Experiment – 1 Lexical analysis using lex tool

1.1) Write a lex program whose output is same as input.

Program:

```
%%
. ECHO;
%%
int yywrap(void) {
return 1;
}
int main(void) {
yylex();
return 0;
}
```

Output:

```
[20A91A0523@Linux ~]$ vi lex1.l
[20A91A0523@Linux ~]$ flex lex1.l
[20A91A0523@Linux ~]$ gcc lex.yy.c -ll
[20A91A0523@Linux ~]$ ./a.out
abdefa
abdefa
```

1.2) Write a lex program which removes comments from its input file

Program:

```
%%
/*regular expression for single line*/
\/\/(.*)\n {};
/*regular expr multiple line comment */
\/\*(.*\n)*.*\*\/ {};
. ECHO;
%%
```

Output: case i

```
#include<stdio.h>
int main()
{
 printf("Hello World");
 //Removin comments program
 return 0;
}
```

Output:

```
[20A91A0523@Linux ~]$ vi lex1.l
[20A91A0523@Linux ~]$ flex lex1.l
[20A91A0523@Linux ~]$ gcc lex.yy.c -ll
[20A91A0523@Linux ~]$ ./a.out
#include<stdio.h>
int main()
{
  printf("Hello World");
    return 0;
}
```

Experiment – 2 Lexical analysis using lex tool

2.1) Write a lex program to identify the patterns in the input file.

Program:

```
%{
#include<stdio.h>
%}
%%
["int""char""for""if""while""then""return""do"]
{printf("keyword :
%s\n");}
[*%+\-] {printf("Operator : %s ", yytext);}
[(){};] {printf("Special Character: %s\n", yytext);}
[0-9]+ {printf("Constant : %s\n", yytext);}
[a-zA-Z_][a-zA-Z0-9_]* {printf("Valid Identifier is : %s\n",
yytext);}
^[^a-zA-Z_] {printf("Invalid Indentifier \n");}
%%
```

Output:

```
[20A91A0523@Linux ~]$ flex Pattern.l
[20A91A0523@Linux ~]$ gcclex.yy.c -ll
[20A91A0523@Linux ~]$ ./a.out<Hello.c
Invalid Indentifier
Valid Identifier is : include
<Valid Identifier is :stdio
.keyword :
>
Valid Identifier is :int
Valid Identifier is : main
Special Character: (
Special Character: )
Special Character: {
Special Character: }
```

2.2) Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs and new lines.

Program:

```
%{
#include<stdio.h>
int i=0,id=0;
%}
%%
[#].*[<].*[>]\n {}
[ \t\n]+ {}
\/\/.*\n {}
\/\*(.*\n)*.*\*\/ {}
auto|break|case|char|const|continue|default|do|double|else|enu
m|extern|float|for|goto|if|int|long|register|return|short|sign
ed|sizeof|static|struct|switch|typedef|union|unsigned|void|vol
a
tile|while {
printf("token : %d < keyword , %s >\n",++i,yytext);
}
[+\-\*\/%<>]{
printf("token: %d < operator , %s >\n",++i,yytext);
}
[();{}]{
printf("token : %d < special char , %s >\n",++i,yytext);
}
[0-9]+ {
printf("token : %d < constant , %s >\n",++i,yytext);
}
[a-zA-Z_]
[a-zA-Z0-9_]*{
printf("token:%d<Id%d ,%s >\n",++i,++id,yytext);
}
^[^a-zA-Z_] {
printf("ERROR Invaild token %s \n",yytext);
}%%
```

Output case i with input file:

```
#include<stdio.h>
int main()
{
 printf("Hello World");
 //Removin comments program
 return 0;
}
```

Output:

```
[20A91A0523@Linux ~]$ flex infix.l
[20A91A0523@Linux ~]$ yacc -d infix.l
[20A91A0523@Linux ~]$ gcc lex.yy.c -ll
[20A91A0523@Linux ~]$ ./a.out<sample.c
token: 1 < keyword , int >
token : 2 < Idl,main >
token : 3 < special char ( >
token: 4 < special char ) >
token: 5 < special char { >
token: 6 < Id2,printf >
token: 7 < special char
"token: 8 < Id3,Hello >
token 9 Id4,World >
"token: 10 < special char,) >
token: 11 < special char
token: 12 < keyword, return >
token :13 < constant, 0 >
token: 14 < special char ; >
token: 15 < special char } >
```

Experiment – 3 First and Follow

3.1) Simulate First and Follow of a Grammar.

