Compiler Construction Lab Manual In Python

LIST OF EXPERIMENTS:

1. Implementation of Lexical Analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments,operators etc.)
2. Implementation of Lexical Analyzer using LEX tool.
3. Implementation of Recursive Descent Parser.
4. Implementation of FIRST() of a given Context-Free Grammar.
5. Implementation of FOLLOW() of a given Context-Free Grammar.
6. Construction of a LL(1) for a given CFG.
7. Write a program for generating derivation sequence for a given terminal string using SLR parsing table.
8. Construction of a Predictive parsing Table for a given CFG.
9. Implementation of Desktop Calculator using LEX and YACC tools.
10. Implementation of Code Generation for simple expressions.
11. Implementation of simple code optimization techniques.

Step To Execute a Program in Linux

```
vi filename.l
lex filename.l
gcc lex.yy.c -ll
./a.out
```

YACC Execution:

```
vi program.l
vi program.y
lex program.l
yacc -d program.y
cc lex.yy.c y.tab.c -ll -w
./a.out
```

LAB 1. Implementation of Lexical Analyzer to recognize a few patterns in C. (Ex. identifiers, constants, comments,operators etc.)

Lexical Analyser in Python

```
program = '''
'''
keywords = {'auto', 'break', 'case', 'char', 'const', 'continue', 'default', 'do', 'double' 'else', 'enum',
'extern', 'float', 'for', 'goto', 'if', 'int', 'long', 'register', 'return', 'short', 'signed', 'sizeof', 'static',
'static', 'struct', 'switch', 'typedef', 'union', 'unsigned', 'void', 'volatile', 'while'}
operators = {'!', '<', '>', '=', '+', '-', '/', '*', '%'}
delimiters = {';', ',', '\n', '\t', ' '}
brackets = {'(', ')', '{', '}', '[', ']'}
names = {}
for i in operators:
	names[i] = 'operator'
for i in keywords:
	names[i] = 'keyword'
for i in brackets:
	names[i] = 'bracket'
for i in delimiters:
	names[i] = 'delimiter'
i = 0
unnamed_tokens = set()
tokens = {}
token = ""
while(i < len(program)):
	if program[i] in names:
	tokens[program[i]] = names[program[i]]
	if token in keywords:
	tokens[token] =
	'keyword'
	else:
	tokens[token] = 'identifier'
	token = ""
	i += 1
	continue
```

```
        token += program[i]
        i += 1
for token in tokens:
        print(token + " " + tokens[token])
```

LAB 2. Implementation of Lexical Analyzer using LEX tool.

2A.Lexical Analyser in LEX for Digits,Tokens.

```
%{
#include <stdio.h>
%}
DIGIT [0-9]
DIGITS {DIGIT}+
OPTIONAL_FRACTION ([.]{DIGITS})?
OPTIONAL_EXPONENT ([Ee][+-]?{DIGITS})?
NUMBER {DIGITS}{OPTIONAL_FRACTION}{OPTIONAL_EXPONENT}
LETTER [a-zA-Z]
IDENTIFIER {LETTER}({LETTER}|{DIGIT})*
KEYWORD
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|while|volatile
DELIMITER [;:\t\n,]
LPARENTHESIS "("
RPARENTHESIS ")"
NON_IDENTIFIER {NUMBER}+{LETTER}+
AOPERATOR "+"|"-"|"*"|"/"
ASSIGNOP =
BLOCK_BEGINS "{"
BLOCK_ENDS "}"
ROPERATOR >|<|>=|<=|==
LITERAL \".*\"
COMMENT \/\*.*\*\/
PREPROCESSOR_DIRECTIVE #.*
%%
{DIGIT} { printf("%s is a digit\n", yytext); }
{NUMBER} { printf("%s is a number\n", yytext); }
{DELIMITER} { printf("%s is a delimiter\n", yytext); }
{KEYWORD} { printf("%s is a keyword\n", yytext); }
{NON_IDENTIFIER} {printf("Could not process %s", yytext); }
{ASSIGNOP} { printf("%s is an Assignment Operator\n", yytext); }
{IDENTIFIER} { printf("%s is an identifier\n", yytext); }
{AOPERATOR} { printf("%s is an arithmetic operator\n", yytext); }
```

```
{ROPERATOR} { printf("%s is a logical operator\n", yytext); }
{BLOCK_BEGINS} { printf("%s Block begins\n", yytext); }
{BLOCK_ENDS} { printf("%s Block ends\n", yytext); }
{LPARENTHESIS} { printf("%s is a Left Parenthesis\n", yytext); }
{RPARENTHESIS} { printf("%s is a Right Parenthesis\n", yytext); }
{LITERAL} {printf("%s is a Literal\n",yytext);}
{COMMENT} {printf("%s is a Comment\n",yytext);}
{PREPROCESSOR_DIRECTIVE} {printf("%s is a Preprocessor Directive\n",yytext);}
%%
main()
{
	yylex();
	return 0;
}
yywrap()
{
	return 0;
}
```


LAB 2B Lex Programs
-
Write a LEX program to recognize string of a's

```
%%
[a]+     {printf("String with a's is recognized");}
.*     {printf("String of a's is not recognized");}
%%
```


