENTIFIER,
 c IDENTIFIER;

 a IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 1 is a NUMBER ;

 b IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 2 is a NUMBER ;

 c IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 a IDENTIFIER+
 b IDENTIFIER;

FUNCTION
        printf(
         "Sum:%d" is a STRING,
 c IDENTIFIER
        )
```

Conclusion:-

Thus , the program to implement lexical analyzer using JLex tool, flex etc. was successfully executed.

EXPERIMENT - 8

Date : 04/04/2022

Write a C program to construct a recursive descent parser .

Objective :-

Construction of a recursive descent parser

Resource :-

Atom editor with GCC Compiler

Program Logic :-

1. Read the input string.
2. Write procedures for the non terminals
3. Verify the next token equals to non terminals if it satisfies the non terminal.
4. If the input string does not match print error.

Procedure :-

- Go to debug-> run or press CTRL+F5 to run the program.

Program :-

```
#include<stdio.h>

#include<string.h>

int E();

int T();

int EP();

int F();

int TP();

char input[100];

int i,l;


int main()
```

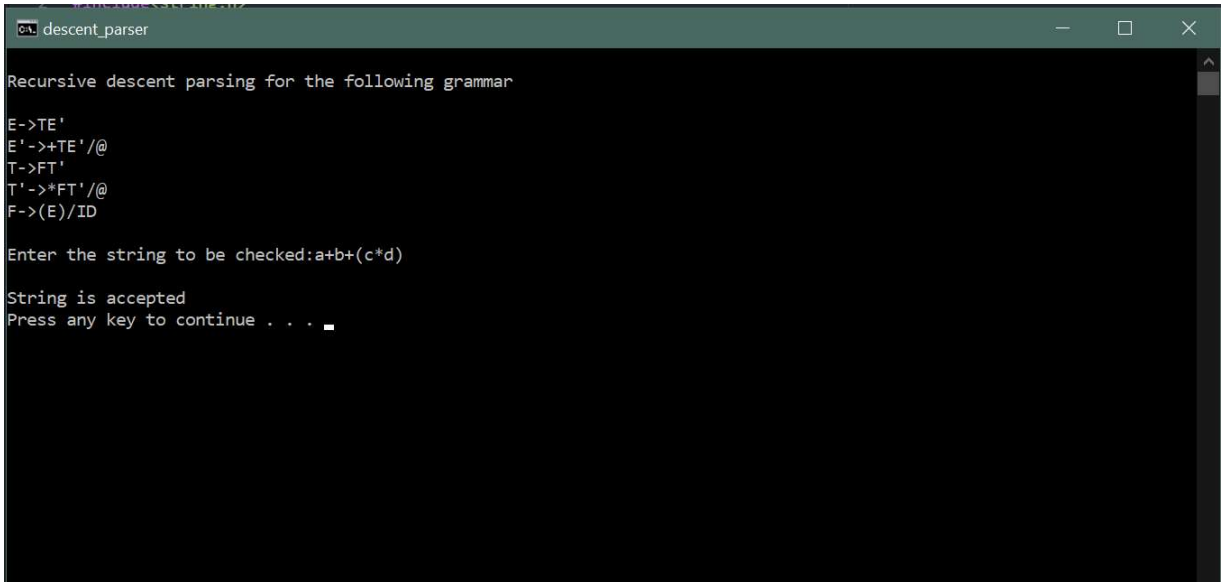
```
{
printf("\nRecursive descent parsing for the following grammar\n");
printf("\nE->TE'\nE'->+TE'/@\nT->FT'\nT'->*FT'/@\nF->(E)/ID\n");
printf("\nEnter the string to be checked:");
scanf("%s",&input);

if(E())
{
if(input[i+1]=='\0')
printf("\nString is accepted");
else
printf("\nString is not accepted");
}
else
printf("\nString not accepted");
}
int E()
{
if(T())
{
if(EP())
return(1);
else
return(0);
}
else
return(0);
}
int EP()
{
if(input[i]=='+')
```