Program:

```
#include<stdio.h>
#include<string.h>
int n, m = 0, p, i = 0, j = 0;
char a[10][10], f[10];
void follow(char c);
void first(char c);
int main(){
int i, z;
char c, ch;
printf("enter the no. of productions:");
scanf("%d", &n);
printf("enter the productions(epsilon = $):\n");
for(i = 0; i < n; i++)
scanf("%s%c", a[i], &ch);
do{
m = 0;
printf("enter the element 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");
follow(c);
printf("FOLLOW(%c) = {", c);
for(;i<m;i++)
printf("%c", f[i]);
printf("}\n");
printf("do you want to continue(0/1)?");
scanf("%d%c", &z, &ch);
 }
```

```
while(z = 1);
}
void follow(char c){
if(a[0][0] = c)
f[m++] = '$';
for(i = 0;i<n;i++){
for(j = 2;j<strlen(a[i]);j++){
if(a[i][j] = c){
if(a[i][j+1] != '\0')first(a[i][j+1]);
if(a[i][j+1] = '\0'&&c != a[i][0])
follow(a[i][0]);
}}}}
void first(char c){
int k;
if(!(isupper(c)))
f[m++] = c;
for(k = 0;k<n;k++){
if(a[k][0] = c){
if(a[k][2] = '$')
follow(a[i][0]);
else if(islower(a[k][2]))
f[m++] = a[k][2];
else
first(a[k][2]);
}}}
```

Output:

```
C:\Users\admin\Documents\exp3a.exe
Enter the number of productions: 3
Enter the productions(Epsilon=$):
S=aSa
S=bSb
S=$
Enter the element to calculate First and Follow: S
First(S) = { a  b  $  a  b }
Follow(S) = { $  a  b }
Do you wish to continue(0/1)?: 1
Enter the element to calculate First and Follow: a
First(a) = { a }
Follow(a) = { a  b  $  a  b }
Do you wish to continue(0/1)?: 1
Enter the element to calculate First and Follow: b
First(b) = { b }
Follow(b) = { a  b  $  a  b }
Do you wish to continue(0/1)?: 0

--------------------------------
Process exited after 32.33 seconds with return value 0
Press any key to continue . . .
```

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

Program:

```
%{
#include<stdio.h>
int i=0,id=0;
%}%%
[#].*[<].*[>]\n {}
[ \t\n]+ {}
\/\/.*\n {}
\/\*(.*\n)*.*\*\/ {}
auto|break|case|char|const|continue|default|do|double|else|enu
m|extern|float|for|goto|if|int|long|register|return|short|sign
ed|sizeof|static|struct|switch|typedef|union|unsigned|void|vol
atile|
while{
printf("token : %d < keyword , %s >\n",++i,yytext);
}
[+\-\*\/%<>] {
printf("token : %d < operator , %s >\n",++i,yytext);
}
[();{}] {
printf("token : %d < special char , %s >\n",++i,yytext);
}
[0-9]+
{
printf("token : %d < constant , %s >\n",++i,yytext);
}
[a-zA-Z_][a-zA-Z0-9_]*
{
printf("token : %d < Id%d ,%s >\n",++i,++id,yytext);
}
^[^a-zA-Z_]{
printf("ERROR Invaild token %s \n",yytext);
}
%%
```

Output:

```
[20A91A0523@Linux ~]$ vi lex1.l
[20A91A0523@Linux ~]$ flex lex1.l
[20A91A0523@Linux ~]$ gcc lex.yy.c -ll
[20A91A0523@Linux ~]$ ./a.out
 token : 1 < keyword  , int >
 token : 2 < Id1 ,main >
 token : 3 < special char , ( >
 token : 4 < special char , ) >
 token : 5 < special char , { >
 token : 6 < Id2 ,printf >
 token : 7 < special char , ( >
 "token : 8 < Id3 ,Hello >
 token : 9 < Id4 ,World >
 "token : 10 < special char , ) >
 token : 11 < special char , ; >
 token : 12 < keyword  , return >
 token : 13 < constant , 0 >
 token : 14 < special char , ; >
 token : 15 < special char , } >
```

Experiment – 4 Top Down Parsing

4.1) Develop an operator precedence parser for a given language.