Write a LEX program to recognize an integer number

```
number [+-]?[0-9]+
%%
{number}    {printf("Integer number is recognized");}
.*    {printf("Integer number is not recognized");}
%%
```


Write a LEX program to recognize float number

```
floatnum [0-9]+[.][0-9]+
%%
{floatnum}    {printf("Float number is recognized");}
.*    {printf("Float number is not recognized");}
%%
```


Write a LEX program to print the type of input string (word, number, string)

```
word [a-zA-Z]+
number [0-9]+
string [0-9 a-z A-Z]+
%%
{word}     {printf("%s is a word", yytext);}
```

```
{number}     {printf("%s is a number", yytext);}
{string}     {printf("%s is a string", yytext);}
%%
```


Write a LEX program for identifying an identifier.

```
letter [a-z]
digit [0-9]
id {letter}({letter}|{digit})*
%%
{id} {printf("%s is an identifier", yytext);}
.*    {printf("%s is not an identifier", yytext);}
%%
```


LEX program to accept strings ending with 00 over an alphabet {0,1}

```
%%
(0|1)*00 {printf("String is accepted");}
.*    {printf("String is not accepted");}
%%
```


LEX program to recognize an unsigned number

```
digit [0-9]
digits {digit}+
optional_fraction ([.]{digits})?
optional_exponent ([eE][+-]?{digits})?
num {digits}{optional_fraction}{optional_exponent}
%%
{num} {printf("Unsigned number is recognized");}
.*    {printf("Unsigne number is not recognized");}
%%
```

LEX Program to recognize the numbers which has 1 in its 6th position from right

Program:

```
%%
[0-9]*1[0-9]{5} {printf("Given String is accepted");}
.*    {printf("Given String is not accepted");}
%%
```

LEX program to replace one string with another string

```
%%
"ENGG"    {printf("ENGINEERING");}
"IT"    {printf("INFORMATION TECHNOLOGY");}
"VCE"    {printf("VASAVI COLLEGE OF ENGINEERING");}
%%
```

LEX program to print the length of the input string

```
string [a-zA-Z0-9]+
%%
{string}  {printf("\n The length of the string is %d", yyleng);}
%%
```

LEX program to covert upper case string into lower case

```
string [a-zA-Z]+
%%
{string} {
   int i;
   for(i=0;i<yyleng;i++)
          yytext[i]=tolower(yytext[i]);
   printf("Lower case string  is: %s",yytext);
}
```

```
%%
main() {
   printf("Enter the upper case string: ");
   yylex();
}
```

LEX program to covert lower case string into upper case

```
string [a-zA-Z]+
%%
{string} {
   int i;
   for(i=0;i<yyleng;i++)
         yytext[i]=toupper(yytext[i]);
   printf("Upper case string  is: %s",yytext);
}
%%
main() {
   printf("Enter the lower case string: ");
   yylex();
}
```

A LEX program to add '3' to the given input if the input number being a divisor of 7

```
%{
   int i=0;
%}
%%
[0-9]+  {
   i = atoi(yytext);
   if (i%7==0)
     printf("The number after adding 3 is: %d", i+3);
   else
         printf("The number is not divisible by 7");
}
%%
```

LEX program to print given string in reverse

```
%{
   int i;
%}
%%
[a-z]+  {
   for(i=yyleng-1;i>=0;i--)
   printf ("%c", yytext[i]);
}
%%
```

LEX program to count number of vowels and consonants

```
%{
   int vowels=0;
   int cons=0;
%}
%%
[aeiou] {vowels++;}
[bcdfghjklmnpqrstvwxyz] {cons++;}
%%
int yywrap() {
   return 1;
}
main() {
   printf("Enter the string.. at the end press Ctrl+d\n");
   yylex();
   printf("No. of vowels=%d\n No. of consonants=%d\n", vowels,cons);
}
```

LEX program for Deleting a comment line

```
comment "/*"([a-z A-Z0-9]|[ ])*"*/"
%%
{comment}    {printf(" ");}
%%
```

Write a LEX program that reads an input from a file and that prints the number of words, number of characters and number of spaces for the given input.

```
%{
   int tchar=0,tword=0,tspace=0;
%}
%%
" " {tspace++;tword++;}
[\t\n] tword++;
[^\n\t] tchar++;
%%
int yywrap() {
   return 1;
}
int main() {
   yyin=fopen("input.txt","r");
   yylex();
   printf("Number of characters:: %d\nNumber of words:: %d\nNumber of spaces::
%d\n",tchar,tword ,tspace);
   return 0;
}
```

LEX program to print Successor of a character/string.

```
%{
   int i,l,m;
%}
```

```
%%
[a-z]+    {
    for(i=0;i<yyleng;i++) {
            l=yytext[i];
            if(l=='z')
                    l='a';
            else
            l++;
            printf("%c",l);

    }
}
%%
```

LEX program to print predecessor of a character/string.

```
%{
    int i,l,m;
%}
%%
[a-z]+    {
    for(i=0;i<yyleng;i++) {
            l=yytext[i];
            if(l=='a')
                    l='z';
            else
                    l--;
            printf("%c",l);
    }
}
%%
```

LAB 3. Implementation of Recursive Descent Parser.

