ANNAMALAI UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B. E. (COMPUTER SCIENCE AND ENGINEERING)

VI - SEMESTER

CSCP607 – COMPILER DESIGN LAB

Name : _____

Reg. No.: _____

ANNAMALAI UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B.E. (Computer Science and Engineering)

VI - SEMESTER

CSCP607 - Compiler Design Lab

Certified that this is the bonafide record of work done by Mr./Ms._____

Reg. No._____of B.E. (Computer Science and Engineering) in the CSCP607 – Compiler Design Lab during even semester of the academic year 2022 –2023.

Staff in-charge

Internal Examiner **External Examiner**

Place: Annamalai Nagar
Date: / / 2023.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
B.E. (COMPUTER SCIENCE AND ENGINEERING)

VISION:

To provide a congenial ambience for individuals to develop and blossom as academically superior, socially conscious and nationally responsible citizens.

MISSION:

- Impart high quality computer knowledge to the students through a dynamic scholastic environment wherein they learn to develop technical, communication and leadership skills to bloom as a versatile professional.
- Develop life-long learning ability that allows them to be adaptive and responsive to the changes in career, society, technology, and environment.
- Build student community with high ethical standards to undertake innovative research and development in thrust areas of national and international needs.
- Expose the students to the emerging technological advancements for meeting the demands of the industry.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

PEO	PEO Statements
PEO1	To prepare the graduates with the potential to get employed in the right role and/or become entrepreneurs to contribute to the society.
PEO2	To provide the graduates with the requisite knowledge to pursue higher education and carry out research in the field of Computer Science.
PEO3	To equip the graduates with the skills required to stay motivated and adapt to the dynamically changing world so as to remain successful in their career.
PEO4	To train the graduates to communicate effectively, work collaboratively and exhibit high levels of professionalism and ethical responsibility.

CSCP607	COMPILER DESIGN LAB	L	T	P	C
		0	0	3	1.5

Course Objectives:

- To make the students to understand the different stages of compiler.
- To impart programming skills needed to develop a compiler.

LIST OF EXERCISES
(The exercises are to be done in python language)

1) a) To implement Lexical Analyzer using python code.
   b) To implement Lexical Analyzer using Lexical tool.
2) To convert the given Regular Expression to NFA.
3) To eliminate Left Recursion in a given grammar.
4) To perform Left Factoring in a given grammar.
5) To compute First and Follow sets.
6) To compute Leading and Trailing sets.
7) To construct Predictive Parsing Table.
8) To implement Shift Reduce Parsing.
9) To compute LR(0) items.
10) To generate Intermediate Code.

Course Outcomes:

At the end of this course, the students will be able to

1. Apply the programming knowledge skill to design and develop a compiler.
2. Understand and implement lexical analyzer, syntax analyser, code optimizer and code generator.
3. Demonstrate an ability to listen and answer the viva questions related to programming skills needed for solving real-world problems in Computer Science and Engineering.

Mapping of Course Outcomes with Programme Outcomes												
	PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**
CO1	2	2	-	-	-	-	-	-	-	-	-	-
CO2	2	2	3	3	-	-	-	-	-	-	-	-
CO3	2	2	-	-	-	-	-	-	-	2	-	2

Rubric for CO3

Rubric for CO3 in Laboratory Courses					
Rubric	**Distribution of 10 Marks for CIE/SEE Evaluation Out of 40/60 Marks**				
	Up To 2.5 Marks	**Up To 5 Marks**	**Up To 7.5 Marks**	**Up To 10 marks**	**Up To 2.5 Marks**
Demonstrate an ability to listen and answer the viva questions related to programming skills needed for solving real-world problems in Computer Science and Engineering.	Poor listening and communication skills. Failed to relate the programming skills needed for solving the problem.	Showed better communication skill by relating the problem with the programming skills acquired but the description showed serious errors.	Demonstrated good communication skills by relating the problem with the programming skills acquired with few errors.	Demonstrated excellent communication skills by relating the problem with the programming skills acquired and have been successful in tailoring the description.	Demonstrate an ability to listen and answer the viva questions related to programming skills needed for solving real-world problems in Computer Science and Engineering.

List of Exercises

Ex. No.	Date	Experiments	Page No.	Marks	Staff Signature
1(a).		Implementation of Lexical Analyzer	1		
1(b)		Implementation of Lexical Tool	4		
2		Regular Expression to NFA	7		
3		Elimination of Left Recursion	14		
4		Left Factoring	18		
5		Computation of First and Follow Sets	21		
6		Computation of Leading and Trailing Sets	26		
7		Construction of Predictive Parsing Table	30		
8		Shift Reduce Parsing	36		
9		Computation of LR(0) items	38		
10		Intermediate Code Generation	43		

EX.NO: 1(a) DATE:	Implementation of Lexical Analyzer

Aim:

To implement Lexical Analysis for given example text file using python coding.

Algorithm:

Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High- level input program into a sequence of Tokens. This sequence of tokens is sent to the parser for syntax analysis. Lexical Analysis can be implemented with the Deterministic finite Automata.

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages The types of token are identifier, numbers, operators, keywords, special symbols etc.

Following are the examples of tokens:

Keywords: Examples-for, while, if etc.
Identifier: Examples-Variable name, function name, etc.
Operators: Examples '+', '++', '-' etc.
Separators: Examples ', ' ';' etc.

The algorithm for lexical analysis is as follows:
1. Read the input expression.
2. If the input is an keyword, store it as a keyword.
3. If the input is an operator, store it as operator.
4. If the input is a delimiter, store it as a delimiter.
5. Check whether input is a sequence of alphabets and/or digits then store it an identifier.
6. If the input is a sequence of digits, then store it as a number.

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", "struct", "switch", "typedef", "union",
"unsigned", "void", "volatile", "while", "printf", "scanf", "%d", "include", "stdio.h", "main"}

operators = {"+","-","*","/","<",">","=","<=",">=","==","!=","++","--","%"}

delimiters = {'(',')','{','}','[',']',"'",'"',';','#',',',"}
```

```
        def detect_keywords(text): arr = []
        for word in text:


                if word in keywords:
                        arr.append(word)
        return list(set(arr))

def detect_operators(text):
        arr = []
        for word in text:
                if word in operators:
                        arr.append(word)
        return list(set(arr))

def detect_delimiters(text):
        arr = []
        for word in text:
                if word in delimiters:
                        arr.append(word)
        return list(set(arr))

def detect_num(text):
        arr = []
        for word in text:
                try:
                        a = int(word)
                        arr.append(word)
                except:
                        pass
        return list(set(arr))

def detect_identifiers(text):
        k = detect_keywords(text)
        o = detect_operators(text)
        d = detect_delimiters(text)
        n = detect_num(text)
        not_ident = k + o + d + n
        arr = []
        for word in text:
                if word not in not_ident:
                        arr.append(word)
        return arr

with open('e1-example.txt') as t:
        text = t.read().split()

print ("Keywords: ", detect_keywords(text))
print ("Operators: ", detect_operators(text))
print ("Delimiters: ", detect_delimiters(text))
```