Program:

```
#include<stdio.h>
#include<string.h>
char stack[20],temp;
int top=-1;
void push(char item){
if(top>=20){
printf("STACK OVERFLOW");
return;
}
stack[++top]=item;
}
char pop(){
if(top<=-1){
printf("STACK UNDERFLOW");
return;
}
char c;
c=stack[top--];
printf("Popped element:%c\n",c);
return c;
}
char TOS(){
return stack[top];
}
int convert(char item){
switch(item){
case 'i':return 0;
case '+':return 1;
case '*':return 2;
case '$':return 3;
}}
int main(){
char pt[4][4]={
```

```
{'-','>','>','>'},
{'<','>','<','>'},
{'<','>','>','>'},
{'<','<','<','1'}};
char input[20];
int lkh=0;
printf("Enter input with $ at the end\n");
scanf("%s",input);
push('$');
while(lkh<=strlen(input)){
if(TOS()=='$'&&input[lkh]=='$'){
printf("SUCCESS\n");
return 1;
}
else if(pt[convert(TOS())][convert(input[lkh])]=='<'){
push(input[lkh]);
printf("Push---%c\n",input[lkh]);
lkh++;
}
else{
pop();
}}
return 0;
}
```

Output:

```
C:\Users\admin\Documents\cd4b.exe
Enter input with $ at the end
i*i+i*i+i$
Push---i
Popped element:i
Push---*
Push---i
Popped element:i
Popped element:*
Push---+
Push---i
Popped element:i
Push---*
Push---i
Popped element:i
Popped element:*
Popped element:+
Push---+
Push---i
Popped element:i
Popped element:+
SUCCESS

--------------------------------
Process exited after 38.64 seconds with return value 1
Press any key to continue . . .
```

4.2) Construct a recursive descent parser for an expression.

Program:

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
void Tp();
void Ep();
void E();
void T();
void check();
int count,flag;
char expr[10];
int main(){
count=0;
flag=0;
printf("\nEnter an Algebraic Expression:\t");
scanf("%s",expr);
E();
if((strlen(expr)==count)&&(flag==0))
printf("\nThe expression %s is valid\n",expr);
else
printf("\nThe expression %s is invalid\n",expr);
return 0;
}
void E(){
T();
Ep();
}
void T(){
check();
Tp();
}
void Tp(){
if(expr[count]=='*')
{
count++;
```

```
check();
Tp();
}}
void check(){
if(isalnum(expr[count]))
count++;
else if(expr[count]=='(') {
count++;
E();
if(expr[count]==')')
count++;
else
flag=1;
}
else
flag=1;
}
void Ep(){
if(expr[count]=='+'){
count++;
T();
Ep();
}}
```

Output:

```
[20A91A0523@Linux ~]$ vi bit.c
[20A91A0523@Linux ~]$ cc bit.c
[20A91A0523@Linux ~]$ ./a.out

Enter an Algebraic Expression:  (7+8)*5

The expression (7+8)*5 is valid
[20A91A0521@Linux ~]$ ./a.out

Enter an Algebraic Expression:  (8*))9

The expression (8*))9 is invalid
```

Experiment – 5 Bottom up Parsing

5.1) Construct a LL(1) parser for an expression

Program:

```
#include<stdio.h>
#include<string.h>
int stack[20],top=-1;
void push(int item){
if(top>=20){
printf("stack overflow");
return;
}
stack[++top]=item;
}
int pop(){
int ch;
if(top<=-1){
printf("underflow");
return;
}
ch=stack[top--];
return ch;
}
char convert(int item){
char ch;
switch(item){
case 0:return('E');
case 1:return('e');
case 2:return('T');
case 3:return('t');
case 4:return('F');
case 5:return('i');
case 6:return('+');
case 7:return('*');
case 8:return('(');
case 9:return(')');
case 10:return('$');
}}
void main(){
int m[10][10],i,j,k;
char ips[20];
```