RDP

```
#include<stdio.h>
#include<string.h>
void E(),E1(),T(),T1(),F();
int ip=0;

static char s[10];
void main() {
	char k;
	int l;
	ip=0;
	printf("Enter the string:\n");
	scanf("%s",s);
	E();
	if(s[ip]=='$' && strlen(s)>1 && s[ip+1]=='\0')
	printf("\nString is accepted.\nString Length - %d\n",strlen(s)-1);
	else
	printf("\nString not accepted.\n");
}
void E() {
	T();
	E1();
	return;
}
void E1() {
	if(s[ip]=='+') {
	ip++;
	T();
	E1();
	}
	return;
}
```

```
void T() {
	F();
	T1();
	return;
}
void T1() {
	if(s[ip]=='*') {
	ip++;
	F();
	T1();
	}
	return;
}
void F() {
	if(s[ip]=='(') {
	ip++;
	E();
	if(s[ip]==')')
	ip++;
	else
	printf("Syntax Error");
	}
	else if(s[ip]=='i')
	ip++;
	else
	exit(1);
	return;
}
```


LAB 4. Implementation of FIRST() of a given Context-Free Grammar.

5. Implementation of FOLLOW() of a given Context-Free Grammar.

FIRST AND FOLLOW

```
import sys
sys.setrecursionlimit(60)

def first(string):
    #print("first({})".format(string))
    first_ = set()
    if string in non_terminals:
    alternatives = productions_dict[string]

    for alternative in alternatives:
    first_2 = first(alternative)
    first_ = first_ |first_2

    elif string in terminals:
    first_ = {string}

    elif string=='' or string=='@':
    first_ = {'@'}

    else:
    first_2 = first(string[0])
    if '@' in first_2:
    i = 1
    while '@' in first_2:
        #print("inside while")

        first_ = first_ | (first_2 - {'@'})
        #print('string[i:]=', string[i:])
        if string[i:] in terminals:
        first_ = first_ | {string[i:]}
        break
        elif string[i:] == '':
        first_ = first_ | {'@'}
        break
```

```
                first_2 = first(string[i:])
                first_ = first_ | first_2 - {'@'}
                i += 1
        else:
        first_ = first_ | first_2


        #print("returning for first({})".format(string),first_)
        return  first_


def follow(nT):
        follow_ = set()
        prods = productions_dict.items()
        if nT==starting_symbol:
        follow_ = follow_ | {'$'}
        for nt,rhs in prods:
        #print("nt to rhs", nt,rhs)
        for alt in rhs:
        for char in alt:
                if char==nT:
                following_str = alt[alt.index(char) + 1:]
                if following_str=='':
                if nt==nT:
                        continue
                else:
                        follow_ = follow_ | follow(nt)
                else:
                follow_2 = first(following_str)
                if '@' in follow_2:
                        follow_ = follow_ | follow_2-{'@'}
                        follow_ = follow_ | follow(nt)
                else:
                        follow_ = follow_ | follow_2
        #print("returning for follow({})".format(nT),follow_)
        return follow_
```

```
no_of_terminals=int(input("Enter no. of terminals: "))

terminals = []

print("Enter the terminals :")
for _ in range(no_of_terminals):
	terminals.append(input())

no_of_non_terminals=int(input("Enter no. of non terminals: "))

non_terminals = []

print("Enter the non terminals :")
for _ in range(no_of_non_terminals):
	non_terminals.append(input())

starting_symbol = input("Enter the starting symbol: ")

no_of_productions = int(input("Enter no of productions: "))

productions = []

print("Enter the productions:")
for _ in range(no_of_productions):
	productions.append(input())


#print("terminals", terminals)

#print("non terminals", non_terminals)

#print("productions",productions)


productions_dict = {}

for nT in non_terminals:
	productions_dict[nT] = []
```

```
#print(32"productions_dict",productions_dict)

for production in productions:
        nonterm_to_prod = production.split("->")
        alternatives = nonterm_to_prod[1].split("/")
        for alternative in alternatives:
        productions_dict[nonterm_to_prod[0]].append(alternative)

#print("productions_dict",productions_dict)

#print("nonterm_to_prod",nonterm_to_prod)
#print("alternatives",alternatives)


FIRST = {}
FOLLOW = {}

for non_terminal in non_terminals:
        FIRST[non_terminal] = set()

for non_terminal in non_terminals:
        FOLLOW[non_terminal] = set()

#print("FIRST",FIRST)

for non_terminal in non_terminals:
        FIRST[non_terminal] = FIRST[non_terminal] | first(non_terminal)

#print("FIRST",FIRST)


FOLLOW[starting_symbol] = FOLLOW[starting_symbol] | {'$'}
for non_terminal in non_terminals:
        FOLLOW[non_terminal] = FOLLOW[non_terminal] | follow(non_terminal)

#print("FOLLOW", FOLLOW)

print("{: ^20}{: ^20}{: ^20}".format('Non Terminals','First','Follow'))
for non_terminal in non_terminals:
```

```
    print("{: ^20}{: ^20}{: ^20}".format(non_terminal,str(FIRST[non_terminal]),str(FOLLOW[non_terminal])))
```


LAB 6. Construction of a LL(1) for a given CFG.

