Compiler Construction Programs

Lex Programs

1. Implement Lexical Analyzer
2. Lex program to recognize the numbers which has 1 in its 5^{th} position from right
3. Lex program to recognize the Strings which are starting or ending with 'k'
4. Lex program to recognize the Strings ending with 11
5. Lex program to recognize Keywords
6. Lex program to recognize the Strings ending with 00
7. Lex program to assign line numbers for source code
8. Lex program to recognize Identifiers
9. Lex program to recognize operators

Compiler Programs

1. Implement the SLR (1) parsing table for the given grammar (Python)
    a. E->E+T|T
    b. T->T*F|T
    c. F->id|(E)
2. Implement Scanner using C (C)
3. Implement the Three Address Code using YACC
4. Construct DAG for the given three address code
5. Implement the Dependency Graph (Python)
6. Implement the Recursive Descent Parser (Python)
7. Implement Intermediate Code Generation using YACC
8. Implement First & Follow (Python)
9. Implement a YACC specification for simple arithmetic calculations
10. Implement LL(1) Parser (Python)

Lex Programs :

1. **Implement Lexical Analyzer**

```
%{
 #include<stdio.h>
 #include<stdlib.h>

 int line;
 int loc;
 int id;
 char name[100];
 FILE* fp;

%}


keyword
char|short|int|long|double|float|if|else|for|do|while|void|switch|break|continu
e|case|return
identifier  [_a-zA-Z][_a-zA-Z0-9]*
number     [0-9]+
arithmetic  (\+)|(\-)|(\*)|(\/)|(\%)
relational  <|>|<=|>=|!=|==
assignment  =
special    \(|\)|[|]|\{|\}|;|\"|\'|#|\?|:|\.


%%
{keyword}    {printf("ID: %-8dType: Keyword     Line: %3d[%-3d]  Symbol: %s\n",id++,
line, loc, yytext); loc+=yyleng;}
{identifier} {printf("ID: %-8dType: Identifier   Line: %3d[%-3d]  Symbol: %s\n",id++,
line, loc, yytext); loc+=yyleng;}
{number}     {printf("ID: %-8dType: Number      Line: %3d[%-3d]  Symbol: %s\n",id++,
line, loc, yytext); loc+=yyleng;}
{arithmetic} {printf("ID: %-8dType: ArithOper    Line: %3d[%-3d]  Symbol: %s\n",id++,
line, loc, yytext); loc+=yyleng;}
{relational} {printf("ID: %-8dType: RelatOper    Line: %3d[%-3d]  Symbol: %s\n",id++,
line, loc, yytext); loc+=yyleng;}
{assignment} {printf("ID: %-8dType: AssignOper   Line: %3d[%-3d]  Symbol:
%s\n",id++, line, loc, yytext); loc+=yyleng;}
{special}    {printf("ID: %-8dType: SpecialChar  Line: %3d[%-3d]  Symbol: %s\n",id++,
line, loc, yytext); loc+=yyleng;}
\n          {line++;loc=1;}
.          {loc+=yyleng;}
%%
```

```
int main()
{
id = 0;
line = 1;
loc = 1;
printf("Enter file name : ");
scanf("%s",name);
fp = fopen(name, "r");
if(!fp)
{
printf("Could not open the file");
exit(0);
}
yyin=fp;
printf("Lex output : \n");
yylex();
fclose(fp);
printf("Over");
return 0;
}

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

2. <u>Lex program to recognize the numbers which has 1 in its 5th position from right</u>

```
%{
#include<stdio.h>
#include<stdlib.h>
%}

%%
[0-9]*1[0-9]{4}$ printf("%s has 1 at position 5 from right\n", yytext);
.* printf("%s does not match pattern\n",yytext);
%%

int main()
{
yylex();
return 0;
}
```

3. **Lex program to recognize the Strings which are starting or ending with 'k'**

```
%{
#include<stdio.h>
#include<stdlib.h>
%}

%%
^[k](.)* {printf("%s starts with k\n",yytext);}
(.)*[k]$ {printf("%s ends with k\n",yytext);}
(.)* {printf("%s does not match k-pattern\n",yytext);}
%%

int main()
{
yylex();
return 0;
}
```

4. **Lex program to recognize the Strings ending with 11**

```
%{
#include<stdio.h>
#include<stdlib.h>
%}


%%
(.)*11$ {printf("%s ends with 11\n",yytext);}
.* {printf("%s does not end with 11\n",yytext);}
%%

int main()
{
yylex();
return 0;
}
```

5. **Lex program to recognize Keywords**

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
```

```
%%
char|short|int|long|double|float|if|else|for|do|while|void|switch|break|continu
e|case|return {printf("%s is a keyword\n",yytext);}
.* printf("%s not a keyword\n",yytext);
%%

int main()
{
yylex();
return 0;
}
```

6. **Lex program to recognize the Strings ending with 00**

```
%{
#include<stdio.h>
#include<stdlib.h>
%}


%%
(.)*00$ {printf("%s ends with 00\n",yytext);}
.* {printf("%s does not end with 00\n",yytext);}
%%

int main()
{
yylex();
return 0;
}
```

7. **Lex program to assign line numbers for source code**

```
%{
#include<stdio.h>
#include<string.h>
int line;
int j;
int dataline;
char name[100];
FILE* fp;
char data[100][199];
%}
```

```
%%
[\n] {strcat(data[dataline], "\n\0"); dataline++; line++; data[dataline][0]=line+48;
data[dataline][1]='\0';}
. {strcat(data[dataline],yytext);}
%%

int main()
{
dataline = 0;
line = 1;
printf("Enter file name : ");
scanf("%s",name);
fp = fopen(name, "r");
yyin = fp;
yylex();
fclose(fp);

fp = fopen(name, "w");
fprintf(fp,"1");
for(j=0; j<line; j++)
fprintf(fp, data[j]);
fclose(fp);
return 0;
}
```

8. <u>Lex program to recognize Identifiers</u>

```
%{
#include<stdio.h>
#include<stdlib.h>
%}

%%
^[a-zA-Z_][a-zA-Z0-9_]*$ printf("%s is valid identifier\n", yytext);
.* printf("%s is invalid\n",yytext);
%%

int main()
{
yylex();
return 0;
}
```

9. Lex program to recognize operators

```
%{
#include<stdio.h>
#include<stdlib.h>
%}

%%
[\+\-\*\\] printf("%s is arithmetic operator\n", yytext);
[=] printf("%s is assignment operator\n", yytext);
[,] printf("%s is comma operator\n", yytext);
[<|>]|(<=)|(>=)|(==)|(!=) printf("%s is relational operator\n", yytext);
.* printf("%s is not operator\n",yytext);
%%

int main()
{
yylex();
return 0;
}
```

Compiler Programs :

1. Implement the SLR (1) parsing table for the given grammar (Python)

a. E->E+T|T

b. T->T*F|T

c. F->id|(E)

```
// (on cmd) pip install firfol==0.2.1

from collections import deque
from collections import OrderedDict
from pprint import pprint
from firfol import makeGrammar, findFirsts, findFollows

rules   = ["E->TA",
"A->+TA|eps",
"T->FB",
"B->*FB|eps",
"F->i|(E)"]
start   = 'E'
aug     = ''
nt_list = ['E', 'A', 'T', 'B', 'F']
t_list  = ['$', '+', '*', 'i', '(', ')']
g       = makeGrammar(rules)
firsts  = findFirsts(g)
follows = findFollows(g, start)

class State:
    _id=0
    def __init__(self, closure):
        self.closure=closure
        self.no=State._id
        State._id+=1

class Item(str):
    def __new__(cls, item):
        self=str.__new__(cls, item)
        return self
    def __str__(self):
        return super(Item, self).__str__()
```

```
def closure(items):
    def exists(newitem, items):
        for i in items:
            if i==newitem:
                return True
        return False
    global g
    while True:
        flag=0
        for i in items:
            if i.index('.')==len(i)-1: continue
            Y=i.split('->')[1].split('.')[1][0]
            if i.index('.')+1<len(i)-1 and i[-1] in nt_list:
                lastr=list(firsts[i[i.index('.')+2]]-set(chr(1013)))
            for prod in g.keys():
                head, body=prod, g[prod]
                if head!=Y: continue
                for b in body:
                    newitem=Item(Y+'->.'+b)
                    if not exists(newitem, items):
                        items.append(newitem)
                        flag=1
        if flag==0: break
    return items

def goto(items, symbol):
    initial=[]
    for i in items:
        if i.index('.')==len(i)-1: continue
        head, body=i.split('->')
        seen, unseen=body.split('.')
        if unseen[0]==symbol and len(unseen) >= 1:
            initial.append(Item(head+'->'+seen+unseen[0]+'.'+unseen[1:]))
    return closure(initial)

def calc_states():
    def contains(states, t):
        for s in states:
            if len(s) != len(t): continue
            if sorted(s)==sorted(t):
                for i in range(len(s)):
                    if s[i]!=t[i]: break
                else: return True
        return False
    global g, nt_list, t_list, aug
```

```
    head, body=aug, g[aug]
    for b in body:
        states=[closure([Item(head+'->.'+b)])]
    while True:
        flag=0
        for s in states:
            for e in nt_list+t_list:
                t=goto(s, e)
                if t == [] or contains(states, t): continue
                states.append(t)
                flag=1
        if not flag: break
    return states

def make_table(states):
    global nt_list, t_list
    def getstateno(t):
        for s in states:
            if len(s.closure) != len(t): continue
            if sorted(s.closure)==sorted(t):
                for i in range(len(s.closure)):
                    if s.closure[i]!=t[i]: break
                else: return s.no
        return -1
    def getprodno(closure):
        closure=''.join(closure).replace('.', '')
        return list(g.keys()).index(closure.split('->')[0])
    SLR_Table=OrderedDict()
    for i in range(len(states)):
        states[i]=State(states[i])
    for s in states:
        SLR_Table[s.no]=OrderedDict()
        for item in s.closure:
            head, body=item.split('->')
            if body=='.':
                for term in follows[item.split('->')[0]]:
                    if term not in SLR_Table[s.no].keys():
                        SLR_Table[s.no][term]={'r'+str(getprodno(item))}
                    else: SLR_Table[s.no][term] |= {'r'+str(getprodno(item))}
                continue
            nextsym=body.split('.')[1]
            if nextsym=='':
                if getprodno(item)==0:
                    SLR_Table[s.no]['$']='A'
                else:
                    for term in follows[item.split('->')[0]]:
```

```
                    if term not in SLR_Table[s.no].keys():
                        SLR_Table[s.no][term]={'r'+str(getprodno(item))}
                    else: SLR_Table[s.no][term] |= {'r'+str(getprodno(item))}
                continue
            nextsym=nextsym[0]
            t=goto(s.closure, nextsym)
            if t != []:
                if nextsym in t_list:
                    if nextsym not in SLR_Table[s.no].keys():
                        SLR_Table[s.no][nextsym]={'s'+str(getstateno(t))}
                    else: SLR_Table[s.no][nextsym] |= {'s'+str(getstateno(t))}
                else: SLR_Table[s.no][nextsym] = str(getstateno(t))
    return SLR_Table

def augment_grammar():
    global start, aug
    for i in range(ord('Z'), ord('A')-1, -1):
        if chr(i) not in nt_list:
            g[chr(i)]=start
            aug = chr(i)
            return

def main():
    global ntl, nt_list, tl, t_list
    augment_grammar()
    follows[aug] = ['$']
    nt_list = list(g.keys())
    j = calc_states()
    ctr=0
    for s in j:
        print("Item{}:".format(ctr))
        for i in s:
            print("\t", i)
        ctr+=1
    table=make_table(j)
    print('_____________________________________________________________')
    print("\n\tSLR(1) TABLE\n")
    sym_list = nt_list + t_list
    print('_____________________________________________________________')
    print('\t|  ','\t|  '.join(sym_list),'\t\t|')
    print('_____________________________________________________________')
    for i, j in table.items():
        print(i, "\t|  ", '\t|  '.join(list(j.get(sym,' ') if type(j.get(sym))in (str , None) else
next(iter(j.get(sym,' ')))  for sym in sym_list)),'\t\t|')
        s, r=0, 0
        for p in j.values():
```

```
            if p!='accept' and len(p)>1:
                p=list(p)
                if('r' in p[0]): r+=1
                else: s+=1
                if('r' in p[1]): r+=1
                else: s+=1
        print('_____')
        return

main()
```

2. Implement Scanner using C (C)

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


char* keywords[40] = {
   "auto",    "break",    "case",     "char",
   "const",   "continue",  "default",   "do",
   "double",  "else",     "enum",     "extern",
   "float",   "for",      "goto",     "if",
   "int",     "long",     "register",  "return",
   "short",   "signed",   "sizeof",   "static",
   "struct",  "switch",   "typedef",  "union",
   "unsigned", "void",     "volatile",  "while",
   "main",    "include"
};
int KEYWORDS = 34;


char* operators[50] = {
   "/",  "/=",  "//",  "/*",

   "+",  "++",  "+=",
   "-",  "--",  "-=",
   "<",  "<=",  "<<",
   ">",  ">=",  ">>",
   "*",  "*=",  "*/",

   "%",  "%=",
   "!",  "!=",
   "&",  "&&",
   "|",  "||",
   "=",  "==",

   "^", "~", ".", ";", "#", "?", ":", "'", "\"",
   "(", ")", "[", "]", "{", "}", "\'"
};
int OPERATORS = 45;


int scomment = 0;
int mcomment = 0;
int string   = 0;
int number   = 0;
int unidentified = 0;
```

```
int character = 0;
int lines;

char stack[1000];
int tos = 0;


void op(int a){
   switch(a){
      case 2:
         scomment = 1;
         break;
      case 3:
         mcomment = 1;
         break;
      case 18:
         mcomment = 0;
         break;
      case 37:
         string = -string + 1;
         break;
      case 44:
         character = -character + 1;
   }
   return;
}


int checkid(){
   if(tos==0) return 0;
   if(stack[tos]!='_'  &&
    !(stack[tos]>='A' && stack[tos]<='Z') &&
    !(stack[tos]>='a' && stack[tos]<='z'))
       return 0;
   int k;
   for(k=1; k<tos; k++){
      if(!(stack[k]=='_' ||
      (stack[k]>='A' && stack[k]<='Z') ||
      (stack[k]>='a' && stack[k]<='z') ||
      (stack[k]>='0' && stack[k]<='9')))
         return 0;
   }
   return 1;
}
```

```
int checkkey(){
   char new[tos+1];
   int k;
   for(k=0; k<tos; k++)
   new[k] = stack[k];
   new[tos]='\0';

   for(k=0; k<KEYWORDS; k++){
      if (strcmp(new, keywords[k])==0) return 1;
   }
   return 0;
}


int push(char a){
   if (tos==1000){
      printf("Stack Overflow..");
      return -1;
   }

   if(tos==0 && (a>='0' && a<='9')) number=1;
   stack[tos] = a;
   tos++;
   return tos;
}



void stackdump(){
   if(tos==0) return;

   if(scomment || mcomment)       printf("%-14dComment       ",lines);
   else if(string)               printf("%-14dString        ",lines);
   else if(character)            printf("%-14dCharacter     ",lines);
   else if(unidentified)         printf("%-14dUnidentified  ",lines);
   else if(number)               printf("%-14dConstant      ",lines);
   else if(checkkey())           printf("%-14dKeyword       ",lines);
   else if(checkid())            printf("%-14dIdentifier    ",lines);

   int k;
   for(k=0; k<tos; k++){
      printf("%c", stack[k]);
   }

   printf("\n");
```

```
    tos = 0;
    number = 0;
    return;
}


int isoperator(char* c){
    int k;
    for(k=0; k<OPERATORS; k++){
        if(strcmp(c, operators[k])==0){
            stackdump();
            op(k);
            printf("%-14dOperator       %s\n",lines, c);
            return 1;
        }
    }
    return 0;
}


int main(int argc, char* argv[])
{
    if(argc<2){
        printf("File name missing..\n");
        return -1;
    }

    FILE* file = fopen(argv[1], "r");

    if(!file){
        printf("Unable to open : %s..\n", argv[1]);
        return -1;
    }

    char line[500];
    int ptr = 0;
    lines = 0;
    char temp[2];
    char stemp[3];
    temp[1] = '\0';
    stemp[2] = '\0';

    printf("Line No.      Token type     Token\n-----------------------------------\n");

    while(fgets(line, sizeof(line), file)!=NULL){
      lines++;
```

```
        for(ptr = 0; ;ptr++){
            temp[0] = line[ptr];
            if(temp[0]=='\n'){
                stackdump();
                scomment = 0;
                string   = 0;
                break;
            }

            if(temp[0]=='\0'){
                stackdump();
                break;
            }

            stemp[0] = temp[0];
            stemp[1] = line[ptr+1];

            if(isoperator(stemp)){
                ptr++;
                continue;
            }

            if(temp[0]==' ')
                stackdump();
            else if ((temp[0]>='A' && temp[0]<='Z') ||
                    (temp[0]>='a' && temp[0]<='z') ||
                    (temp[0]>='0' && temp[0]<='9') ||
                    temp[0]=='_'){
                if (number && !(temp[0]>='0' && temp[0]<='9')) unidentified = 1;
                if (push(temp[0])==-1) return -1;
            }else if (isoperator(temp))
                continue;
        }
        memset(line, '\0', sizeof(line));
    }

    printf("Lines seen : %d\n", lines);
    return 0;
}
```

3. Implement the Three Address Code using YACC

3add.y

```
%{
#include<stdio.h>
#include<string.h>
int nIndex = 0;
struct Intercode{
char operand1;
char operand2;
char opera;};
%}

%union{char sym;}
%token <sym> letter number
%type <sym> expr
%left '-''+'
%right '*''/'

%%
statement: letter'='expr';' {addtotable((char)$1, (char)$3, '=');}
|expr;
;
expr: expr'+'expr {$$=addtotable((char)$1,(char)$3, '+');}
|expr'-'expr {$$=addtotable((char)$1,(char)$3, '-');}
|expr'*'expr {$$=addtotable((char)$1,(char)$3, '*');}
|expr'/'expr {$$=addtotable((char)$1,(char)$3, '/');}
|'('expr')' {$$=(char)$2;}
|number {$$=(char)$1;}
|letter {$$=(char)$1;}
%%

yyerror(char *s){
printf("%s",s);
exit(0);}

struct Intercode code[20];
char addtotable(char operand1, char operand2, char opera){
char temp='A';
code[nIndex].operand1 = operand1;
code[nIndex].operand2 = operand2;
code[nIndex].opera = opera;
nIndex++;
temp++;
return temp;
}
```

```
threeaddresscode(){
int nCnt=0;
char temp='A';
printf("\n\n\t three address codes\n\n");
temp++;
while(nCnt < nIndex){
printf("%c:=\t",temp);
if(isalpha(code[nCnt].operand1))
printf("%c\t", code[nCnt].operand1);
else
printf("%c\t", temp);
printf("%c\t",code[nCnt].opera);
if(isalpha(code[nCnt].operand2))
printf("%c\t", code[nCnt].operand2);
else
printf("%c\t", temp);
printf("\n");
nCnt++;
temp++;}}

main(){
printf("Enter expression : ");
yyparse();
threeaddresscode();}

yywrap(){
return 1;}
```

3addlex.l

```
%{
#include "y.tab.h"
extern char yyval;
%}
number [0-9]+
letter [a-zA-Z]+
%%
{number} {yylval.sym=(char)yytext[0]; return number;}
{letter}  {yylval.sym=(char)yytext[0]; return letter;}
\n {return 0;}
{return yytext[0];}
%%
```

4. Construct DAG for the given three address code

3add.y

```
%{
#include<stdio.h>
#include<string.h>
int nIndex = 0;
struct Intercode{
char operand1;
char operand2;
char opera;};
%}

%union{char sym;}
%token <sym> letter number
%type <sym> expr
%left '-''+'
%right '*''/'

