Experiment 1:

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

Algorithm:

1. Read the input string.
2. Check whether the string is starting with '/' and check next character is '/' or '*'.
3. If condition satisfies, then print comment.
4. Else not a comment.

Source Code/Program:

```
#include<stdio.h>
#include<string.h>
int main(){
    char com[30];
    int len;
    printf("\nenter comment : ");
    gets(com);
    len = strlen(com);
    if(com[0] == '/' && com[1] == '/'){
        printf("it is a single line comment");
    }
    else if(com[0] == '/' && com[1] == '*' && com[len - 1] == '/' && com[len - 2] ==
'*' ){
        printf("it is a multi line comment");
    }
    else{
        printf("it is not a comment");
    }
}
```

Experiment 2:

2. **Write a C program to design a lexical analyser for given language, which should ignore the redundant spaces, tabs, new lines, find the tokens and also count the number of lines using C program**

Procedure:

- It is the first phase of the compiler. It gets input from the source program and produces tokens as output.
- It reads the characters one by one, starting from left to right and forms the tokens.

Token: It represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.

Example: a+b=20

Here, a,b,+,=,20 are all separate tokens.

Group of characters forming a token is called the Lexeme.

The lexical analyzer not only generates a token but also enters the lexeme into the symbol table if it is not already there. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Logic/Algorithm:

1. Read the C program as input and stores in a file.
2. Check all the characters from the file from left to right whether character is alphabet or digit or special symbol.
3. If the input is operator prints as special symbol.
4. If the input is number prints as number.
5. If the input is identifier prints as identifier.
6. If the input is keyword prints as keyword.
7. Print no lines of code

Source Code/Program Code:

```
#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||strc
mp("float",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||strcmp("static",str)==0||st
rcmp("switch",str)==0||strcmp("case",str)==0)
printf("\n%s is a keyword",str);
```

```
else
printf("\n%s is an identifier",str);
}

int main()
{
FILE *f1,*f2,*f3;
char c, str[10];
int num[100], lineno=0, tokenvalue=0,i=0,j=0,k=0;
printf("\n Enter 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("\n The 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 identifiers are:");
while((c=getc(f2))!=EOF)
{
if(c!=' ')
str[k++]=c;
else
{
str[k]='\0';
keyword(str);
k=0;
}
}
fclose(f2);
f3=fopen("specialchar","r");
printf("\n Special characters are : ");
while((c=getc(f3))!=EOF)
printf("%c",c);
printf("\n");
fclose(f3);
printf("Total no. of lines are:%d", lineno);
}
```

Experiment 3:

Write a C program to design a lexical analyzer for given language, which should ignore the redundant spaces, tabs, new lines, find the tokens and also count the number of lines using lex tool.

Procedure:

Contents of a lex program:

Declarations
%%
Translation rules
%%
Auxiliary functions

1. The declarations section can contain declarations of variables, manifest constants, and regular definitions. The declarations section can be empty.
2. The translation rules are each of the form pattern {action}
   - Each pattern is a regular expression which may use regular definitions defined in the declarations section.
   - Each action is a fragment of C-code.
3. The auxiliary functions section starting with the second %% is optional. Everything in this section is copied directly to the file lex.yy.c and can be used in the actions of the translation rules.

Source Code/Program Code

```
%{
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
#.* {printf ("\n %s is a Preprocessor Directive",yytext);}
int |
float |
main |
if |
else |
printf |
scanf |
for |
char |
getch |
while {printf("\n %s is a Keyword",yytext);}
"/*" {COMMENT=1;}
"*/" {COMMENT=0;}
```

```
{identifier}\( {if(!COMMENT) printf("\n Function:\t %s",yytext);}
\{ {if(!COMMENT) printf("\n Block Begins");
\} {if(!COMMENT) printf("\n Block Ends");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s is an Identifier",yytext);}
\".*\" {if(!COMMENT) printf("\n %s is a String",yytext);}
[0-9]+ {if(!COMMENT) printf("\n %s is a Number",yytext);}
\)(\;)? {if(!COMMENT) printf("\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT) printf("\n%s is an Assmt oprtr",yytext);}
\<= |
\>= |
\< |
== {if(!COMMENT) printf("\n %s is a Rel. Operator",yytext);}
.|\n
%%
int main(int argc, char **argv)
{
if(argc>1)
{
FILE *file;
file=fopen(argv[1],"r");
if(!file)
{
printf("\n Could not open the file: %s",argv[1]);
exit(0);
}
yyin=file;
}
yylex();
printf("\n\n");
return 0;
}
int yywrap()
{
return 0;
}
```