```
{
i++;
if(T())
{
if(EP())
return(1);
else
return(0);
}
else
return(0);
}
else
return(1);
}
int T()
{
if(F())
{
if(TP())
return(1);
else
return(0);
}
else
return(0);
}
int TP()
{
if(input[i]=='*')
{
```

```
i++;
if(F())
{
if(TP())
return(1);
else
return(0);
}
else
return(0);
}
else
return(1);
}
int F()
{
if(input[i]=='(')
{
i++;
if(E())
{
if(input[i]==')')
{
i++;
return(1);
}
else
return(0);
}
else
return(0);
```

```
}
else if(input[i]>='a'&& input[i]<='z'||input[i]>='A'&& input[i]<='Z')
{
i++;
return(1);
}
else
return(0);
}
```

Output :-



Conclusion :-

Thus, the program to construct a recursive descent parser was successfully implemented.

EXPERIMENT - 9

Date : 04/04/2022

Write a C program to implement operator precedence parsing .

Objective :-

Implement operator precedence parsing

Resource :-

Atom editor with GCC Compiler

Program Logic :-

1. read the arithmetic input string
2. verify the precedence between terminals and symbols
3. find the handle enclosed in <.> and reduce it to production symbol
4. repeat the process till we reach the start node

Procedure :-

- Go to debug-> run or press CTRL+F5 to run the program.

Program :-

```
#include<stdio.h>


#include<conio.h>


void main() {
  char stack[20], ip[20], opt[10][10][1], ter[10];
  int i, j, k, n, top = 0, col, row;
  for (i = 0; i < 10; i++) {
    stack[i] = NULL;
    ip[i] = NULL;
    for (j = 0; j < 10; j++) {
      opt[i][j][1] = NULL;
```

```
  }
}
printf("Enter the no.of terminals :\n");
scanf("%d", & n);
printf("\nEnter the terminals :\n");
scanf("%s", & ter);
printf("\nEnter the table values :\n");
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    printf("Enter the value for %c %c:", ter[i], ter[j]);
    scanf("%s", opt[i][j]);
  }
}
printf("\n**** OPERATOR PRECEDENCE TABLE ****\n");
for (i = 0; i < n; i++) {
  printf("\t%c", ter[i]);
}
printf("\n");
for (i = 0; i < n; i++) {
  printf("\n%c", ter[i]);
  for (j = 0; j < n; j++) {
    printf("\t%c", opt[i][j][0]);
  }
}
stack[top] = '$';
printf("\nEnter the input string:");
scanf("%s", ip);
i = 0;
printf("\nSTACK\t\t\tINPUT STRING\t\t\tACTION\n");
```

```
printf("\n%s\t\t\t%s\t\t\t", stack, ip);
while (i <= strlen(ip)) {
  for (k = 0; k < n; k++) {
    if (stack[top] == ter[k])
      col = k;
    if (ip[i] == ter[k])
      row = k;
  }
  if ((stack[top] == '$') && (ip[i] == '$')) {
    printf("String is accepted\n");
    break;
  } else if ((opt[col][row][0] == '<') || (opt[col][row][0] == '=')) {
    stack[++top] = opt[col][row][0];
    stack[++top] = ip[i];
    printf("Shift %c", ip[i]);
    i++;
  } else {
    if (opt[col][row][0] == '>') {
      while (stack[top] != '<') {
        --top;
      }
      top = top - 1;
      printf("Reduce");
    } else {
      printf("\nString is not accepted");
      break;
    }
  }
  printf("\n");
```

```
    for (k = 0; k <= top; k++) {
      printf("%c", stack[k]);
    }
    printf("\t\t\t");
    for (k = i; k < strlen(ip); k++) {
      printf("%c", ip[k]);
    }
    printf("\t\t\t");
  }
  getch();
}
```

Output:-

```
parsing
Enter the no.of terminals :
4

Enter the terminals :
+*i$

Enter the table values :
Enter the value for + +:>
Enter the value for + *:<
Enter the value for + i:<
Enter the value for + $:>
Enter the value for * +:>
Enter the value for * *:>
Enter the value for * i:<
Enter the value for * $:>
Enter the value for i +:>
Enter the value for i *:>
Enter the value for i i:=
Enter the value for i $:>
Enter the value for $ +:<
Enter the value for $ *:<
Enter the value for $ i:<
Enter the value for $ $:A

**** OPERATOR PRECEDENCE TABLE ****
        +       *       i       $

+       >       <       <       >
*       >       >       <       >
i       >       >       =       >
$       <       <       <       A
Enter the input string:i+i*i$
```

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

STACK                   INPUT STRING                    ACTION

$                       i+i*i$                  Shift i
$<i                     +i*i$                   Reduce
$                       +i*i$                   Shift +
$<+                     i*i$                    Shift i
$<+<i                   *i$                     Reduce
$<+                     *i$                     Shift *
$<+<*                   i$                      Shift i
$<+<*<i                 $                       Reduce
$<+<*                   $                       Reduce
$<+                     $                       Reduce
$                       $                       String is accepted
```

Conclusion :-

Thus, the program to implement operator precedence parsing was successful .

EXPERIMENT - 10

Date : 11/04/2022

Write a C program to design LALR Bottom up parser .

Objective :-

Design LALR Bottom up parser

Resource :-

Atom editor with GCC

Program Logic :-

1. Read the input string
2. Push the input symbol with its state symbols in to the stack by referring lookaheads
3. We perform shift the reduce actions to parse the grammar
4. Parsing is completed when the $ symbol is reached.

Procedure :-

- Go to debug-> run or press CTRL+F5 to run the program.

Program :-

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
void push(char *,int *,char);
char stacktop(char *);
void isproduct(char,char);
int ister(char);
int isnter(char);
int isstate(char);
void error();
```

```
void isreduce(char,char);
char pop(char *,int *);
void printt(char *,int *,char [],int);
void rep(char [],int);
struct action
{
char row[6][5];
};
const struct action A[12]={
{"sf","emp","emp","se","emp","emp"},
{"emp","sg","emp","emp","emp","acc"},
{"emp","rc","sh","emp","rc","rc"},
{"emp","re","re","emp","re","re"},
{"sf","emp","emp","se","emp","emp"},
{"emp","rg","rg","emp","rg","rg"},
{"sf","emp","emp","se","emp","emp"},
{"sf","emp","emp","se","emp","emp"},
{"emp","sg","emp","emp","sl","emp"},
{"emp","rb","sh","emp","rb","rb"},
{"emp","rb","rd","emp","rd","rd"},
{"emp","rf","rf","emp","rf","rf"}
};
struct gotol
{
char r[3][4];
};
const struct gotol G[12]={
```

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

```
{'F',"(E)"},
{'F',"i"},
};
void main()
{
char inp[80],x,p,dl[80],y,bl='a';
int i=0,j,k,l,n,m,c,len;
printf(" Enter the input :");
scanf("%s",inp);
len=strlen(inp);
inp[len]='$';
inp[len+1]='\0';
push(stack,&top,bl);
printf("\n stack \t\t\t input");
printt(stack,&top,inp,i);
do
{
x=inp[i];
p=stacktop(stack);
isproduct(x,p);
if(strcmp(temp,"emp")==0)
error();
if(strcmp(temp,"acc")==0)
break;
else
{
if(temp[0]=='s')
```

```
{
push(stack,&top,inp[i]);
push(stack,&top,temp[1]);
i++;
}
else
{
if(temp[0]=='r')
{
j=isstate(temp[1]);
strcpy(temp,rl[j-2].right);
dl[0]=rl[j-2].left;
dl[1]='\0';
n=strlen(temp);
for(k=0;k<2*n;k++)
pop(stack,&top);
for(m=0;dl[m]!='\0';m++)
push(stack,&top,dl[m]);
l=top;
y=stack[l-1];
isreduce(y,dl[0]);
for(m=0;temp[m]!='\0';m++)
push(stack,&top,temp[m]);
}
}
}
printt(stack,&top,inp,i);
```

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

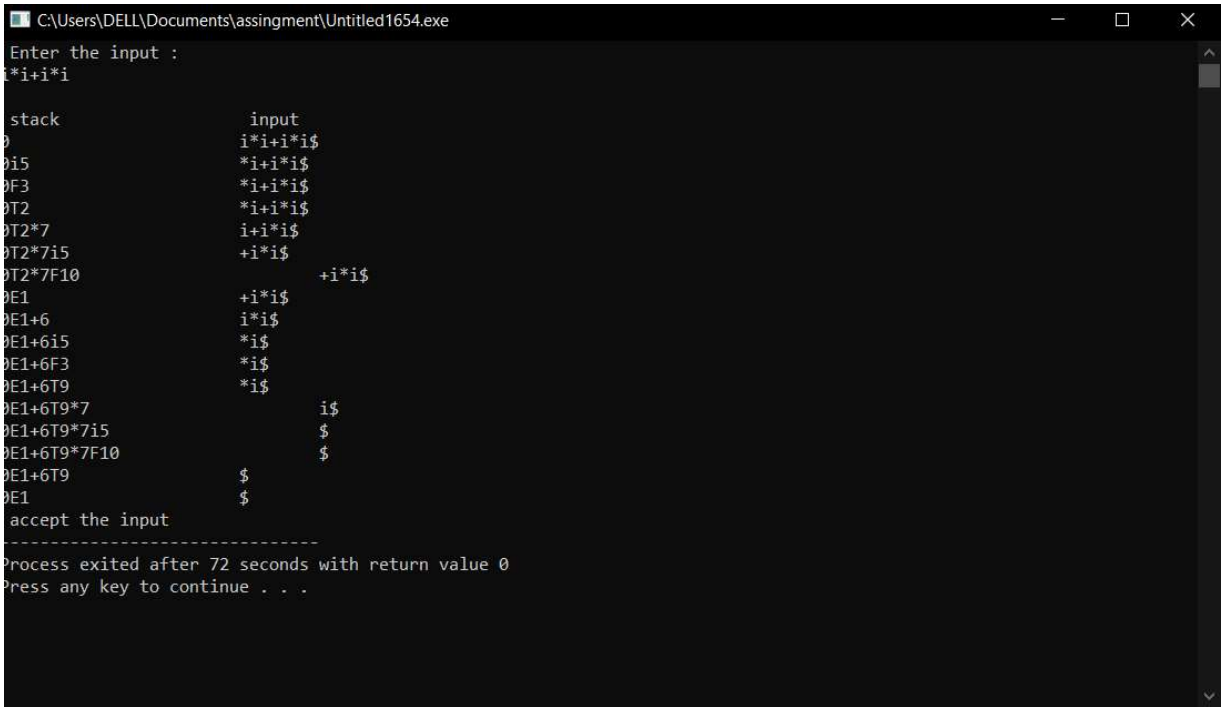
```
l=isstate(p);
strcpy(temp,A[l-1].row[k-1]);
}
int ister(char x)
{
int i;
for(i=0;i<6;i++)
if(x==ter[i])
return i+1;
return 0;
}
int isnter(char x)
{
int i;
for(i=0;i<3;i++)
if(x==nter[i])
return i+1;
return 0;
}
int isstate(char p)
{
int i;
for(i=0;i<12;i++)
if(p==states[i])
return i+1;
return 0;
}
```

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

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

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

Output:-



Conclusion:-

Thus , the program to implement the LALR Bottom up parser was successful .