LL1

```
import re
import string
import pandas as pd


def parse(user_input,start_symbol,parsingTable):

    #flag
    flag = 0

    #appending dollar to end of input
    user_input = user_input + "$"

    stack = []

    stack.append("$")
    stack.append(start_symbol)

    input_len = len(user_input)
    index = 0


    while len(stack) > 0:

        #element at top of stack
        top = stack[len(stack)-1]

        print ("Top =>",top)

        #current input
        current_input = user_input[index]

        print ("Current_Input => ",current_input)

        if top == current_input:
```

```
            stack.pop()
            index = index + 1
        else:

            #finding value for key in table
            key = top , current_input
            print (key)

            #top of stack terminal => not accepted
            if key not in parsingTable:
                flag = 1
                break

            value = parsingTable[key]
            if value !='@':
                value = value[::-1]
                value = list(value)

                #poping top of stack
                stack.pop()

                #push value chars to stack
                for element in value:
                    stack.append(element)
            else:
                stack.pop()

    if flag == 0:
        print ("String accepted!")
    else:
        print ("String not accepted!")



def ll1(follow, productions):

    print ("\nParsing Table\n")

    table = {}
    for key in productions:
```

```
        for value in productions[key]:
            if value!='@':
                for element in first(value, productions):
                    table[key, element] = value
            else:
                for element in follow[key]:
                    table[key, element] = value

    for key,val in table.items():
        print (key,"=>",val)

    new_table = {}
    for pair in table:
        new_table[pair[1]] = {}

    for pair in table:
        new_table[pair[1]][pair[0]] = table[pair]


    print ("\n")
    print( "\nParsing Table in matrix form\n")
    print (pd.DataFrame(new_table).fillna('-'))
    print ("\n")

    return table

def follow(s, productions, ans):
    if len(s)!=1 :
        return {}

    for key in productions:
        for value in productions[key]:
            f = value.find(s)
            if f!=-1:
                if f==(len(value)-1):
                    if key!=s:
                        if key in ans:
                            temp = ans[key]
                        else:
                            ans = follow(key, productions, ans)
```

```
                        temp = ans[key]
                    ans[s] = ans[s].union(temp)
            else:
                first_of_next = first(value[f+1:], productions)
                if '@' in first_of_next:
                    if key!=s:
                        if key in ans:
                            temp = ans[key]
                        else:
                            ans = follow(key, productions, ans)
                            temp = ans[key]
                        ans[s] = ans[s].union(temp)
                        ans[s] = ans[s].union(first_of_next) - {'@'}
                else:
                    ans[s] = ans[s].union(first_of_next)
    return ans

def first(s, productions):
    c = s[0]
    ans = set()
    if c.isupper():
        for st in productions[c]:
            if st == '@' :
                if len(s)!=1 :
                    ans = ans.union( first(s[1:], productions) )
                else :
                    ans = ans.union('@')
            else :
                f = first(st, productions)
                ans = ans.union(x for x in f)
    else:
        ans = ans.union(c)
    return ans

if __name__=="__main__":
    productions=dict()
    grammar = open("grammar2", "r")
    first_dict = dict()
    follow_dict = dict()
    flag = 1
```

```
start = ""
for line in grammar:
        l = re.split("( |->|\n|\||)*", line)
        lhs = l[0]
        rhs = set(l[1:-1])-{''}
        if flag :
                flag = 0
                start = lhs
        productions[lhs] = rhs

print ('\nFirst\n')
for lhs in productions:
        first_dict[lhs] = first(lhs, productions)
for f in first_dict:
        print (str(f) + " : " + str(first_dict[f]))
print ("")

print ('\nFollow\n')

for lhs in productions:
        follow_dict[lhs] = set()

follow_dict[start] = follow_dict[start].union('$')

for lhs in productions:
        follow_dict = follow(lhs, productions, follow_dict)

for lhs in productions:
        follow_dict = follow(lhs, productions, follow_dict)

for f in follow_dict:
        print (str(f) + " : " + str(follow_dict[f]))

ll1Table = ll1(follow_dict, productions)

#parse("a*(a+a)",start,ll1Table)
parse("ba=a+23",start,ll1Table)

# tp(ll1Table)
```


LAB 7. Write a program for generating derivation sequence for a given terminal string using SLR parsing table.

SLR Parser

```
 # SLR(1)

import copy

# perform grammar augmentation
def grammarAugmentation(rules, nonterm_userdef,
                                        start_symbol):

    # newRules stores processed output rules
    newRules = []

    # create unique 'symbol' to
    # - represent new start symbol
    newChar = start_symbol + "'"
    while (newChar in nonterm_userdef):
        newChar += "'"

    # adding rule to bring start symbol to RHS
    newRules.append([newChar,
                                ['.', start_symbol]])

    # new format => [LHS,[.RHS]],
    # can't use dictionary since
    # - duplicate keys can be there
    for rule in rules:

        # split LHS from RHS
        k = rule.split("->")
        lhs = k[0].strip()
        rhs = k[1].strip()

        # split all rule at '|'
        # keep single derivation in one rule
```

```
        multirhs = rhs.split('|')
        for rhs1 in multirhs:
            rhs1 = rhs1.strip().split()

            # ADD dot pointer at start of RHS
            rhs1.insert(0, '.')
            newRules.append([lhs, rhs1])
    return newRules


# find closure
def findClosure(input_state, dotSymbol):
    global start_symbol, \
        separatedRulesList, \
        statesDict

    # closureSet stores processed output
    closureSet = []

    # if findClosure is called for
    # - 1st time i.e. for I0,
    # then LHS is received in "dotSymbol",
    # add all rules starting with
    # - LHS symbol to closureSet
    if dotSymbol == start_symbol:
        for rule in separatedRulesList:
            if rule[0] == dotSymbol:
                closureSet.append(rule)
    else:
        # for any higher state than I0,
        # set initial state as
        # - received input_state
        closureSet = input_state

    # iterate till new states are
    # - getting added in closureSet
    prevLen = -1
    while prevLen != len(closureSet):
        prevLen = len(closureSet)
```

```
        # "tempClosureSet" - used to eliminate
        # concurrent modification error
        tempClosureSet = []

        # if dot pointing at new symbol,
        # add corresponding rules to tempClosure
        for rule in closureSet:
            indexOfDot = rule[1].index('.')
            if rule[1][-1] != '.':
                dotPointsHere = rule[1][indexOfDot + 1]
                for in_rule in separatedRulesList:
                    if dotPointsHere == in_rule[0] and \
                            in_rule not in tempClosureSet:
                        tempClosureSet.append(in_rule)

        # add new closure rules to closureSet
        for rule in tempClosureSet:
            if rule not in closureSet:
                closureSet.append(rule)
    return closureSet


def compute_GOTO(state):
    global statesDict, stateCount

    # find all symbols on which we need to
    # make function call - GOTO
    generateStatesFor = []
    for rule in statesDict[state]:
        # if rule is not "Handle"
        if rule[1][-1] != '.':
            indexOfDot = rule[1].index('.')
            dotPointsHere = rule[1][indexOfDot + 1]
            if dotPointsHere not in generateStatesFor:
                generateStatesFor.append(dotPointsHere)

    # call GOTO iteratively on all symbols pointed by dot
    if len(generateStatesFor) != 0:
        for symbol in generateStatesFor:
            GOTO(state, symbol)
```

```
    return


def GOTO(state, charNextToDot):
    global statesDict, stateCount, stateMap

    # newState - stores processed new state
    newState = []
    for rule in statesDict[state]:
        indexOfDot = rule[1].index('.')
        if rule[1][-1] != '.':
            if rule[1][indexOfDot + 1] == \
                    charNextToDot:
                # swapping element with dot,
                # to perform shift operation
                shiftedRule = copy.deepcopy(rule)
                shiftedRule[1][indexOfDot] = \
                    shiftedRule[1][indexOfDot + 1]
                shiftedRule[1][indexOfDot + 1] = '.'
                newState.append(shiftedRule)

    # add closure rules for newState
    # call findClosure function iteratively
    # - on all existing rules in newState

    # addClosureRules - is used to store
    # new rules temporarily,
    # to prevent concurrent modification error
    addClosureRules = []
    for rule in newState:
        indexDot = rule[1].index('.')
        # check that rule is not "Handle"
        if rule[1][-1] != '.':
            closureRes = \
                findClosure(newState, rule[1][indexDot + 1])
            for rule in closureRes:
                if rule not in addClosureRules \
                        and rule not in newState:
                    addClosureRules.append(rule)
```

```
    # add closure result to newState
    for rule in addClosureRules:
        newState.append(rule)

    # find if newState already present
    # in Dictionary
    stateExists = -1
    for state_num in statesDict:
        if statesDict[state_num] == newState:
            stateExists = state_num
            break

    # stateMap is a mapping of GOTO with
    # its output states
    if stateExists == -1:

        # if newState is not in dictionary,
        # then create new state
        stateCount += 1
        statesDict[stateCount] = newState
        stateMap[(state, charNextToDot)] = stateCount
    else:

        # if state repetition found,
        # assign that previous state number
        stateMap[(state, charNextToDot)] = stateExists
    return


def generateStates(statesDict):
    prev_len = -1
    called_GOTO_on = []

    # run loop till new states are getting added
    while (len(statesDict) != prev_len):
        prev_len = len(statesDict)
        keys = list(statesDict.keys())

        # make compute_GOTO function call
        # on all states in dictionary
```

```
        for key in keys:
            if key not in called_GOTO_on:
                called_GOTO_on.append(key)
                compute_GOTO(key)
    return

# calculation of first
# epsilon is denoted by '#' (semi-colon)

# pass rule in first function
def first(rule):
    global rules, nonterm_userdef, \
        term_userdef, diction, firsts

    # recursion base condition
    # (for terminal or epsilon)
    if len(rule) != 0 and (rule is not None):
        if rule[0] in term_userdef:
            return rule[0]
        elif rule[0] == '#':
            return '#'

    # condition for Non-Terminals
    if len(rule) != 0:
        if rule[0] in list(diction.keys()):

            # fres temporary list of result
            fres = []
            rhs_rules = diction[rule[0]]

            # call first on each rule of RHS
            # fetched (& take union)
            for itr in rhs_rules:
                indivRes = first(itr)
                if type(indivRes) is list:
                    for i in indivRes:
                        fres.append(i)
                else:
                    fres.append(indivRes)
```

```
            # if no epsilon in result
            # - received return fres
            if '#' not in fres:
                return fres
            else:

                # apply epsilon
                # rule => f(ABC)=f(A)-{e} U f(BC)
                newList = []
                fres.remove('#')
                if len(rule) > 1:
                    ansNew = first(rule[1:])
                    if ansNew != None:
                        if type(ansNew) is list:
                            newList = fres + ansNew
                        else:
                            newList = fres + [ansNew]