```
print ("Identifiers: ", detect_identifiers(text))
print ("Numbers: ", detect_num(text))
```

Input:

```
# include < stdio.h > // This is a header file
int main ( )
{
   int a ;
   a = 10 ;
   printf ( " The value of a is % d ", a ) ;
   return 0 ;
}
```

Output:

```
Keywords: ['main', 'return', 'int', 'stdio.h', 'include', 'printf']
Operators: ['=', '>', '%', '<']
Delimiters: [')', '#', '"', '}', '{', ';', '(']
Identifiers: ['//', 'This', 'is', 'a', 'header', 'file', 'a', 'a', '(', 'The', 'value', 'of', 'a', 'is', 'd', '",', 'a']
Numbers: ['10', '0']
```

Result:

Thus, the python program to implement the Lexical Analyzer is executed successfully and tested with various samples.

EX.NO: 01 (b) DATE:	Implementation of Lexical Tool

Aim:

To implement Lexical Analyzer using a Lexical Tool.

Algorithm:

1. Define the set of tokens or lexemes for the programming language you want to process. These can be keywords, identifiers, operators, literals, etc. Store them in a dictionary or a list for easy access.
2. Read the input source code file to be processed.
3. Initialize a cursor or pointer to the beginning of the input source code.
4. Create a loop that iterates through the source code, character by character.
5. Implement a finite state machine (FSM) to recognize tokens. The FSM can have states representing different types of tokens, such as "keyword", "identifier", "operator", etc.
6. For each character encountered in the source code, update the FSM state based on the current character and the current state. If the current state is a final state, store the recognized token and reset the FSM state to the initial state. If the current state is not a final state and the current character does not transition to any valid state, then raise a lexical error.
7. Continue the loop until the end of the input source code is reached.
8. Output the recognized tokens along with their corresponding lexeme value or token type.

Program:

```
import ply.lex as lex

# List of token names. This is always required
tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
```

```
)

# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore = ' \t'

# Error handling rule
def t_error(t):
    print ("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()

# Test it out
data = '''
3 + 4 * 10
  + -20 *2
'''

# Give the lexer some input
lexer.input(data)

# Tokenize
while True:
    tok = lexer.token()
    if not tok:
        break      # No more input
```

```
print(tok)
```

Output:

```
LexToken(NUMBER,3,2,1)
LexToken(PLUS,'+',2,3)
LexToken(NUMBER,4,2,5)
LexToken(TIMES,'*',2,7)
LexToken(NUMBER,10,2,10)
LexToken(PLUS,'+',3,14)
LexToken(MINUS,'-',3,16)
LexToken(NUMBER,20,3,18)
LexToken(TIMES,'*',3,20)
LexToken(NUMBER,2,3,21)
```

Result:

Thus, the Lexical Analyzer is implemented using the Lexical tool.

EX.NO: 02 DATE:	Regular Expression to NFA

Aim:

To write a python program to convert the given Regular Expression to NFA.

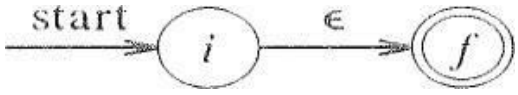
Algorithm:

Thompson's Construction of an NFA from a Regular Expression:

Input: A regular expression r over the alphabet.
Output: An NFA N accepting L(r)

METHOD: Begin by parsing r into its constituent subexpressions. The rules for constructing an NFA consist of the following basics rules.

For expression e construct the NFA,

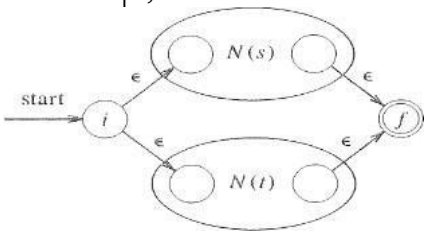


Here, i is a new state, the start state of this NFA, and f is another new state, the accepting state for the NFA.

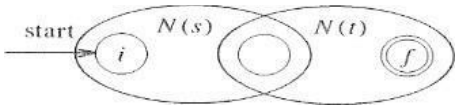
For any subexpressions , construct the NFA,

INDUCTION: Suppose N(s) and N (t) are NFA'sfor regularexpressions s and t, respectively.

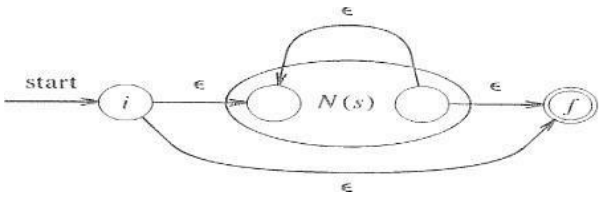
a) For the regular expression s|t ,



b) For the regular expression st,



c) For the regular expression S*,



d) Finally, suppose r = (s), then L(r) = L(s), and we can use the NFA, N(s) as N(r).

Program:

```
inp = "((e+a).b*)*"#input("")

_="-"

start = 1 # denotes start of e-nfa table
end = 1 # denotes end of our table which is initially same as start
cur = 1   # denotes current position of our pointer
table = [["state","epsilon","a","b"],
          [1,_,_,_]]

def print_t(table):
    i = table [0]
    print(f'{i[0]: <10}'+f'| {i[1]: <10}'+f'| {i[2]: <10}'+f'| {i[3]: <10}')
    print("-"*46)
    for i in table[1:]:
        try:
            x = " ".join([str(j) for j in i[1]])
        except:
            x = ""
        try:
            y = " ".join([str(j) for j in i[2]])
        except:
            y = ""
        try:
            z = " ".join([str(j) for j in i[3]])
        except:
            z = ""
        print(f'{i[0]: <10}'+f'| {x: <10}'+f'| {y: <10}'+f'| {z: <10}')

def e_(cur,ed=end):
    temp = table[cur]
```

```
    try:
        table[cur] = [cur,temp[1].append(cur+1),temp[2],temp[3]]
    except:
        table[cur] = [cur,[cur+1],temp[2],temp[3]]
    try:
        nv = table([cur+1])
    except:
        table.append([ed+1,_,_,_])
        ed+=1
    return ed


def a_(cur,ed=end):
    temp = table[cur]
    try:
        table[cur] = [cur,temp[1],temp[2].append(cur+1),temp[3]]
    except:
        table[cur] = [cur,temp[1],[cur+1],temp[3]]
    try:
        nv = table([cur+1])
    except:
        table.append([ed+1,_,_,_])
        ed+=1
    return ed

def b_(cur,ed=end):
    temp = table[cur]
    try:
        table[cur] = [cur,temp[1],temp[2],temp[3].append(cur+1)]
    except:
        table[cur] = [cur,temp[1],temp[2],[cur+1]]
    try:
        nv = table([cur+1])
    except:
        table.append([ed+1,_,_,_])
        ed+=1
    return ed

def or_b(cur,ed=end):
    temp = table[cur]
    try:
        table[cur] = [cur,temp[1],temp[2],temp[3].append(cur+1)]
    except:
        table[cur] = [cur,temp[1],temp[2],[cur+1]]


def or_a(cur,ed=end):
    temp = table[cur]
    try:
        table[cur] = [cur,temp[1],temp[2].append(cur+1),temp[3]]
    except:
        table[cur] = [cur,temp[1],[cur+1],temp[3]]
```

```
def and_a(cur,ed=end):
    cur+=1
    temp = table[cur]
    try:
        table[cur] = [cur,temp[1],temp[2].append(cur+1),temp[3]]
    except:
        table[cur] = [cur,temp[1],[cur+1],temp[3]]
    try:
        nv = table([cur+1])
    except:
        table.append([cur+1,_,_,_])
        ed+=1
    return cur,ed

def and_b(cur,ed=end):
    cur+=1
    temp = table[cur]
    try:
        table[cur] = [cur,temp[1],temp[2],temp[3].append(cur+1)]
    except:
        table[cur] = [cur,temp[1],temp[2],[cur+1]]
    try:
        nv = table([cur+1])
    except:
        table.append([cur+1,_,_,_])
        ed+=1
    return cur,ed

def star(cur,ed=end):
    table.append([ed+1,_,_,_])
    table.append([ed+2,_,_,_])
    ed+=2
    for i in range(cur,ed):
        temp = [table[ed-i+cur][0]]+table[ed-i+cur-1][1:4]
        for j in [1,2,3]:
            try:
                temp[j] = [x+1 for x in table[ed-i+cur-1][j]]
            except:
                pass
        table[ed-i+cur] = temp
    table[cur]=[cur,_,_,_]

    temp = table[cur]
    try:
        table[cur] = [temp[0],temp[1]+[cur+1,ed],temp[2],temp[3]]
    except:
        table[cur] = [temp[0],[cur+1,ed],temp[2],temp[3]]

    temp = table[ed-1]
    try:
        table[ed-1] = [temp[0],temp[1]+[cur+1,ed],temp[2],temp[3]]
```

```
    except:
        table[ed-1] = [temp[0],[cur+1,ed],temp[2],temp[3]]

    return ed-1,ed


def mod_table(inp,start,cur,end,table):
    #print(inp)
    k = 0
    while k<len(inp):
        #print(start,cur,end,k,inp[k:],len(table)-1)
        if inp[k]=="a":
            end = a_(cur,end)
            #print("in a_")
        elif inp[k]=="b":
            end = b_(cur,end)
            #print("in b_")
        elif inp[k]=="e":
            end = e_(cur,end)
        elif inp[k]==".":
            k+=1
            if inp[k]=="a":
                #k-=1
                cur,end = and_a(cur,end)
            elif inp[k]=="b":
                cur,end = and_b(cur,end)
                #k-=1
            elif inp[k]=="(":
                li = ["("]
                l = k
                for i in inp[k+1:]:
                    if i == "(":
                        li.append("(")
                    if i == ")":
                        try:
                            del li[-1]
                        except:
                            break
                    if len(li)==0:
                        break
                    l+=1
                m  =  k
                k=l+1
                cur+=1
                start,cur,end,table = mod_table(inp[m+1:l+1],start,cur,end,table)

        elif inp[k]=="+":
            k+=1
            if inp[k]=="a":
                or_a(cur,end)
                #print("in or_a")
            elif inp[k]=="b":
```

```
            or_b(cur,end)
            #print("in or_b")
        else:
            print(f"ERROR at{k }Done:{inp[:k+1]}Rem{inp[k+1:]}")

    elif inp[k]=="*":
        #print("in star")
        cur,end = star(cur,end)
    elif inp[k]=="(":
        li = ["("]
        l = k
        for i in inp[k+1:]:
            if i == "(":
                li.append("(")
            if i == ")":
                try:
                    del li[-1]
                except:
                    break
            if len(li)==0:
                break

            l+=1
        m = k
        k=l+1
        try:
            if inp[k+1]=="*":
                cur_ = cur
        except:
            pass
        #print(inp[m+1:l+1])
        start,cur,end,table = mod_table(inp[m+1:l+1],start,cur,end,table)
        try:
            if inp[k+1]=="*":
                cur = cur_
        except:
            pass
    else:
        print(f'error{k}{inp[k]}')
    k+=1
return start,cur,end,table



start,cur,end,table = mod_table(inp,start,cur,end,table)

  print_t(table)
```

Output:

state	epsilon	a	b
1	2 7	-	-
2	3	3	-
3	4 6	-	-
4	-	-	5
5	4 6	-	-
6	2 7	-	-
7	-	-	-

Result:

Thus, the python program for construction of NFA table from Regular Expression is executed successfully and tested with various samples.

EX.NO: 03 DATE:	Elimination of Left Recursion

Aim:

To write a python program to implement Elimination of Left Recursion for given sample data.

Algorithm:

A grammar is left recursive if it has a nonterminal A such that there is a derivation A +=> Aα for some string. Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.

1) Arrange the nonterminals in some order A1, A2,……,An.

2) for each i from 1 to n

3) for each j from 1 to i-1

4) replace each production of the form Ai →Ajγ by the productions Ai→δ1γ |δ2γ |…. |δκγ, where Aj→δ1 | δ2 |….. | δκ are all current Aj-productions

5)Eliminate the immediate left recursion among the Ai –

productions.

Program:

```
gram = {}

def add(str):
    str = str.replace(" ", "").replace("    ", "").replace("\n", "")
    x = str.split("->")
    y = x[1]
    x.pop()
    z = y.split("|")
    x.append(z)
    gram[x[0]]=x[1]
```

```
def removeDirectLR(gramA, A):
	temp = gramA[A]
	tempCr = []
	tempInCr = []
	for i in temp:
		if i[0] == A:
			#tempInCr.append(i[1:])
			tempInCr.append(i[1:]+[A+"'"])
		else:
			#tempCr.append(i)
			tempCr.append(i+[A+"'"])
	tempInCr.append(["e"])
	gramA[A] = tempCr
	gramA[A+"'"] = tempInCr
	return gramA


def checkForIndirect(gramA, a, ai):
	if ai not in gramA:
		return False
	if a == ai:
		return True
	for i in gramA[ai]:
		if i[0] == ai:
			return False
		if i[0] in gramA:
			return checkForIndirect(gramA, a, i[0])
	return False

def rep(gramA, A):
	temp = gramA[A]
	newTemp = []
	for i in temp:
		if checkForIndirect(gramA, A, i[0]):
			t = []
			for k in gramA[i[0]]:
				t=[]
				t+=k
				t+=i[1:]
				newTemp.append(t)

		else:
			newTemp.append(i)
	gramA[A] = newTemp
	return gramA
```

```
def rem(gram):
	c = 1
	conv = {}
	gramA = {}
	revconv = {}
	for j in gram:
		conv[j] = "A"+str(c)
		gramA["A"+str(c)] = []
		c+=1

	for i in gram:
		for j in gram[i]:
			temp = []
			for k in j:
				if k in conv:
					temp.append(conv[k])
				else:
					temp.append(k)
			gramA[conv[i]].append(temp)


	#print(gramA)
	for i in range(c-1,0,-1):
		ai =  "A"+str(i)
		for j in range(0,i):
			aj = gramA[ai][0][0]
			if ai!=aj :
				if aj in gramA and checkForIndirect(gramA,ai,aj):
					gramA = rep(gramA, ai)

	for i in range(1,c):
		ai = "A"+str(i)
		for j in gramA[ai]:
			if ai==j[0]:
				gramA = removeDirectLR(gramA, ai)
				break

	op = {}
	for i in gramA:
		a = str(i)
		for j in conv:
			a = a.replace(conv[j],j)
		revconv[i] = a

	for i in gramA:
```

```
                l = []
                for j in gramA[i]:
                        k = []
                        for m in j:
                                if m in revconv:
                                        k.append(m.replace(m,revconv[m]))
                                else:
                                        k.append(m)
                        l.append(k)
                op[revconv[i]] = l
        return op

n = int(input("Enter No of Production: "))
for i in range(n):
   txt=input()
   add(txt)
result = rem(gram)
for x,y in result.items():
   print(f'{x} -> ', end="")
   for index, i in enumerate(y):
      for j in i:
         print(j, end="")
         if (index != len(y) - 1):
            print(" | ", end="")
   print()
```

Input:

```
Enter No of Production: 3
E -> E + T | T
T -> T * F | F
F -> ( E ) | id
```

Output:

```
E -> TE'
T -> FT'
F -> ( | E | ) | id
E' -> + | T | E' | e
T' -> * | F | T' | e
```

Result:

Thus, the python program to implement elimination of left recursion is executed successfully and tested with various samples.

EX.NO: 04 DATE:	Left Factoring

Aim:

To write a python program to implement Left Factoring for given sample data.

Algorithm:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing.When the choice between two alternative A-productions is not clear,we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

INPUT: Grammar G.

OUTPUT: An equivalent left-factored grammar.

METHOD:

1) For each nonterminal A, find the longest prefix α common to two or more of its alternatives.

2) If a=∈ ,there is a non trivial common prefix and hence replace all of A-productions, A→αβ1| αβ2 |…..| αβn| γ, where γ represents all alternatives that do not begin with α ,by

A →αA'| γ

A' →β1| β2|…… |βn

Here,A' is new non terminal. Repeatedly apply this transformation until two alternatives for a non terminal have common prefix

Program:

```
from itertools import takewhile


s= "S->iEtS|iEtSeS|a"


def groupby(ls):
   d = {}
   ls = [ y[0] for y in rules ]
```

```
    initial = list(set(ls))
    for y in initial:
        for i in rules:
            if i.startswith(y):
                if y not in d:
                    d[y] = []
                d[y].append(i)
    return d

def prefix(x):
    return len(set(x)) == 1
starting=""
rules=[]
common=[]
alphabetset=["A'","B'","C'","D'","E'","F'","G'","H'","I'","J'","K'","L'","M'","N'","O'","P'","Q'","R'","
S'","T'","U'","V'","W'","X'","Y'","Z'"]
s = s.replace(" ", "").replace("", "").replace("\n", "")
while(True):
            rules=[]
            common=[]
            split=s.split("->")
            starting=split[0]
            for i in split[1].split("|"):
                        rules.append(i)
            #logic for taking commons out
                for k, l in groupby(rules).items():
                    r = [l[0] for l in takewhile(prefix, zip(*l))]
                    common.append(".join(r))
            #end of taking commons
                for i in common:
                    newalphabet=alphabetset.pop()
```

```
            print(starting+"->"+i+newalphabet)
            index=[]
            for k in rules:
                if(k.startswith(i)):
                    index.append(k)
            print(newalphabet+"->",end="")
            for j in index[:-1]:
                stringtoprint=j.replace(i,"", 1)+"|"
                if stringtoprint=="|":
                    print("\u03B5","|",end="")
                else:
                    print(j.replace(i,"", 1)+"|",end="")
            stringtoprint=index[-1].replace(i,"", 1)+"|"
            if stringtoprint=="|":
                print("\u03B5","",end="")
            else:
                print(index[-1].replace(i,"", 1)+"",end="")
            print("")
        break
```

Output:

S->aZ'

Z'->ε

S->iEtSY'

Y'->ε |eS

Result:

Thus, the python program to implement elimination of left factoring is executed successfully and tested with various samples.

EX.NO: 05 DATE:	Computation of First and Follow Sets

Aim:

To write a python program to Compute First and Follow Sets for given sample data.

Algorithm:

- Create a dictionary to store the grammar rules.
- Create a dictionary to store the First sets for each nonterminal symbol.
- Create a dictionary to store the Follow sets for each nonterminal symbol.
- For each nonterminal symbol in the grammar, initialize its First set to an empty set in the First sets dictionary.
- For each terminal symbol in the grammar, add it to the First set of the production that generates it.
- Repeat the following until no more changes occur:
    - For each production in the grammar:
    i. If the production starts with a terminal symbol, add it to the First set of the nonterminal symbol that the production belongs to.
    ii. If the production starts with a nonterminal symbol, add the First set of that symbol (excluding epsilon) to the First set of the nonterminal symbol that the production belongs to. If epsilon is in the First set of the nonterminal symbol, add the First set of the next symbol in the production to the First set of the nonterminal symbol that the production belongs to, until a terminal symbol is found or the end of the production is reached.
- For each nonterminal symbol in the grammar, initialize its Follow set to an empty set in the Follow sets dictionary.
- Add the end-of-string marker ('$') to the Follow set of the start symbol.
- Repeat the following until no more changes occur:
    - For each nonterminal symbol in the grammar:
    i. For each production that contains the nonterminal symbol:
    1. If the nonterminal symbol is at the end of the production, add the Follow set of the nonterminal symbol that the production belongs to to the Follow set of the nonterminal symbol.
    2. Otherwise, add the First set of the next symbol in the production (excluding epsilon) to the Follow set of the nonterminal symbol. If epsilon is in the First set of the next symbol, repeat this step for the next symbol in the production, until a non-epsilon symbol is found or the end of the production is reached. If the end of the production is reached and epsilon is in the First set of the last symbol, add the Follow set of the nonterminal symbol that the production belongs to to the Follow set of the nonterminal symbol.

Return the First sets and Follow sets dictionaries.

Program:

```
gram = {
        "E":["E+T","T"],
        "T":["T*F","F"],
        "F":["(E)","i"]
}

def removeDirectLR(gramA, A):
        temp = gramA[A]
        tempCr = []
        tempInCr = []
        for i in temp:
                if i[0] == A:
                        #tempInCr.append(i[1:])
                        tempInCr.append(i[1:]+[A+"'"])
                else:
                        #tempCr.append(i)
                        tempCr.append(i+[A+"'"])
        tempInCr.append(["e"])
        gramA[A] = tempCr
        gramA[A+"'"] = tempInCr
        return gramA


def checkForIndirect(gramA, a, ai):
        if ai not in gramA:
                return False
        if a == ai:
                return True
        for i in gramA[ai]:
                if i[0] == ai:
                        return False
                if i[0] in gramA:
                        return checkForIndirect(gramA, a, i[0])
        return False

def rep(gramA, A):
        temp = gramA[A]
        newTemp = []
        for i in temp:
                if checkForIndirect(gramA, A, i[0]):
                        t = []
                        for k in gramA[i[0]]:
                                t=[]
                                t+=k
```

```
                        t+=i[1:]
                        newTemp.append(t)

            else:
                newTemp.append(i)
        gramA[A] = newTemp
        return gramA

def rem(gram):
        c = 1
        conv = {}
        gramA = {}
        revconv = {}
        for j in gram:
            conv[j] = "A"+str(c)
            gramA["A"+str(c)] = []
            c+=1

        for i in gram:
            for j in gram[i]:
                temp = []
                for k in j:
                    if k in conv:
                        temp.append(conv[k])
                    else:
                        temp.append(k)
                gramA[conv[i]].append(temp)


        #print(gramA)
        for i in range(c-1,0,-1):
            ai =  "A"+str(i)
            for j in range(0,i):
                aj = gramA[ai][0][0]
                if ai!=aj :
                    if aj in gramA and checkForIndirect(gramA,ai,aj):
                        gramA = rep(gramA, ai)

        for i in range(1,c):
            ai = "A"+str(i)
            for j in gramA[ai]:
                if ai==j[0]:
                    gramA = removeDirectLR(gramA, ai)
                    break

        op = {}
```

```
        for i in gramA:
                a = str(i)
                for j in conv:
                        a = a.replace(conv[j],j)
                revconv[i] = a

        for i in gramA:
                l = []
                for j in gramA[i]:
                        k = []
                        for m in j:
                                if m in revconv:
                                        k.append(m.replace(m,revconv[m]))
                                else:
                                        k.append(m)
                        l.append(k)
                op[revconv[i]] = l

        return op

result = rem(gram)



def first(gram, term):
        a = []
        if term not in gram:
                return [term]
        for i in gram[term]:
                if i[0] not in gram:
                        a.append(i[0])
                elif i[0] in gram:
                        a += first(gram, i[0])
        return a

firsts = {}
for i in result:
        firsts[i] = first(result,i)
        print(f'First({i}):',firsts[i])

def follow(gram, term):
        a = []
        for rule in gram:
                for i in gram[rule]:
                        if term in i:
                                temp = i
                                indx = i.index(term)
```

```
                                        if indx+1!=len(i):
                                                if i[-1] in firsts:
                                                        a+=firsts[i[-1]]
                                                else:
                                                        a+=[i[-1]]
                                        else:

                                                a+=["e"]
                                        if rule != term and "e" in a:
                                                a+= follow(gram,rule)
                return a


        follows = {}
        for i in result:
                follows[i] = list(set(follow(result,i)))
                if "e" in follows[i]:
                        follows[i].pop(follows[i].index("e"))
                follows[i]+=["$"]
                print(f'Follow({i}):',follows[i])
```

Output:

```
First(E): ['(', 'i']
First(T): ['(', 'i']
First(F): ['(', 'i']
First(E'): ['+', 'e']
First(T'): ['*', 'e']
Follow(E): [')', '$']
Follow(T): [')', '+', '$']
Follow(F): ['*', ')', '+', '$']
Follow(E'): [')', '$']
Follow(T'): [')', '+', '$']
```

Result:

Thus, the python program to implement the Computation of First and Follow sets is executed successfully and tested with various samples.

EX.NO: 06 DATE:	Computation of Leading and Trailing Sets

Aim:

To write a python program to compute Leading and Trailing Sets for given sample data.

Algorithm:

1. Create a dictionary to store the grammar rules.
2. Create a dictionary to store the Leading sets for each nonterminal symbol.
3. Create a dictionary to store the Trailing sets for each nonterminal symbol.
4. For each nonterminal symbol in the grammar, initialize its Leading set to an empty set in the Leading sets dictionary.
5. For each terminal symbol in the grammar, add it to the Leading set of the production that generates it.
6. Repeat the following until no more changes occur:
    i. For each production in the grammar:
    - If the production starts with a terminal symbol, add it to the Leading set of the nonterminal symbol that the production belongs to.
    - If the production starts with a nonterminal symbol, add the Leading set of that symbol (excluding epsilon) to the Leading set of the nonterminal symbol that the production belongs to. If epsilon is in the Leading set of the nonterminal symbol, add the Leading set of the next symbol in the production to the Leading set of the nonterminal symbol that the production belongs to, until a terminal symbol is found or the end of the production is reached.
7. For each nonterminal symbol in the grammar, initialize its Trailing set to an empty set in the Trailing sets dictionary.
8. Add the end-of-string marker ('$') to the Trailing set of the start symbol.
9. Repeat the following until no more changes occur:
    a. For each nonterminal symbol in the grammar:
    i. For each production that contains the nonterminal symbol:
    1. If the nonterminal symbol is at the end of the production, add the Trailing set of the nonterminal symbol that the production belongs to to the Trailing set of the nonterminal symbol.
    2. Otherwise, add the Trailing set of the next symbol in the production (excluding epsilon) to the Trailing set of the nonterminal symbol. If epsilon is in the Trailing set of the next symbol, repeat this step for the next symbol in the production, until a non-epsilon symbol is found or the beginning of the production is reached. If the beginning of the production is reached and epsilon is in the Trailing set of the first symbol, add the Trailing set of the nonterminal symbol that the production belongs to to the Trailing set of the nonterminal symbol.
10. Return the Leading sets and Trailing sets dictionaries.

Program:

```
a = ["E=E+T",
     "E=T",
     "T=T*F",
     "T=F",
     "F=(E)",
     "F=i"]

rules = {}
terms = []
for i in a:
    temp = i.split("=")
    terms.append(temp[0])
    try:
        rules[temp[0]] += [temp[1]]
    except:
        rules[temp[0]] = [temp[1]]

terms = list(set(terms))
print(rules,terms)

def leading(gram, rules, term, start):
    s = []
    if gram[0] not in terms:
        return gram[0]
    elif len(gram) == 1:
        return [0]
    elif gram[1] not in terms and gram[-1] is not start:
        for i in rules[gram[-1]]:
            s+= leading(i, rules, gram[-1], start)
            s+= [gram[1]]
```

```
        return s

def trailing(gram, rules, term, start):
    s = []
    if gram[-1] not in terms:
        return gram[-1]
    elif len(gram) == 1:
        return [0]
    elif gram[-2] not in terms and gram[-1] is not start:

        for i in rules[gram[-1]]:
            s+= trailing(i, rules, gram[-1], start)
            s+= [gram[-2]]
        return s

leads = {}
trails = {}
for i in terms:
    s = [0]
    for j in rules[i]:
        s+=leading(j,rules,i,i)
    s = set(s)
    s.remove(0)
    leads[i] = s
    s = [0]
    for j in rules[i]:
        s+=trailing(j,rules,i,i)
    s = set(s)
    s.remove(0)
    trails[i] = s
```

```
for i in terms:
    print("LEADING("+i+"):",leads[i])
for i in terms:
    print("TRAILING("+i+"):",trails[i])
```

Output:

```
{'E': ['E+T', 'T'], 'T': ['T*F', 'F'], 'F': ['(E)', 'i']} ['T', 'F', 'E']
LEADING(T): {'(', 'i', '*'}
LEADING(F): {'(', 'i'}
LEADING(E): {'(', 'i', '+', '*'}
TRAILING(T): {'*', 'i', ')'}
TRAILING(F): {'i', ')'}
TRAILING(E): {'i', '+', ')', '*'}
```

Result:

Thus, the python program to implement the Computation of Leading and Trailing sets is executed successfully and tested with various samples.

EX.NO: 07 DATE:	Construction of Predictive Parsing Table

Aim:

To write a python program to construct the Predictive Parsing Table for given sample data.

Algorithm:

The algorithm for constructing the predictive parsing table works as follows:

Step 1:

Iterate over all non-terminal symbols in the grammar, and for each symbol, iterate over all of its productions.

Step 2:

For each production, iterate over all the terminals that can be derived from the first symbol in the production using the **`first()`** function, and for each such terminal, add the production to the corresponding cell in the predictive parsing table.

Step 3:

If the production derives the empty string (epsilon), iterate over all the terminals that can follow the non-terminal symbol in any derivation using the `follow` function, and for each such terminal, add the production to the corresponding cell in the predictive parsing table.

Step 4:

If any cell in the predictive parsing table contains more than one production, the grammar is not LL(1) and cannot be parsed by a predictive parser.

Program:

```
# #example for direct left recursion
# gram = {"A":["Aa","Ab","c","d"]
# }
#example for indirect left recursion
gram = {
        "E":["E+T","T"],
        "T":["T*F","F"],
        "F":["(E)","i"]
}
```

```
def removeDirectLR(gramA, A):
	"""gramA is dictonary"""
	temp = gramA[A]
	tempCr = []
	tempInCr = []
	for i in temp:
	  if i[0] == A:
		#tempInCr.append(i[1:])
		tempInCr.append(i[1:]+[A+"'"])
	  else:
		#tempCr.append(i)
		tempCr.append(i+[A+"'"])
	tempInCr.append(["e"])
	gramA[A] = tempCr
	gramA[A+"'"] = tempInCr
	return gramA


def checkForIndirect(gramA, a, ai):
	if ai not in gramA:
	  return False
	if a == ai:
	  return True
	for i in gramA[ai]:
	  if i[0] == ai:
		return False
	  if i[0] in gramA:
		return checkForIndirect(gramA, a, i[0])
	return False

def rep(gramA, A):
	temp = gramA[A]
	newTemp = []
	for i in temp:
	  if checkForIndirect(gramA, A, i[0]):
		t = []
		for k in gramA[i[0]]:
			t=[]
			t+=k
			t+=i[1:]
			newTemp.append(t)

	  else:
		newTemp.append(i)
	gramA[A] = newTemp
```

```
    return gramA

def rem(gram):
    c = 1
    conv = {}
    gramA = {}
    revconv = {}
    for j in gram:
      conv[j] = "A"+str(c)
      gramA["A"+str(c)] = []
      c+=1

    for i in gram:
      for j in gram[i]:
            temp = []
            for k in j:
                  if k in conv:
                        temp.append(conv[k])
                  else:
                        temp.append(k)
            gramA[conv[i]].append(temp)


    #print(gramA)
    for i in range(c-1,0,-1):
      ai = "A"+str(i)
      for j in range(0,i):
            aj = gramA[ai][0][0]
            if ai!=aj :
                  if aj in gramA and checkForIndirect(gramA,ai,aj):
                        gramA = rep(gramA, ai)

    for i in range(1,c):
      ai = "A"+str(i)
      for j in gramA[ai]:
            if ai==j[0]:
                  gramA = removeDirectLR(gramA, ai)
                  break

    op = {}
    for i in gramA:
      a = str(i)
      for j in conv:
            a = a.replace(conv[j],j)
      revconv[i] = a
```

```
    for i in gramA:
      l = []
      for j in gramA[i]:
            k = []
            for m in j:
                  if m in revconv:
                        k.append(m.replace(m,revconv[m]))
                  else:
                        k.append(m)
            l.append(k)
      op[revconv[i]] = l

    return op


result = rem(gram)
terminals = []
for i in result:
    for j in result[i]:
      for k in j:
            if k not in result:
                  terminals+=[k]
terminals = list(set(terminals))
#print(terminals)

def first(gram, term):
    a = []
    if term not in gram:
      return [term]
    for i in gram[term]:
      if i[0] not in gram:
            a.append(i[0])
      elif i[0] in gram:
            a += first(gram, i[0])
    return a

firsts = {}
for i in result:
    firsts[i] = first(result,i)
#   print(f'First({i}):',firsts[i])

def follow(gram, term):
    a = []
    for rule in gram:
      for i in gram[rule]:
            if term in i:
                  temp = i
```

```
                        indx = i.index(term)
                        if indx+1!=len(i):
                                if i[-1] in firsts:
                                        a+=firsts[i[-1]]
                                else:
                                        a+=[i[-1]]
                        else:

                                a+=["e"]
                        if rule != term and "e" in a:
                                a+= follow(gram,rule)
        return a


follows = {}
for i in result:
        follows[i] = list(set(follow(result,i)))
        if "e" in follows[i]:
           follows[i].pop(follows[i].index("e"))
        follows[i]+=["$"]
#       print(f'Follow({i}):',follows[i])

resMod = {}
for i in result:
        l = []
        for j in result[i]:
           temp = ""
           for k in j:
                   temp+=k
           l.append(temp)
        resMod[i] = l

# create predictive parsing table
tterm = list(terminals)
tterm.pop(tterm.index("e"))
tterm+=["$"]
pptable = {}
for i in result:
        for j in tterm:
           if j in firsts[i]:
                   pptable[(i,j)]=resMod[i][0]
           else:
                   pptable[(i,j)]=""
        if "e" in firsts[i]:
           for j in tterm:
                   if j in follows[i]:
                           pptable[(i,j)]="e"
pptable[("F","i")] = "i"
```

```
toprint = f'{"": <10}'
for i in tterm:
    toprint+= f'|{i: <10}'
print(toprint)
for i in result:
    toprint = f'{i: <10}'
    for j in tterm:
      if pptable[(i,j)]!="":
            toprint+=f'|{i+"->"+pptable[(i,j)]: <10}'
      else:
            toprint+=f'|{pptable[(i,j)]: <10}'
    print(f'{"-":-<76}')
    print(toprint)
```

Output:

```
          |(         |i         |)         |+         |*         |$
---------------------------------------------------------------------------
E         |E->TE'    |E->TE'    |          |          |          |
---------------------------------------------------------------------------
T         |T->FT'    |T->FT'    |          |          |          |
---------------------------------------------------------------------------
F         |F->(E)    |F->i      |          |          |          |
---------------------------------------------------------------------------
E'        |          |          |E'->e     |E'->+TE'  |          |E'->e
---------------------------------------------------------------------------
T'        |          |          |T'->e     |T'->e     |T'->*FT'  |T'->e
```

Result:

Thus, the python program to implement the Construction of Predictive Parsing Table is executed successfully and tested with various samples.

EX.NO: 08 DATE:	Shift Reduce Parsing

Aim:

To write a python program to implement Shift Reduce Parsing for given sample data.

Algorithm:

Step 1: Define the grammar: Define the grammar rules in a dictionary format, where each non-terminal is a key, and its corresponding productions are values. Also, define the starting non-terminal and the input string.

Step 2: Initialize the stack: Initialize the stack with the starting non-terminal and a dollar sign ($).

Step 3: Iterate until the input string is fully processed or an error is encountered:

a. Check for any reduction rules: Iterate through the grammar rules for the current non-terminal symbol on top of the stack. If any of the productions of the current non-terminal is present in the stack, replace it with the non-terminal symbol and print a reduction message.

b. If no reduction rules apply, check for a shift: If the input string is not empty, add the next input symbol to the stack and move the input pointer to the next symbol. Print a shift message

c. If neither reduction nor shift is possible, reject the input string.

Step 4: If the stack contains only the starting non-terminal and a dollar sign and the input string is empty, accept the input string. Otherwise, reject the input string.

Program:

```
# example 1
gram = {
   "S":["S+S","S*S","id"]
}
starting_terminal = "S"
inp =  "id+id+id$"
stack = "$"
print(f'{"Stack": <15}'+"|"+f'{"Input Buffer": <15}'+"|"+f'Parsing Action')
```

```
print(f'{"-":-<50}')
while True:
  action = True
  i = 0
  while i<len(gram[starting_terminal]):
        if gram[starting_terminal][i] in stack:
              stack = stack.replace(gram[starting_terminal][i],starting_terminal)
              print(f'{stack:          <15}'+"|"+f'{inp:          <15}'+"|"+f'Reduce          S-
>{gram[starting_terminal][i]}')
              i=-1
              action = False
        i+=1
  if len(inp)>1:
        stack+=inp[0]
        inp=inp[1:]
        print(f'{stack: <15}'+"|"+f'{inp: <15}'+"|"+f'Shift')
        action = False
  if inp == "$" and stack == ("$"+starting_terminal):
        print(f'{stack: <15}'+"|"+f'{inp: <15}'+"|"+f'Accepted')
        break
  if action:
        print(f'{stack: <15}'+"|"+f'{inp: <15}'+"|"+f'Rejected')
        break
```

Output:

```
Stack          |Input Buffer    |Parsing Action
--------------------------------------------------
$i             |d+id+id$        |Shift
$id            |+id+id$         |Shift
$S             |+id+id$         |Reduce S->id
$S+            |id+id$          |Shift
$S+i           |d+id$           |Shift
$S+id          |+id$            |Shift
$S+S           |+id$            |Reduce S->id
$S             |+id$            |Reduce S->S+S
$S+            |id$             |Shift
$S+i           |d$              |Shift
$S+id          |$               |Shift
$S+S           |$               |Reduce S->id
$S             |$               |Reduce S->S+S
$S             |$               |Accepted
```

Result:

Thus, the python program to implement Shift Reduce Parsing is executed successfully and tested with various samples.

EX.NO: 09 DATE:	Computation of LR(0) items

Aim:

To write a python program to implement Computation of LR(0) items for given sample data.

Algorithm:

1. Start with the augmented grammar, which is the original grammar with an additional production S' → S, where S is the start symbol of the original grammar and S' is a new start symbol.
2. Create the first item of the initial state, which is the augmented start production with a dot at the beginning: S' → .S.
3. For each item A → α.Bβ, where B is a non-terminal symbol and α and β are strings of symbols (possibly empty), create the following items:
    - A → α.B.β
    - For each production B → γ, create the item B → .γ.
4. Repeat step 3 until no new items can be created.
5. Group the items by the non-terminal symbol that appears to the right of the dot.
6. For each group of items, create a new state in the LR(0) parsing table.
7. For each state, create a transition to a new state for each terminal or non-terminal symbol that appears to the right of the dot in any item in the state.
8. If a group of items contains an item of the form A → α., then add a reduce action for the state and symbol A in the parsing table.
9. If a group of items contains an item of the form S' → S., then add an accept action for the state and end-of-input symbol in the parsing table.
10. Repeat steps 6 to 9 until all states have been created and all actions have been added to the parsing table.

Program:

```
gram = {
        "S":["CC"],
        "C":["aC","d"]
}
start = "S"
terms = ["a","d","$"]

non_terms = []
for i in gram:
        non_terms.append(i)
gram["S'"]= [start]

new_row = {}
for i in terms+non_terms:
        new_row[i]=""

non_terms += ["S'"]
# each row in state table will be dictionary {nonterms ,term,$}
stateTable = []
# I = [(terminal, closure)]
# I = [("S","A.A")]

def Closure(term, I):
        if term in non_terms:
                for i in gram[term]:
                        I+=[(term,"."+i)]
        I = list(set(I))
        for i in I:
                # print("." != i[1][-1],i[1][i[1].index(".")+1])
                if "." != i[1][-1] and i[1][i[1].index(".")+1] in non_terms and i[1][i[1].index(".")+1]
```

```
!= term:
                        I += Closure(i[1][i[1].index(".")+1], [])
        return I


Is = []
Is+=set(Closure("S'", []))

countI =  0
omegaList = [set(Is)]
while countI<len(omegaList):
        newrow = dict(new_row)
        vars_in_I = []
        Is = omegaList[countI]
        countI+=1
        for i in Is:
                if i[1][-1]!=".":
                        indx = i[1].index(".")
                        vars_in_I+=[i[1][indx+1]]
        vars_in_I = list(set(vars_in_I))
        # print(vars_in_I)
        for i in vars_in_I:
                In = []
                for j in Is:
                        if "."+i in j[1]:
                                rep = j[1].replace("."+i,i+".")
                                In+=[(j[0],rep)]
                if (In[0][1][-1]!="."):
                        temp = set(Closure(i,In))
                        if temp not in omegaList:
                                omegaList.append(temp)
                        if i in non_terms:
```

```
                        newrow[i] = str(omegaList.index(temp))
                    else:
                        newrow[i] = "s"+str(omegaList.index(temp))
                    print(f'Goto(I{countI-1},{i}):{temp} That is I{omegaList.index(temp)}')
                else:
                    temp = set(In)
                    if temp not in omegaList:
                        omegaList.append(temp)
                    if i in non_terms:
                        newrow[i] = str(omegaList.index(temp))
                    else:
                        newrow[i] = "s"+str(omegaList.index(temp))
                    print(f'Goto(I{countI-1},{i}):{temp} That is I{omegaList.index(temp)}')
        stateTable.append(newrow)
print("\n\nList of I's\n")
for i in omegaList:
        print(f'I{omegaList.index(i)}: {i}')
#populate replace elements in state Table
I0 = []
for i in list(omegaList[0]):
        I0 += [i[1].replace(".","")]
print(I0)
for i in omegaList:
        for j in i:
            if "." in j[1][-1]:
                if j[1][-2]=="S":
                    stateTable[omegaList.index(i)]["$"] = "Accept"
                    break
                for k in terms:
                    stateTable[omegaList.index(i)][k] =
"r"+str(I0.index(j[1].replace(".","")))
```

