Practical-1

Aim: To implement lexical analyse to recognize all distinct token classes.

Code:

```
/*STANDALONE SCANNER PROGRAM*/
#include<stdio.h>
#include<ctype.h>
#include<string.h>
int main()
{ FILE *input, *output;
int l=1;  int t=0;  int j=0;  int i,flag;  char ch,str[20];  input =
fopen("input.txt","r");  output = fopen("output.txt","w");  char
keyword[30][30]  =  {"int","main","if","else","do","while"};
fprintf(output,"Line no. \t Token no. \t Token \t Lexeme\n\n");
while(!feof(input))
{ i=0; flag=0; ch=fgetc(input);
if( ch=='+' || ch== '-' || ch=='*' || ch=='/' )
{ fprintf(output,"%7d\t\t %7d\t\t Operator\t %7c\n",l,t,ch);  t++;
}
else if( ch==';' || ch=='{' || ch=='}' || ch=='(' || ch==')' || ch=='?' || ch=='@' || ch=='!' || ch=='%')
{ fprintf(output,"%7d\t\t %7d\t\t Special symbol\t %7c\n",l,t,ch);  t++;
}
else if(isdigit(ch))
{ fprintf(output,"%7d\t\t %7d\t\t Digit\t\t %7c\n",l,t,ch);  t++;
}
else if(isalpha(ch))
{ str[i]=ch;  i++;
ch=fgetc(input);
while(isalnum(ch) && ch!=' ')
{ str[i]=ch;  i++;
ch=fgetc(input); }  str[i]='\0';
for(j=0;j<=30;j++)
{ if(strcmp(str,keyword[j])==0)
{ flag=1;
break;  } }
if(flag==1
)
{ fprintf(output,"%7d\t\t %7d\t\t Keyword\t %7s\n",l,t,str);
t++;  }
```

```
else{ fprintf(output,"%7d\t\t %7d\t\t Identifier\t %7s\n",l,t,str);
t++;  }}
else if(ch=='\n')
{   l++;       }}
fclose(input);
fclose(output);
return 0;
}
```

Input:

```
//input.txt
#include<stdio.h>
void main()
{  printf(“Hello
World”);
}
```

Output:

```
//output.txt
Line no. Token no. Token Lexeme
1 0 Identifier include
1 1 Identifier stdio
1 2 Identifier h
2 3 Identifier void
2 4 Keyword main
2 5 Special symbol )
3 6 Special symbol {
4 7 Identifier printf
4 8 Identifier Hello
4 9 Identifier World
4 10 Special symbol )
4 11 Special symbol ;
5 12 Special symbol }
```

Practical-2

Aim:To implement a Recursive Descent Parser Algorithm for the grammar.

Code:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
char input[100];
int i,l;
int main()
{ printf("recursive decent parsing for the grammar");
printf("\n E->TEP|\nEP->+TEP|@|\nT->FTP|\nTP->*FTP|@|\nF->(E)|ID\n");
printf("enter the string to check:");
scanf("%s",input);
if(E()){  if(input[i]=='$')
printf("\n string is accepted\n");  else
printf("\n string is not accepted\n");
} }  E(){
if(T()){  if(EP())
return(1);  else
return(0); }  else
return(0); }
EP(){
if(input[i]=='+'){
i++;  if(T()){
if(EP())
return(1);  else
return(0);
}
else return(1);
} }
T(){if(F()){
if(TP())
return(1);  else
return(0);
}
else return(0);
printf("String is not accpeted\n");
}
```

```
TP(){
if(input[i]=='*'){
i++;  if(F()){
if(TP())
return(1);  else
return(0);
}  else
return(0);
printf("The string is not accepted\n");
}
else return(1);
}  F(){
if(input[i]=='('){
i++;  if(E()){
if(input[i]==')'){
i++;  return(1);
} else {return(0);
printf("String is not accepted\n");
} }  else
return(0);
}
else if(input[i]>='a'&&input[i]<='z'||input[i]>='A'&&input[i]<='Z')
{               i++;
return(1);        }
else return(0); }
```

OUTPUT:

```
[cse410@cc5 ~]$ cc rdp.c  [cse410@cc5
~]$ ./a.out
recursive decent parsing for the grammar
E->TEP|
EP->+TEP|@|
T->FTP|
TP->*FTP|@|  F->(E)|ID
enter the string to check:(i+i)*i  string
is accepted
```

Practical-3

Aim: To find the First () and Follow () of a grammar.

Code :

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

void followfirst(char, int, int); void
follow(char c);
void findfirst(char, int, int);

int count, n = 0; char
calc_first[10][100]; char
calc_follow[10][100]; int
m = 0; char
production[10][10]; char
f[10], first[10];
int k; char
ck;
int e;
int main(int argc, char** argv)
{
 int jm = 0;   int km
= 0;   int i, choice;
char c, ch;
          count = 8;

          strcpy(production[0], "X=TnS");
strcpy(production[1], "X=Rm");        strcpy(production[2],
"T=q");          strcpy(production[3], "T=#");
strcpy(production[4], "S=p");          strcpy(production[5],
"S=#");          strcpy(production[6], "R=om");
          strcpy(production[7], "R=ST");

 int kay;  char
done[count];
          int ptr = -1;

          for (k = 0; k < count; k++) {   for
(kay = 0; kay < 100; kay++) {
```

```
            calc_first[k][kay] = '!';
                }
        }
        int point1 = 0, point2, xxx;

        for (k = 0; k < count; k++) {
c = production[k][0];       point2 = 0;
            xxx = 0;
        for (kay = 0; kay <= ptr; kay++)
                if (c == done[kay])
                        xxx = 1;
        if (xxx == 1)
           continue;

 findfirst(c, 0, 0);  ptr += 1;
        done[ptr] = c;
        printf("\n First(%c) = { ", c);
        calc_first[point1][point2++] = c;

        for (i = 0 + jm; i < n; i++) {
int lark = 0, chk = 0;

        for (lark = 0; lark < point2; lark++) {
                if (first[i] == calc_first[point1][lark]) {
                chk = 1;
                        break;
                }
        }
        if (chk == 0) {
printf("%c, ", first[i]);
                calc_first[point1][point2++] = first[i];
                        }
        }
        printf("}\n");
jm = n;
point1++;
        }
        printf("\n");
        printf("-----------------------------------------------\n\n"); char
        donee[count];
        ptr = -1;
        for (k = 0; k < count; k++) {             for
(kay = 0; kay < 100; kay++) {
                        calc_follow[k][kay] = '!';
```

```
                }
        }
        point1 = 0;     int land = 0;
for (e = 0; e < count; e++) {
                ck = production[e][0];
        point2 = 0;
                xxx = 0;

        for (kay = 0; kay <= ptr; kay++)
                if (ck == donee[kay])
                xxx = 1;
        if (xxx == 1)
                continue;
land += 1;              follow(ck);
ptr += 1;       donee[ptr] = ck;
printf(" Follow(%c) = { ", ck);
         calc_follow[point1][point2++] = ck;

         for (i = 0 + km; i < m; i++) {
int lark = 0, chk = 0;          for (lark = 0; lark <
point2; lark++) {
                        if (f[i] == calc_follow[point1][lark]) {
                                chk = 1;
                                break;
                                }
                        }
                        if (chk == 0) {
                printf("%c, ", f[i]);
                                calc_follow[point1][point2++] = f[i];
                        }
                }
                printf(" }\n\n");
                km = m;
point1++;
        }
}
void follow(char c)
{
        int i, j;
        if (production[0][0] == c) {
                f[m++] = '$';
        }
```

```
        for (i = 0; i < 10; i++) {                  for (j = 2; j <
10; j++) {                          if (production[i][j] == c) {
                if (production[i][j + 1] != '\0') {
                                        followfirst(production[i][j + 1], i,(j + 2));
                                }
                                if (production[i][j + 1] == '\0'&& c != production[i][0]) {
                                        follow(production[i][0]);
                                }
                        }
                }
        }
}
void findfirst(char c, int q1, int q2)
{
        int j;
        if (!(isupper(c))) {
                first[n++] = c;
        }
        for (j = 0; j < count; j++) {               if
(production[j][0] == c) {                           if
(production[j][2] == '#') {
                                if (production[q1][q2] == '\0')
                                        first[n++] = '#';
                                else if (production[q1][q2] != '\0'&& (q1 != 0 || q2 != 0)) {
                                        findfirst(production[q1][q2], q1,(q2 + 1));
                                }
                                else
                                        first[n++] = '#';
                        }
                        else if (!isupper(production[j][2])) {
        first[n++] = production[j][2];
                        }
                        else {
                                findfirst(production[j][2], j, 3);
                        }
                }
        }
}
void followfirst(char c, int c1, int c2)
{
        int k;
        if (!(isupper(c)))
```

```
                f[m++] = c;
else {          int i = 0, j =
1;
                for (i = 0; i < count; i++) {
        if (calc_first[i][0] == c)
                        break;
                }
                while (calc_first[i][j] != '!') {
                        if (calc_first[i][j] != '#') {
                                f[m++] = calc_first[i][j];
                        }
                        else {
                                if (production[c1][c2] == '\0') {

                                        follow(production[c1][0]);
                                }
                                else {
                                        followfirst(production[c1][c2], c1,c2 + 1);
                                }
                        }j++;
                }
        }
}
```