                    else:
                        newList = fres
                    return newList

                # if result is not already returned
                # - control reaches here
                # lastly if eplison still persists
                # - keep it in result of first
                fres.append('#')
                return fres


# calculation of follow
def follow(nt):
    global start_symbol, rules, nonterm_userdef, \
        term_userdef, diction, firsts, follows

    # for start symbol return $ (recursion base case)
    solset = set()
    if nt == start_symbol:
        # return '$'
        solset.add('$')
```

```
# check all occurrences
# solset - is result of computed 'follow' so far

# For input, check in all rules
for curNT in diction:
    rhs = diction[curNT]

    # go for all productions of NT
    for subrule in rhs:
        if nt in subrule:

            # call for all occurrences on
            # - non-terminal in subrule
            while nt in subrule:
                index_nt = subrule.index(nt)
                subrule = subrule[index_nt + 1:]

                # empty condition - call follow on LHS
                if len(subrule) != 0:

                    # compute first if symbols on
                    # - RHS of target Non-Terminal exists
                    res = first(subrule)

                    # if epsilon in result apply rule
                    # - (A->aBX)- follow of -
                    # - follow(B)=(first(X)-{ep}) U follow(A)
                    if '#' in res:
                        newList = []
                        res.remove('#')
                        ansNew = follow(curNT)
                        if ansNew != None:
                            if type(ansNew) is list:
                                newList = res + ansNew
                            else:
                                newList = res + [ansNew]
                        else:
                            newList = res
                        res = newList
                else:
```

```
                        # when nothing in RHS, go circular
                        # - and take follow of LHS
                        # only if (NT in LHS)!=curNT
                        if nt != curNT:
                            res = follow(curNT)

                    # add follow result in set form
                    if res is not None:
                        if type(res) is list:
                            for g in res:
                                solset.add(g)
                        else:
                            solset.add(res)
    return list(solset)


def createParseTable(statesDict, stateMap, T, NT):
    global separatedRulesList, diction

    # create rows and cols
    rows = list(statesDict.keys())
    cols = T+['$']+NT

    # create empty table
    Table = []
    tempRow = []
    for y in range(len(cols)):
        tempRow.append('')
    for x in range(len(rows)):
        Table.append(copy.deepcopy(tempRow))

    # make shift and GOTO entries in table
    for entry in stateMap:
        state = entry[0]
        symbol = entry[1]
        # get index
        a = rows.index(state)
        b = cols.index(symbol)
        if symbol in NT:
```

```
            Table[a][b] = Table[a][b]\
                + f"{stateMap[entry]} "
        elif symbol in T:
            Table[a][b] = Table[a][b]\
                + f"S{stateMap[entry]} "

# start REDUCE procedure

# number the separated rules
numbered = {}
key_count = 0
for rule in separatedRulesList:
    tempRule = copy.deepcopy(rule)
    tempRule[1].remove('.')
    numbered[key_count] = tempRule
    key_count += 1

# start REDUCE procedure
# format for follow computation
addedR = f"{separatedRulesList[0][0]} -> " \
    f"{separatedRulesList[0][1][1]}"
rules.insert(0, addedR)
for rule in rules:
    k = rule.split("->")

    # remove un-necessary spaces
    k[0] = k[0].strip()
    k[1] = k[1].strip()
    rhs = k[1]
    multirhs = rhs.split('|')

    # remove un-necessary spaces
    for i in range(len(multirhs)):
        multirhs[i] = multirhs[i].strip()
        multirhs[i] = multirhs[i].split()
    diction[k[0]] = multirhs

# find 'handle' items and calculate follow.
for stateno in statesDict:
    for rule in statesDict[stateno]:
```

```
                    if rule[1][-1] == '.':

                        # match the item
                        temp2 = copy.deepcopy(rule)
                        temp2[1].remove('.')
                        for key in numbered:
                            if numbered[key] == temp2:

                                # put Rn in those ACTION symbol columns,
                                # who are in the follow of
                                # LHS of current Item.
                                follow_result = follow(rule[0])
                                for col in follow_result:
                                    index = cols.index(col)
                                    if key == 0:
                                        Table[stateno][index] = "Accept"
                                    else:
                                        Table[stateno][index] =\
                                            Table[stateno][index]+f"R{key} "

    # printing table
    print("\nSLR(1) parsing table:\n")
    frmt = "{:>8}" * len(cols)
    print(" ", frmt.format(*cols), "\n")
    ptr = 0
    j = 0
    for y in Table:
        frmt1 = "{:>8}" * len(y)
        print(f"{{:>3}} {frmt1.format(*y)}"
              .format('I'+str(j)))
        j += 1

def printResult(rules):
    for rule in rules:
        print(f"{rule[0]} ->"
              f" {' '.join(rule[1])}")

def printAllGOTO(diction):
    for itr in diction:
        print(f"GOTO ( I{itr[0]} ,"
```

```
                    f" {itr[1]} ) = I{stateMap[itr]}")

# *** MAIN *** - Driver Code

# uncomment any rules set to test code
# follow given format to add -
# user defined grammar rule set
# rules section - *START*

# example sample set 01
rules = ["E -> E + T | T",
         "T -> T * F | F",
         "F -> ( E ) | id"
         ]
nonterm_userdef = ['E', 'T', 'F']
term_userdef = ['id', '+', '*', '(', ')']
start_symbol = nonterm_userdef[0]

# example sample set 02
# rules = ["S -> a X d | b Y d | a Y e | b X e",
#          "X -> c",
#          "Y -> c"
#          ]
# nonterm_userdef = ['S','X','Y']
# term_userdef = ['a','b','c','d','e']
# start_symbol = nonterm_userdef[0]

# rules section - *END*
print("\nOriginal grammar input:\n")
for y in rules:
   print(y)

# print processed rules
print("\nGrammar after Augmentation: \n")
separatedRulesList = \
   grammarAugmentation(rules,
                       nonterm_userdef,
                       start_symbol)
printResult(separatedRulesList)
```

```
# find closure
start_symbol = separatedRulesList[0][0]
print("\nCalculated closure: I0\n")
I0 = findClosure(0, start_symbol)
printResult(I0)

# use statesDict to store the states
# use stateMap to store GOTOs
statesDict = {}
stateMap = {}

# add first state to statesDict
# and maintain stateCount
# - for newState generation
statesDict[0] = I0
stateCount = 0

# computing states by GOTO
generateStates(statesDict)

# print goto states
print("\nStates Generated: \n")
for st in statesDict:
    print(f"State = I{st}")
    printResult(statesDict[st])
    print()

print("Result of GOTO computation:\n")
printAllGOTO(stateMap)

# "follow computation" for making REDUCE entries
diction = {}

# call createParseTable function
createParseTable(statesDict, stateMap,
                             Term_userdef,
```