```
print("\nStateTable")
print(f'{" ": <9}',end="")
for i in new_row:
    print(f'|{i: <11}',end="")
print(f'\n{"-":-<66}')
for i in stateTable:
    print(f'{"I("+str(stateTable.index(i))+")": <9}',end="")
    for j in i:
        print(f'|{i[j]: <10}',end=" ")
    print()
```

Output:

```
Goto(I0,d):{('C', 'd.')} That is I1
Goto(I0,S):{("S'", 'S.')} That is I2
Goto(I0,a):{('C', '.d'), ('C', '.aC'), ('C', 'a.C')} That is I3
Goto(I0,C):{('C', '.d'), ('S', 'C.C'), ('C', '.aC')} That is I4
Goto(I3,d):{('C', 'd.')} That is I1
Goto(I3,a):{('C', '.d'), ('C', '.aC'), ('C', 'a.C')} That is I3
Goto(I3,C):{('C', 'aC.')} That is I5
Goto(I4,d):{('C', 'd.')} That is I1
Goto(I4,a):{('C', '.d'), ('C', '.aC'), ('C', 'a.C')} That is I3
Goto(I4,C):{('S', 'CC.')} That is I6


List of I's

I0: {('C', '.d'), ('S', '.CC'), ('C', '.aC'), ("S'", '.S')}
I1: {('C', 'd.')}
I2: {("S'", 'S.')}
I3: {('C', '.d'), ('C', '.aC'), ('C', 'a.C')}
I4: {('C', '.d'), ('S', 'C.C'), ('C', '.aC')}
I5: {('C', 'aC.')}
I6: {('S', 'CC.')}
['d', 'CC', 'aC', 'S']

StateTable
          |a          |d          |$          |S          |C
------------------------------------------------------------------
I(0)      |s3         |s1         |           |2          |4
I(1)      |r0         |r0         |r0         |           |
I(2)      |           |           |Accept     |           |
I(3)      |s3         |s1         |           |           |5
I(4)      |s3         |s1         |           |           |6
I(5)      |r2         |r2         |r2         |           |
I(6)      |r1         |r1         |r1         |           |
```

Result:

Thus, the python program to compute LR(0) items is executed successfully and tested with various samples.

EX.NO: 10 DATE:	Intermediate Code Generation

Aim:

To write a python program to implement Intermediate Code Generation.

Algorithm:

Algorithm of Infix_To_Postfix:

1. Create an empty stack and an empty postfix string.
2. Loop through each token in the infix expression from left to right.
3. If the token is an operand (a number or a variable), add it to the postfix string.
4. If the token is a left parenthesis, push it onto the stack.
5. If the token is a right parenthesis, pop operators from the stack and add them to the postfix string until a left parenthesis is encountered. Discard the left parenthesis.
6. If the token is an operator (+, -, *, /, etc.), pop operators from the stack and add them to the postfix string while they have higher or equal precedence than the current operator (with the exception of left parenthesis). Then push the current operator onto the stack.
7. After processing all tokens, pop any remaining operators from the stack and add them to the postfix string.
8. The postfix string is the result of the conversion.

Algorithm of Infix_To_Prefix:

1. Reverse the infix expression
2. Replace opening parentheses with closing parentheses and vice versa
3. Create an empty stack
4. Initialize an empty prefix expression
5. For each symbol in the reversed infix expression, do the following:

a. If the symbol is an operand, add it to the prefix expression
b. If the symbol is a closing parenthesis, push it onto the stack
c. If the symbol is an operator, do the following:

i. If the stack is empty, push the operator onto the stack

ii. If the stack is not empty, compare the precedence of the operator with the top element of the stack:

1. If the precedence of the operator is higher, push the operator onto the stack

2. If the precedence of the operator is lower or equal, pop elements from the stack and add them to the prefix expression until an element with lower precedence is encountered. Then push the operator onto the stack

d. If the symbol is an opening parenthesis, do the following:

i. Pop elements from the stack and add them to the prefix expression until a closing parenthesis is encountered. Discard the closing parenthesis

6. Pop any remaining elements from the stack and add them to the prefix expression

7. Reverse the prefix expression to get the final result.

Algorithm of Three Address Code Generation:

1. Initialize an empty stack.

2. For each symbol in the input: a. If the symbol is an operand, push it onto the stack. b. If the symbol is an operator, pop the top two symbols from the stack and generate a new three-address code instruction with the operator and the two operands. Push the result onto the stack.

3. The final result is the top symbol on the stack.

Program:

```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])
PRI = {'+':1, '-':1, '*':2, '/':2}

### INFIX ===> POSTFIX ###
def infix_to_postfix(formula):
    stack = [] # only pop when the coming op has priority
    output = ''
    for ch in formula:
        if ch not in OPERATORS:
            output += ch
        elif ch == '(':
            stack.append('(')
        elif ch == ')':
            while stack and stack[-1] != '(':
                output += stack.pop()
            stack.pop() # pop '('
        else:
            while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:
                output += stack.pop()
            stack.append(ch)
    # leftover
    while stack:
```

```
            output += stack.pop()
    print(f'POSTFIX: {output}')
    return output

### INFIX ===> PREFIX ###
def infix_to_prefix(formula):
    op_stack = []
    exp_stack =  []
    for ch in formula:
        if not ch in OPERATORS:
            exp_stack.append(ch)
        elif ch == '(':
            op_stack.append(ch)
        elif ch == ')':
            while op_stack[-1] != '(':
                op = op_stack.pop()
                a = exp_stack.pop()
                b = exp_stack.pop()
                exp_stack.append( op+b+a )
            op_stack.pop() # pop '('
        else:
            while op_stack and op_stack[-1] != '(' and PRI[ch] <= PRI[op_stack[-1]]:
                op = op_stack.pop()
                a = exp_stack.pop()
                b = exp_stack.pop()
                exp_stack.append( op+b+a )
            op_stack.append(ch)

    #  leftover
    while op_stack:
        op = op_stack.pop()
        a = exp_stack.pop()
        b = exp_stack.pop()
        exp_stack.append( op+b+a )
    print(f'PREFIX: {exp_stack[-1]}')
    return exp_stack[-1]

### THREE ADDRESS CODE GENERATION ###
def generate3AC(pos):
            print("### THREE ADDRESS CODE GENERATION ###")
            exp_stack = []
            t = 1

            for i in pos:
                        if i not in OPERATORS:
                                    exp_stack.append(i)
```

```
            else:
                print(f't{t} := {exp_stack[-2]} {i} {exp_stack[-1]}')
                exp_stack=exp_stack[:-2]
                exp_stack.append(f't{t}')
                t+=1

expres = input("INPUT THE EXPRESSION: ")
pre = infix_to_prefix(expres)
pos = infix_to_postfix(expres)
generate3AC(pos)
```

Outputs:

INPUT THE EXPRESSION: (a+(b/c)*d)

PREFIX: +a*/bcd

POSTFIX: abc/d*+

THREE ADDRESS CODE GENERATION

t1 := b / c

t2 := t1 * d

t3 := a + t2

Result:

Thus, the python program to implement Intermediate Code Generation is executed successfully and tested with various samples.