%%
statement: letter'='expr';' {addtotable((char)$1, (char)$3, '=');}
|expr;
;
expr: expr'+'expr {$$=addtotable((char)$1,(char)$3, '+');}
|expr'-'expr {$$=addtotable((char)$1,(char)$3, '-');}
|expr'*'expr {$$=addtotable((char)$1,(char)$3, '*');}
|expr'/'expr {$$=addtotable((char)$1,(char)$3, '/');}
|'('expr')' {$$=(char)$2;}
|number {$$=(char)$1;}
|letter {$$=(char)$1;}
%%

yyerror(char *s){
printf("%s",s);
exit(0);}

struct Intercode code[20];
char addtotable(char operand1, char operand2, char opera){
char temp='A';
code[nIndex].operand1 = operand1;
code[nIndex].operand2 = operand2;
code[nIndex].opera = opera;
nIndex++;
temp++;
return temp;
}
```

```
threeaddresscode(){
int nCnt=0;
char temp='A';
printf("\n\n\t three address codes\n\n");
temp++;
while(nCnt < nIndex){
printf("%c:=\t",temp);
if(isalpha(code[nCnt].operand1))
printf("%c\t", code[nCnt].operand1);
else
printf("%c\t", temp);
printf("%c\t",code[nCnt].opera);
if(isalpha(code[nCnt].operand2))
printf("%c\t", code[nCnt].operand2);
else
printf("%c\t", temp);
printf("\n");
nCnt++;
temp++;}}

main(){
printf("Enter expression : ");
yyparse();
threeaddresscode();}

yywrap(){
return 1;}
```

3addlex.l

```
%{
#include "y.tab.h"
extern char yyval;
%}
number [0-9]+
letter [a-zA-Z]+
%%
{number} {yylval.sym=(char)yytext[0]; return number;}
{letter}  {yylval.sym=(char)yytext[0]; return letter;}
\n {return 0;}
{return yytext[0];}
%%
```

5. Implement the Dependency Graph (Python)

```
rules = {
   "S":("E"),
   "E":("T+E", "T*T", "T+T"),
   "T":("d")
}

def getType(a):
  if a in ['1','2','3','4','5','6','7','8','9','0']:
    print('d.val = '+a)
    return 'T'
  for c in rules.keys():
    if a in rules[c]:
      return c

def parse(a):
  if len(a)==1:
    rep   = (getType(a), int(a))
    print(rep[0]+'.val =', rep[1])
    return rep
  if '+' in a:
    ind = a.find('+')
    terms = [a[:ind], a[ind+1:]]
    t2    = parse(terms[1])
    t1    = parse(terms[0])
    rep   = (getType(t1[0]+'+'+t2[0]), t1[1]+t2[1])
    print(rep[0]+'.val =', rep[1])
    return rep
  if '*' in a:
    ind = a.find('*')
    terms = [a[:ind], a[ind+1:]]
    t2    = parse(terms[1])
    t1    = parse(terms[0])
    rep   = (getType(t1[0]+'*'+t2[0]), t1[1]*t2[1])
    print(rep[0]+'.val =', rep[1])
    return rep

inp = input('Enter an expression that follows the regular expression : ([0-9]\+)*\*[0-
9] i.e. a+b+c+...+d*e \nExpression  : ')

print('\n\nFlow of actions in dependancy graph : ')
out = parse(inp)
if out[0]=='E':
  print('S.val =', out[1])
```

6. Implement the Recursive Descent Parser (Python)