```
int ip[10],a,b,t;
m[0][0]=m[0][3]=21;
m[1][1]=621;
m[1][4]=m[1][5]=-2;
m[2][0]=m[2][3]=43;
m[3][1]=m[3][4]=m[3][5]=-2;
m[3][2]=743;
m[4][0]=5;
m[4][3]=809;
printf("\nenter the input string with $ at ehe end:(EX: i+i*i$)\n");
scanf("%s",ips);
for(i=0;i<strlen(ips);i++){
switch(ips[i]){
case 'E':k=0;break;
case 'e':k=1;break;
case 'T':k=2;break;
case 't':k=3;break;
case 'F':k=4;break;
case 'i':k=5;break;
case '+':k=6;break;
case '*':k=7;break;
case '(':k=8;break;
case ')':k=9;break;
case '$':k=10;break;
}
ip[i]=k;
}
ip[i]=-1;
push(10);
push(0);
i=0;
printf("\tstack\t\t input \n");
while(1){
printf("\t");
for(j=0;j<=top;j++)
printf("%c",convert(stack[j]));
printf("\t\t");
for(k=i;ip[k]!=-1;k++)
printf("%c",convert(ip[k]));
printf("\n");
if(stack[top]==ip[i]){
```

```
if(ip[i]==10){
printf("\t\t success\n");
return;
}
else{
top--;
i++;
}}
else if(stack[top]<=4&&stack[top]>=0){
a=stack[top];
b=ip[i]-5;
t=m[a][b];
top--;
while(t>0){
push(t%10);
t=t/10;
}}
else{
printf("error\n");
return;
}}}
```

Output:

```
[20A91A0523@Linux ~]$ vi bit.c
[20A91A0523@Linux ~]$ cc bit.c
[20A91A0523@Linux ~]$ ./a.out

enter the input string with $ at ehe end:(EX: i+i*i$)
i*i$
        stack            input
        $E               i*i$
        $eT              i*i$
        $etF             i*i$
        $eti             i*i$
        $et              *i$
        $etF*            *i$
        $etF             i$
        $eti             i$
        $et              $
        $e               $
        $                $
                success
```

5.2) Design a LALR bottom up parser for the given language.

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void push(char *,int *,char);
char stacktop(char *);
void isproduct(char,char);
int ister(char);
int isnter(char);
int isstate(char);
void error();
void isreduce(char,char);
char pop(char *,int *);
void printt(char *,int *,char [],int);
void rep(char [],int);
struct action{
char row[6][5];
};
const struct action A[12]={
{"sf","emp","emp","se","emp","emp"},
{"emp","sg","emp","emp","emp","acc"},
{"emp","rc","sh","emp","rc","rc"},
{"emp","re","re","emp","re","re"},
{"sf","emp","emp","se","emp","emp"},
{"emp","rg","rg","emp","rg","rg"},
{"sf","emp","emp","se","emp","emp"},
{"sf","emp","emp","se","emp","emp"},
{"emp","sg","emp","emp","sl","emp"},
{"emp","rb","sh","emp","rb","rb"},
{"emp","rb","rd","emp","rd","rd"},
{"emp","rf","rf","emp","rf","rf"}};
struct gotol{
char r[3][4];
};
const struct gotol G[12]={
{"b","c","d"},
{"emp","emp","emp"},
{"emp","emp","emp"},
{"emp","emp","emp"},
{"i","c","d"},
{"emp","emp","emp"},
{"emp","j","d"},
```

```
{"emp","emp","k"},
{"emp","emp","emp"},
{"emp","emp","emp"},};
char ter[6]={'i','+','*',')','(','$'};
char nter[3]={'E','T','F'};
char states[12]={'a','b','c','d','e','f','g','h','m','j','k','l'};
char stack[100];
int top=-1;
char temp[10];
struct grammar{
char left;
char right[5];
};
const struct grammar rl[6]={
{'E',"e+T"},
{'E',"T"},
{'T',"T*F"},
{'T',"F"},
{'F',"(E)"},
{'F',"i"},};
void main(){
char inp[80],x,p,dl[80],y,bl='a';
int i=0,j,k,l,n,m,c,len;
printf(" Enter the input :");
scanf("%s",inp);
len=strlen(inp);
inp[len]='$';
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 ");
}
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:

```
[20A91A0523@Linux ~]$ vi bit.c
[20A91A0523@Linux ~]$ cc bit.c
[20A91A0523@Linux ~]$ ./a.out
 Enter the input :i*i+i*i

 stack                          input
0                              i*i+i*i$
0i5                            *i+i*i$
0F3                            *i+i*i$
0T2                            *i+i*i$
0T2*7                          i+i*i$
0T2*7i5                        +i*i$
0T2*7F10                                +i*i$
0E1                            +i*i$
0E1+6                          i*i$
0E1+6i5                        *i$
0E1+6F3                        *i$
0E1+6T9                        *i$
0E1+6T9*7                               i$
0E1+6T9*7i5                             $
0E1+6T9*7F10                            $
0E1+6T9                        $
0E1                            $
```

Experiment – 6 Optimization Phase

6.1) Write a program to perform loop unrolling.

Program:

```
#include<stdio.h>
void main() {
unsigned int n;
int x;
char ch;
printf("\nEnter N\n");
scanf("%u", & n);
printf("\n1. Loop Roll\n2. Loop UnRoll\n");
printf("\nEnter ur choice\n");
scanf(" %c", & ch);
switch (ch) {
case '1':
x = countbit1(n);
printf("\nLoop Roll: Count of 1's : %d", x);
break;
case '2':
x = countbit2(n);
printf("\nLoop UnRoll: Count of 1's : %d", x);
break;
default:
printf("\n Wrong Choice\n");
}}
int countbit1(unsigned int n) {
int bits = 0, i = 0;
while (n != 0) {
if (n & 1) bits++;
n >>= 1;
i++;
}
printf("\n no of iterations %d", i);
return bits;
}
int countbit2(unsigned int n) {
```

```
int bits = 0, i = 0;
while (n != 0) {
if (n & 1) bits++;
if (n & 2) bits++;
if (n & 4) bits++;
if (n & 8) bits++;
n >>= 4;
i++;
}
printf("\n no of iterations %d", i);
return bits;
}
```

Output:

```
[20A91A0523@Linux ~]$ vi bit.c
[20A91A0523@Linux ~]$ cc bit.c
[20A91A0523@Linux ~]$ ./a.out

Enter N
2

1. Loop Roll
2. Loop UnRoll

Enter ur choice
1

 no of iterations  2
Loop Roll: Count of  1's    :  1
```

6.2) Write a program for constant propagation

Program:

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
void input();
void output();
void change(int p,char *res);
void constant();
struct expr{
char op[2],op1[5],op2[5],res[5];
int flag;
}arr[10];
int n;
void main(){
input();
constant();
output();
}
void input(){
int i;
printf("\n\nEnter the maximum number of expressions : ");
scanf("%d",&n);
printf("\nEnter the input : \n");
for(i=0;i<n;i++){
scanf("%s",arr[i].op);
scanf("%s",arr[i].op1);
scanf("%s",arr[i].op2);
scanf("%s",arr[i].res);
arr[i].flag=0;
}}
void constant(){
int i;
int op1,op2,res;
char op,res1[5];
for(i=0;i<n;i++){
if(isdigit(arr[i].op1[0]) && isdigit(arr[i].op2[0]) || strcmp(arr[i].op,"=")==0){
op1=atoi(arr[i].op1);
```

```
op2=atoi(arr[i].op2);
op=arr[i].op[0];
switch(op){
case '+':
res=op1+op2;
break;
case '-':
res=op1-op2;
break;
case '*':
res=op1*op2;
break;
case '/':
res=op1/op2;
break;
case '=':
res=op1;
break;
}
sprintf(res1,"%d",res);
arr[i].flag=1;
change(i,res1);
}}}
void output(){
int i=0;
printf("\nOptimized code is : ");
for(i=0;i<n;i++){
if(!arr[i].flag){
printf("\n%s %s %s %s",arr[i].op,arr[i].op1,arr[i].op2,arr[i].res);
}}}
void change(int p,char *res){
int i;
for(i=p+1;i<n;i++){
if(strcmp(arr[p].res,arr[i].op1)==0)
strcpy(arr[i].op1,res);
else if(strcmp(arr[p].res,arr[i].op2)==0)
strcpy(arr[i].op2,res);
}}
```

Output:

```
[20A91A0523@Linux ~]$ vi bit.c
[20A91A0523@Linux ~]$ cc bit.c
[20A91A0523@Linux ~]$ ./a.out

Enter the maximum number of  expressions : 4

Enter the input :
= 3 - a
+ a b t1
+ a c t2
+ t1 t2 t3

Optimized code is :
+ 3 b t1
+ 3 c t2
+ t1 t2 t3
```