****
****
****
****
****
****

YACC Execution:

vi program.l

vi program.y

lex program.l

yacc -d program.y

cc lex.yy.c y.tab.c -ll -w

./a.out

****
****
****
****
****
****

LAB 8. Construction of a Predictive parsing Table for a given CFG.

Predictive Parsing

```
import sys
sys.setrecursionlimit(60)

def First(string):
	first = set()
	if string in non_terminals:
	RHS = pdict[string]
	for i in RHS:
	first2 = First(i)
	first = first |first2
	elif string in terminals:
	first = {string}
	elif string=='' or string=='@':
	first = {'@'}
	else:
	first2 = First(string[0])
	if '@' in first2:
	i = 1
	while '@' in first2:
		first = first | (first2 - {'@'})
		if string[i:] in terminals:
		first = first | {string[i:]}
		break
		elif string[i:] == '':
		first = first | {'@'}
		break
		first2 = First(string[i:])
		first = first | first2 - {'@'}
		i += 1
	else:
	first = first | first2
	return  first

def Follow(nT):
	follow = set()
```

```
	prods = pdict.items()
	if nT == start:
	follow = follow | {'$'}
	for nt, rhs in prods:
	for alt in rhs:
	for char in range(len(alt)):
		if alt[char] == nT:
		following_str = alt[char + 1:]
		if following_str == '':
		if nt == nT:
			continue
		else:
			follow = follow | Follow(nt)
		else:
		follow2 = First(following_str)
		if '@' in follow2:
			follow = follow | follow2-{'@'}
			follow = follow | Follow(nt)
		else:
			follow = follow | follow2
	return follow

n = int(input("Enter no. of terminals: "))
terminals = []
print("Enter the terminals :")
for i in range(n):
	terminals.append(input())
n = int(input("Enter no. of non terminals: "))
non_terminals = []
print("Enter the non terminals :")
for i in range(n):
	non_terminals.append(input())
start = input("Enter the starting symbol: ")
n = int(input("Enter no of productions: "))
productions = []
print("Enter the productions:")
for i in range(n):
	productions.append(input())
pdict = {}
for nT in non_terminals:
```

```
        pdict[nT] = []
for p in productions:
        prod = p.split("->")
        RHS = prod[1].split("/")
        for i in RHS:
        pdict[prod[0]].append(i)
terminals.append('$')
PPT = dict()
for non_terminal in non_terminals:
        PPT[non_terminal] = dict()
        for terminal in terminals:
        PPT[non_terminal][terminal] = ""

for non_terminal, productions in pdict.items():
        for production in productions:
        first = First(production)
        for terminal in first:
        if terminal != '@':
                PPT[non_terminal][terminal] = non_terminal + '->' + production
        if '@' in first:
        follow = Follow(non_terminal)
        if '$' in follow:
                PPT[non_terminal]['$'] = non_terminal + '->' + production
        for terminal in follow:
                PPT[non_terminal][terminal] = non_terminal + '->' + '@'

print("\t", end = '')
for terminal in terminals:
        print(terminal + '\t', end = ' ')
print()
for non_terminal in non_terminals:
        print(non_terminal, end = "\t")
        for terminal in terminals:
        print(PPT[non_terminal][terminal], end = "\t")
        print()
```

LAB 9 A. Implementation of Desktop Calculator using LEX and YACC tools.
B. Implementation of LEX and YACC tools.