```
n = int(input("Enter no. of production rules : ").strip())
prods = {}

print("Enter production rules in the format :\nSymbol -> production1 | production2
| ...")
print("Note : Enter epsilon as 'epsilon' and do not use any epsilon symbol")
for k in range(n):
  line = input().strip().split("->")
  prods[line[0].strip()] = list(map(str.strip, line[1].split('|')))

nonterminals = set(prods.keys())

print()
start = ''
while start=="":
  start = input("Enter start symbol : ").strip()
  if start not in nonterminals:
    print("Wrong start symbol")
    start = ""
print('\n')


def RecursiveDescentParser(sym, seq):
  if sym=='' and seq!='':
    return False,''
  if seq=='' and sym=='':
    return True,''
  print('Checking for : ', sym, 'and', seq)
  if seq=='' and sym[0] not in nonterminals:
    return False,''
  if sym[0] not in nonterminals:
    if sym[0]==seq[0]:
      return RecursiveDescentParser(sym[1:], seq[1:])
    else:
      return False,''
  cases = prods[sym[0]]
  if seq=='' and 'epsilon' in cases:
    trial = RecursiveDescentParser(sym[1:],seq)
    if trial[0]==True:
      return True,sym[0]+'->epsilon\n'+trial[1]
    else:
      return False,''
  for case in cases:
```

```
    if case=='epsilon':
      trial = RecursiveDescentParser(sym[1:], seq)
    else:
      trial = RecursiveDescentParser(case+sym[1:], seq)
    if trial[0]==True:
      return True,sym[0]+'->'+case+'\n'+trial[1]
  return False,''

word  = input("Enter the string to be checked : ")
result,prod  = RecursiveDescentParser(start, word)
if result!=False:
  print('The given string can be accepted according to the production : \n'+prod)
else:
  print('The given string cannot be accepted')
```

7. Implement Intermediate Code Generation using YACC

lex.l

```
%{
#include “y.tab.h”
extern char yyval;
%}

%%
[0-9]+ {yylval.symbol=(char)(yytext[0]); return NUMBER;}
[a-z] {yylval.symbol=(char)(yytext[0]); return LETTER;}
. {return yytext[0];}
\n {return 0;}
%%
```

yacc.y

```
%{
#include “y.tab.h”
#include<stdio.h>
char addtotable(char,char,char);

int index1=0;
char temp=’A’-1;

struct expr{
char operand1;
char operand2;
char opera;
char result;
};
%}

%union{char symbol;}
%left ‘+’ ‘-‘
%right ‘/’ ‘*’
%token <symbol> LETTER NUMBER
%type <symbol> exp

%%
statement: LETTER ‘=’ exp ‘;’ {addtotable((char)$1, (char)$3, ‘=’);};
exp: exp ‘+’ exp {$$ = addtotable((char)$1, (char)$3, ‘+’);}
|exp ‘-’ exp {$$ = addtotable((char)$1, (char)$3, ‘-’);}
|exp ‘/’ exp {$$ = addtotable((char)$1, (char)$3, ‘/’);}
|exp ‘*’ exp {$$ = addtotable((char)$1, (char)$3, ‘*’);}
|’(‘ exp ‘)’ {$$= (char)$2;}
|NUMBER {$$= (char)$1;}
```

```
|LETTER {(char)$1;};
%%

struct expr arr[20];

void yyerror(char *s){
printf("Error %s", s);
}

char addtotable(char a, char b, char o){
temp++;
arr[index1].operand1 = a;
arr[index1].operand2 = b;
arr[index1].opera = o;
arr[index1].result = temp;
index1++;
return temp;
}

void threeAdd(){
int i=0;
char temp='A';
while(i<index1){
printf("%c:=\t", arr[i].result);
printf("%c\t",arr[i].operand1);
printf("%c\t",arr[i].opera);
printf("%c\t",arr[i].operand2);
i++;
temp++;
printf("\n");
}}

void fourAdd(){
int i=0;
char temp='A';
while(i<index1){
printf("%c\t",arr[i].opera);
printf("%c\t",arr[i].operand1);
printf("%c\t",arr[i].operand2);
printf("%c", arr[i].result);
i++;
temp++;
printf("\n");
}}

int find(char l){
```

```
int i;
for(i=0; i<index1; i++)
if(arr[i].result==l) break;
return i;
}

void triple(){
int i=0;
char temp='A';
while(i<index1){
printf("%c\t",arr[i].opera);
if(!isupper(arr[i].operand1))
printf("%c\t",arr[i].operand1);
else{
printf("pointer");
printf("%d\t",find(arr[i].operand1);
}
if(!isupper(arr[i].operand2))
printf("%c\t",arr[i].operand2);
else{
printf("pointer");
printf("%d\t",find(arr[i].operand2);
}
i++;
temp++;
printf("\n");
}}

int yywrap(){
return 1;}

int main(){
printf("Enter the expression : ");
yyparse();
threeAdd();
printf("\n");
fouradd();
printf("\n");
triple();
return 0;
}
```

8. Implement First & Follow (Python)

```
prods = {
   'S':('ABd', 'CBd'),
   'A':('aB','kB'),
   'B':('b'),
   'C':('c')
}
nonterminals = set(prods.keys())
start = 'S'

firsts = {k:[] for k in nonterminals}
follows = {k:set() for k in nonterminals}

def fillfirst(symbol):
 if firsts[symbol]!=[]:
  return
 prodcases = prods[symbol]
 anslist = set()
 for case in prodcases:
  if case=='epsilon':
   anslist.add('epsilon')
   continue
  while case!='':
   if case[0] in nonterminals:
    fillfirst(case[0])
    anslist = anslist.union(firsts[case[0]])
    if 'epsilon' in prods[case[0]]:
     case = case[1:]
    else:
     case = ''
   else:
    anslist.add(case[0])
    case = ''
 firsts[symbol]=anslist

for symbol in nonterminals:
 fillfirst(symbol)

for k in prods.keys():
 print('FIRST(',k,") : ",firsts[k],sep='')

for key in prods.keys():
 anslist = set()
 for symbol in prods.keys():
  if symbol==key:
```

```
      continue
    prodcases = prods[symbol]
    for case in prodcases:
      if key not in case:
        continue
      if case.find(key)==len(case)-1:
        anslist = anslist.union(follows[symbol])
      else:
        rem = case[case.find(key)+1:]
        while rem!="":
          nextsym = rem[0]
          if nextsym in nonterminals:
            anslist = anslist.union(firsts[nextsym])
            if 'epsilon' in firsts[nextsym]:
              rem = rem[1:]
              continue
            else:
              break
          else:
            anslist.add(nextsym)
            break
        if rem=="":
          anslist = anslist.union(follows[symbol])
  if 'epsilon' in anslist:
    anslist.remove('epsilon')
  if key==start:
    anslist.add('$')
  follows[key] = anslist

print('\n\n')

for k in prods.keys():
  print('FOLLOWS(',k,") : ",follows[k],sep='')
```

9. Implement a YACC specification for simple arithmetic calculations

yacc1.y

```
%{
#include<stdio.h>
#include<ctype.h>
%}

%token NUM

%%
cmd:E {printf("%d\n", $1);}
E: E'+'T {$$= $1+$3;}
|T {$$= $1;};
E: E'-'T {$$= $1-$3;};
T: T'*'F {$$= $1*$3;}
|F {$$=$1;};
T: T'/'F {$$= $1/$3;};
F: '('E')' {$$= $2;};
NUM {$$= $1;};
%%

int yyerror(char* s){
printf("%s\n", s);
return 0;}

int main(){
yyparse();
return 0;}
```

yacclex1.l

```
%{
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+ {yylval=atoi(yytext); return NUM;}
\n {return 0;}
 {return yytext[0];}
%%

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

10. Implement LL(1) Parser (Python)

(on cmd run : pip install firfol==0.2.1)

```
from firfol import makeGrammar, findFirsts, findFollows

prods = makeGrammar(['A->BC', 'C->+BC|eps', 'B->DE', 'E->*DE|eps', 'D->a'])
nonterminals = set(prods.keys())
firsts = findFirsts(prods)
follows = findFollows(prods, 'A')

print('LL(1) Parsing table :')
print('--------------------')

for nt in nonterminals:
  print('\t',nt,":")
  ntprods = prods[nt]
  if 'eps' in ntprods:
    for ntfol in follows[nt]:
      print('\t\t'+ntfol+' : '+nt+'->eps')
    ntprods.remove('eps')
  if ntprods==[]:
    continue
  for ntfir in firsts[nt]:
    if ntfir=='eps':
      continue
    print('\t\t'+ntfir+' : '+nt+'->'+ntprods[0])
```