Output:

First(X) = { q, n, o, p, #,m}

First(T) = { q, #, }

First(S) = { p, #, }

First(R) = { o, p, q, #, }

Follow(X) = { $, }

Follow(T) = { n, m, }

Follow(S) = { $, q, m, }

Follow(R) = { m, }

Practical-4

Aim: To implement the Left most derivation removal algorithm.

Code:

```
#include<stdio.h>
#include<string.h> void
main() {
   char input[100],l[50],r[50],temp[10],tempprod[20],productions[25][50];
int i=0,j=0,flag=0,consumed=0;          printf("Enter the productions: ");
scanf("%1s->%s",l,r);     printf("%s",r);
   while(sscanf(r+consumed,"%[^|]s",temp) == 1 && consumed <= strlen(r)) {
if(temp[0] == l[0]) {
          flag = 1;
          sprintf(productions[i++],"%s->%s%s'\0",l,temp+1,l);
       }
       else
          sprintf(productions[i++],"%s'->%s%s'\0",l,temp,l);
consumed += strlen(temp)+1;
   }     if(flag ==
1) {
       sprintf(productions[i++],"%s->ε\0",l);
       printf("The productions after eliminating Left Recursion are:\n");
for(j=0;j<i;j++)
          printf("%s\n",productions[j]);
   }
else
       printf("The Given Grammar has no Left Recursion");
}
```

Output :

Enter the productions: E->E+E|T
The productions after eliminating Left Recursion are:
E->+EE'
E'->TE'
E->ε

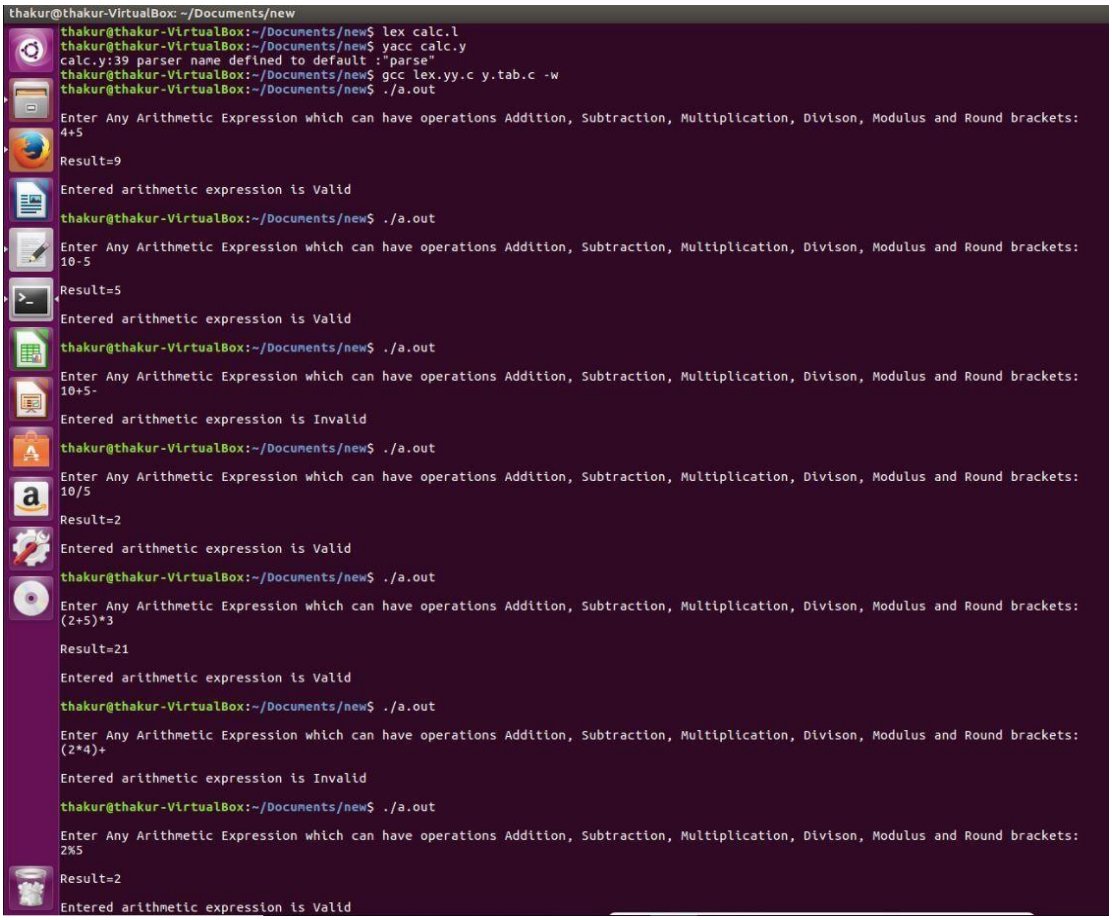
Practical-5

Aim:To implement a calculator in YACC.

Code:

```
%{
  #include<stdio.h>
  int flag=0;
%}
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
%%
ArithmeticExpression: E{printf("\nResult=%d\n", $$); return 0;};
 E:E'+'E {$$=$1+$3;}
 |E'-'E {$$=$1-$3;}
 |E'*'E {$$=$1*$3;}
 |E'/'E {$$=$1/$3;}
 |E'%'E {$$=$1%$3;}
 |'('E')' {$$=$2;}
 | NUMBER {$$=$1;};
%%
void main(){
 printf("\nEnter Any Arithmetic Expression can have operations
Addition,Subtraction,Multiplication,Division,Modulus and Round brackets:\n");
yyparse();    if(flag==0)
   printf("\nEntered arithmetic expression is Valid\n\n");
} void
yyerror(){
   printf("\nEntered arithmetic expression is Invalid\n\n");
flag=1;
}
```

Output:


thakur@thakur-VirtualBox: ~/Documents/new
thakur@thakur-VirtualBox:~/Documents/new$ lex calc.l
thakur@thakur-VirtualBox:~/Documents/new$ yacc calc.y
calc.y:39 parser name defined to default :"parse"
thakur@thakur-VirtualBox:~/Documents/new$ gcc lex.yy.c y.tab.c -w
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
4+5
Result=9
Entered arithmetic expression is Valid
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
10-5
Result=5
Entered arithmetic expression is Valid
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
10+5-
Entered arithmetic expression is Invalid
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
10/5
Result=2
Entered arithmetic expression is Valid
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
(2+5)*3
Result=21
Entered arithmetic expression is Valid
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
(2*4)+
Entered arithmetic expression is Invalid
thakur@thakur-VirtualBox:~/Documents/new$ ./a.out
Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Divison, Modulus and Round brackets:
2%5
Result=2
Entered arithmetic expression is Valid

Practical-6

To generate Three Address code for assignment statement.

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

int i,ch,j,l,addr=100;
char ex[10], exp[10] ,exp1[10],exp2[10],id1[5],op[5],id2[5];
void main() { clrscr();
printf("\nEnter the expression with assignment operator:"); scanf("%s",exp);
i=0;
while(exp[i]!='=')
{ i++; }
strncat(exp2,exp,i);
strrev(exp); exp1[0]='\0';
strncat(exp1,exp,l-(i+1));
strrev(exp1);
printf("Three address code:\ntemp=%s\n%s=temp\n",exp1,exp2);
}
```

Output:

Enter the expression with assignment operator:
a=b
Three address code:
temp=b
a=temp

Practical-7

Aim: To implement grammar rules for control statements, and Loop control.

Code:

1.If-else statement:

```
if (condition) {
  // code to execute if condition is true
} else {
  // code to execute if condition is false
}
```

2.Switch statement:

```
switch (expression) {
case value1:
    // code to execute if expression is equal to value1
break;   case value2:
    // code to execute if expression is equal to value2
break;   default:
    // code to execute if expression is not equal to any of the values
break;
}
```

3.For loop:

```
for (initialization; condition; increment/decrement) {   //
code to execute repeatedly as long as condition is true }
```

4.While loop:

```
while (condition) {
  // code to execute repeatedly as long as condition is true }
```

5.Do-while loop:

```
do {
  // code to execute at least once, then repeatedly as long as condition is true }
while (condition);
```

Practical-8

To implement a Type Checker.

```
#include<stdio.h>  #include<stdlib.h>  int
main() {        int n,i,k,flag=0;       char
vari[15],typ[15],b[15],c;       printf("Enter
the number of variables:");       scanf("
%d",&n);
   for(i=0;i<n;i++)
  {
     printf("Enter the variable[%d]:",i);
scanf(" %c",&vari[i]);
     printf("Enter the variable-type[%d](float-f,int-i):",i);
scanf(" %c",&typ[i]);      if(typ[i]=='f')     flag=1;  }
 printf("Enter the Expression(end with $):");
 i=0;   getchar();
while((c=getchar())!='$')
 {
b[i]=c;
i++;  }
  k=i;
 for(i=0;i<k;i++)
 {
if(b[i]=='/')
{    flag=1;
break;
  } }
for(i=0;i<n;i++
) {
  if(b[0]==vari[i])
{
 if(flag==1) {
if(typ[i]=='f')
{
   printf("\nthe datatype is correctly defined..!\n");
   break;
} else {
```

```
   printf("Identifier %c must be a float type..!\n",vari[i]);
break;
  }
} else
{
   printf("\nthe datatype is correctly defined..!\n");
   break;
  }
} }
return 0;
}
```

Output:

```
Enter the number of variable : 4
Enter the variable[0]: A
Enter the variable-type[0](float-f,int-i): i
Enter the variable[1]: B
Enter the variable-type[1](float-f,int-i): i Enter
the variable[2]: C
Enter the variable-type[2](float-f,int-i): f Enter
the variable[3]: D
Enter the variable-type[3](float-f,int-i): i
Enter the Expression(end with $):A=B*C/D$
Identifier A must be a float type..!
```

Practical-9

To implement Assembly code generator.

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

// Define the opcode for each instruction
typedef enum {      ADD, SUB, MUL,
DIV,MOV
} Opcode;

// Define the type of each operand typedef
enum {
   REGISTER, IMMEDIATE, LABEL
} OperandType;

// Define the operand structure
typedef          struct          {
OperandType type;     union {
char*  reg;              int  imm;
char* label;     } value;
} Operand;

// Define the instruction structure
typedef  struct  {          Opcode
opcode;
   Operand* src;
   Operand* dest;
} Instruction;

// Function to create a new operand
Operand* new_operand(OperandType type, void* value) {
Operand* operand = malloc(sizeof(Operand));     operand-
>type = type;     if (type == REGISTER) {        operand-
>value.reg = (char*) value;
```

```
   } else if (type == IMMEDIATE) {
operand->value.imm = (int) value;     }
else if (type == LABEL) {
       operand->value.label = (char*) value;
   }
   return operand;
}

// Function to create a new instruction
Instruction* new_instruction(Opcode opcode, Operand* src, Operand* dest) {
   Instruction* instr = malloc(sizeof(Instruction));
instr->opcode = opcode;    instr->src = src;    instr-
>dest = dest;     return instr;
}

// Function to output an operand as a string char*
operand_to_string(Operand* operand) {     char*
str = malloc(sizeof(char) * 10);     switch
(operand->type) {         case REGISTER:
sprintf(str, "%%%s", operand->value.reg);
break;         case IMMEDIATE:
           sprintf(str, "$%d", operand->value.imm);
break;         case LABEL:
           sprintf(str, "%s", operand->value.label);
break;
   }
return str;
}

// Function to output an instruction as a string
char* instruction_to_string(Instruction* instr) {
char* opcode_str;     switch (instr->opcode) {
case ADD:             opcode_str = "add";
           break;         case
SUB:             opcode_str =
"sub";            break;
```

```
        case MUL:
opcode_str = "mul";
            break;              case
DIV:              opcode_str =
"div";
            break;
    }
    char* src_str = operand_to_string(instr->src);     char*
dest_str = operand_to_string(instr->dest);     char* str =
malloc(sizeof(char) * 30);     sprintf(str, "%s %s, %s",
opcode_str, dest_str, src_str);
    free(src_str);
free(dest_str);     return
str;
}

int main() {
    Operand* src = new_operand(REGISTER, "ebx");
    Operand* dest = new_operand(REGISTER, "eax");
Instruction* instr = new_instruction(ADD, src, dest);
printf("%s\n", instruction_to_string(instr));
    free(src);
free(dest);
free(instr);     return
0;
}
```

Output:

```
MOV  reg2, reg1
ADD   reg2,1
MOV  reg1,reg2
```

Practical-10

Aim: To implement Code Optimization techniques.

Code:

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

struct op {
char 1;      char
r[20]; }
op[10], pr[10]:

void main()
{
     int a, ik, j, n, z = 0 , m, q;
char * p, * 1;        char tempt;
char * tem;        clrscr();
     printf("enter no of values");
scanf("%d", & n);       for ( i =
0; i < n ;i++)
   {
printf("\tleft\t");
op[i].1 = getche()
printf("\tright:\t");
      scanf("%s", op[i].r);
   }
   printf("intermediate Code\n");
for (i = 0; i < n; i++)
  {
     printf("%c=", op[i].1);
printf("%s\n", op[i].r);
   }
   for (i = 0; i < n -1; i++)
   {        temp = op[i].1;
for (j = 0; j < n; j++)
     {
         p = strchr(op[j].r, temp);
         if (p)
{
             pr[z].1 = op[i].1;
strcpy(pr[z].r, op[i].r);              z++;
```

```
      }
   } } pr[z].1 = op[n-1].1;
strcpy(pr[z].r, op[n-1].r);
z++; printf("optimized
code");  for (i=0; i<z; i++) {
if (pr[i].l!= '\0') {
printf("%c=", pr[i].1);
printf("%s\n", pr[i].r);
    }  }
getch();
}
```

OUTPUT:

```
left a     right: 9 left
b     right: c+d left
e     right: c+d left
f     right: b+e left r
right: f
```

intermediate Code

```
a=9 b=c+d
e=c+d
f=b+e r=f
```

optimized codeb=c+d

```
f=b+b r=f
```