YACC Programs

(1) Simple Desktop Calculator

vi lex.l

```
%{
        #include<stdio.h>
        #include "y.tab.h"

        int yylval;

%}



%%
[0-9]+ {
        yylval = atoi(yytext);
        return DIGIT;
}
"\n"|. return yytext[0];
%%
```

vi lex.y

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

        int yylex();
        void yyerror(char* msg);
%}

%name parse
%token DIGIT

%%
```

```
        L:E'\n' {printf("%d\n", $1);}
        ;

        E:E'+'T {$$=$1+$3;}
        |T
        ;

        T:T'*'F {$$=$1*$3;}
        |F
        ;

        F:'('E')' {$$=$2;}
        |DIGIT
        ;
%%

int main() {
        yyparse();
}

void yyerror(char* msg) {
        printf("%s\n", msg);
}
```

---

(2) Strings {a^nb^n | n >= 0}

vi str_an_bn.l

```
%{
        #include "y.tab.h"
%}



%%
a {return A;}
b {return B;}
"\n"|. {return yytext[0];}
%%
```

```
*vi str_an_bn.y*
%{
	#include <stdio.h>

	int yylex();
	void yyerror(char* msg);
%}

%name parse
%token A B

%%
	stmt:S '\n' {printf("Valid String\n");}
	;
	S:A S B
	|
	;
%%

int main() {
	yyparse();
}

void yyerror(char* msg) {
	printf("%s\n", msg);
}
```

---

(3) Strings {wcw^r}

```
*vi wcwr.l*
%{
	#include <stdio.h>
	#include "y.tab.h"
%}

%%
a {return A;}
b {return B;}
c {return C;}
"\n"|. {return yytext[0];}
```

```
%%

*vi wcwr.y*
%{
	#include <stdio.h>

	int yylex();
	void yyerror(char* msg);
%}

%name parse
%token A B C

%%
	stmt:S '\n' {printf("Valid String\n");}
	;

	S:A S A
	|B S B
	|C
	;
%%

int main() {
	yyparse();
}

void yyerror(char* msg) {
	printf("%s\n", msg);
}
```

---

(4) Strings {a^nb^na^mb^ma^kb^k----}

```
*vi con_an_bn.l*
%{
	#include <stdio.h>
	#include "y.tab.h"
%}

%%
```

```
a {return A;}
b {return B;}
"\n"|. {return yytext[0];}
%%

*vi con_an_bn.y*
%{
	#include <stdio.h>

	int yylex();
	void yyerror(char* msg);
%}

%name parse
%token A B

%%
	stmt:Q '\n' {printf("Valid String\n");}
	;

	Q:P Q
	|
	;

	P:A P B
	|A B
	;
%%

int main() {
	yyparse();
}

void yyerror(char* msg) {
	printf("%s\n", msg);
}
```

LAB 10. Implementation of Code Generation for simple expressions.

CODE GENERATOR

```
def get_register(var):
    for i in range(10):
    if alive[i] == 0:
    registers[i] = var
    return i


registers = [""]*10
alive = [0]*10
program = '''
c=a+b
a=c
d=e-f
g=h*j
'''
lines = program.split()
ops = {}
ops['+'] = 'ADD'
ops['-'] = 'SUB'
ops['*'] = 'MUL'
ops['/'] = 'DIV'
for exp in lines:
    lhs, rhs = exp.split('=')
    if rhs.isalpha():
    print("MOV " + lhs + "," + rhs)
    continue
    for i in rhs:
    if i in ops:
    op1, op2 = rhs.split(i)
    op = i
    reg = get_register(op1)
```

```
if not alive[reg]:
print("MOV R" + str(reg) + "," + op1)
alive[reg] = 1
print(ops[op], end=" ")
if op2 in registers:
k = registers.index(op2)
print("R" + str(k) + ",R" + str(reg))
alive[k] = 0
else:
print(op2 + ",R" + str(reg))
print("MOV " + lhs + ",R" + str(reg))
```

LAB 11. Implementation of simple code optimization techniques.

CODE OPTIMIZER

```
program = '''
a=b+c
b=a-d
c=b+c
d=a-d
'''
lines = program.split()
exps = []
for i in lines:
        exps.append(i.split('='))
n = len(lines)
for i in range(n):
        temp = exps[i][1]
        if len(temp) <= 1:
        continue
        op1, op2 = temp[0], temp[2]
        for j in range(i + 1, n):
        if exps[j][1] == temp:
        exps[j][1] = exps[i][0]
        if exps[j][0] == op1 or exps[j][0] == op2:
        break
print("After Common Sub-Expression Elimination:")
for exp in exps:
        print(exp[0], "=", exp[1])
output = []
for i in range(n):
        for j in range(i + 1, n):
        if exps[i][0] in exps[j][1]:
        output.append(exps[i])
        break
output.append(exps[-1])
print("After Dead-Code Elimination:")
for exp in output:
        print(exp[0], "=", exp[1])
```