Output:

```
$ lex lexp.l
$ cc lex.yy.c
$ ./a.out test.c

#include is a Preprocessor Directive
Function: main( ) Block Begins
int is a Keyword
```

fact is an Identifier
= is an Assignment Operator
1 is a Number
n is an Identifier
Function: for(
int is a Keyword
i is an Identifier
= is an Assignment Operator
1 is a Number
i is an Identifier
<= is a Relational Operator
n is an Identifier
i is an Identifier
); Block Begins
fact is an Identifier
= is an Assignment Operator
fact is an Identifier
i is an Identifier
Block Ends
Function: printf("Factorial Value of N is" is a String
fact is an Identifier);
Function: getch( );
Block Ends

Experiment 4:

Write a C program to recognize strings under 'a*|abb'

Procedure:

1. By using finite automata for the given regular expression, we can verify the given input is accepted or not?
2. If the state recognizes the given pattern rule. Then print string is accepted under a*|abb.
3. Else print string is not accepted.

Source Code/Program Code:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main()
{
char s[20],c;
int state=0, i=0;
printf("\n Enter a string:");
gets(s);
while(s[i]!='\0')
{
switch(state)
{
case 0:
c=s[i++];
if(c=='a')
state=1;
else
state=4;
break;
case 1:
c=s[i++];
if(c=='a')
state=1;
else
if(c=='b')
state=2;
else
state=4;
break;
case 2:
```

```
c=s[i++];
if(c=='b')
state=3;
else
state=4;
break;
case 3:
if((c=s[i++])!='\0');
state=4;
break;
case 4:
printf("\n %s is not recognised.",s);
exit(0);
}
}
if(state==1)
printf("\n %s is accepted under rule 'a'",s);
else
if(state==3)
printf("\n %s is accepted under rule 'abb'",s);
else
printf("\n %s is not accepted",s);
}
```

Experiment 5:

Write a C program to construct a recursive descent parser for an expression.

Procedure:

Recursive Descent Parser uses the technique of Top-Down Parsing without backtracking. It can be defined as a Parser that uses the various recursive procedure to process the input string with no backtracking

Example – Write down the algorithm using Recursive procedures to implement the following Grammar.

$E \rightarrow TE'$ $E' \rightarrow +TE'$ $T \rightarrow FT'$ $T' \rightarrow * FT'|\varepsilon$ $F \rightarrow (E)|id$

Solution

```
Procedure E ( )
{
        T ( );                          E → TE′
        E′( );
}
Procedure E′( )
{
        If input symbol ='+' then        E → + TE′
        advance( );
        T ( );
        E′( );
}
```

```
Procedure T( )
{
        F ( );                           T → F T′
        T′( );
}
Procedure T′( )
{
        If input symbol ='*' then        T′ →* FT′
        advance( );
        F ( );
        T′( );
}
```

```
Procedure F ( )
{
        If input symbol ='id' then                  F → id
        advance ( );
        else if input-symbol ='(' then
        advance ( );
        E ( );
        If input-symbol = ')'                       F → (E)
        advance ( );
        else error ( );
        else error ( );
}
```

One of major drawback:
recursive-descent parsing is that it can be implemented only for those languages which support recursive procedure calls and it suffers from the problem of left-recursion.

Source Code/Program Code:

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

char input[10];
int i,error;
void E();
void T();
void Eprime();
void Tprime();
void F();
        int main()
        {
i=0;
error=0;
            printf("Enter an arithmetic expression   :  "); // Eg: a+a*a
            gets(input);
            E();
            if(strlen(input)==i&&error==0)
                printf("\nAccepted..!!!\n");
            else printf("\nRejected..!!!\n");
                  }
```

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

        else error=1;
         }

        else error=1;
        }
```

Experiment 6:

Write a C program to simulate FIRST of a given Context Free Grammar.

Procedure:

Rules for FIRST ():

1. If X is terminal, then FIRST(X) is {X}.

2. If X → ε is a production, then add ε to FIRST(X).

3. If X is non-terminal and X → aα is a production then add a to FIRST(X).

4. If X is non-terminal and X → Y 1 Y2… Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1),… ,FIRST(Yi-1); that is, Y1,… .Yi-1 => ε. If ε is in FIRST(Yj) for all j=1,2,..,k, then add ε to FIRST(X).

Source Code/Program Code: