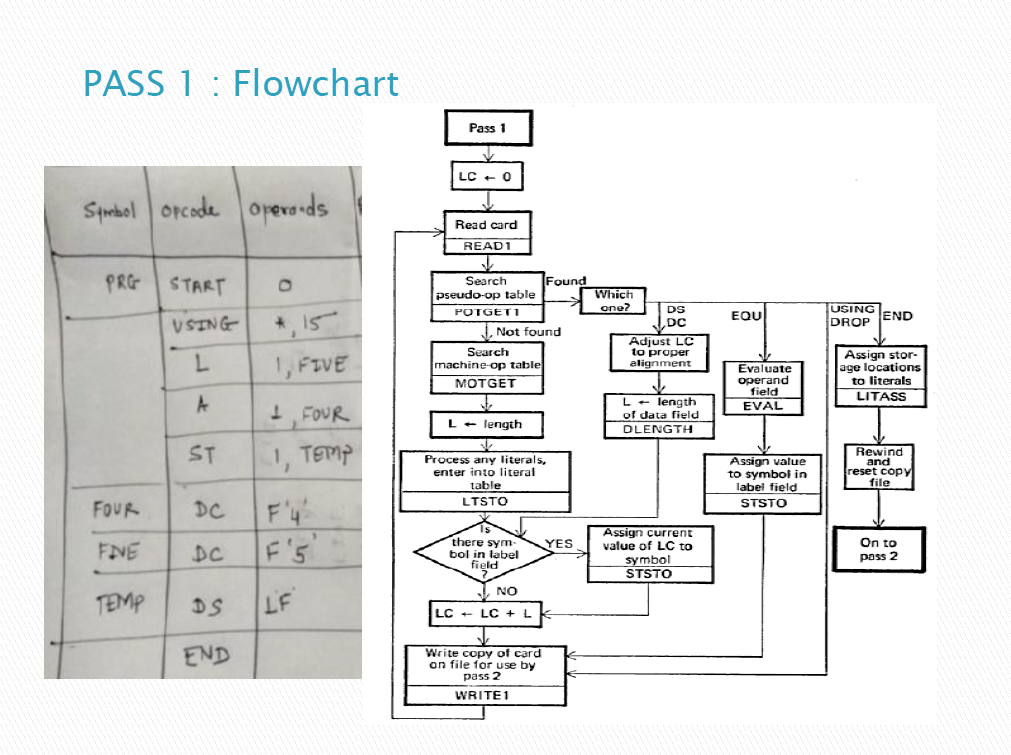
**SPCC ALGOS**

***1. Display LT, ST,BT***

***2. Assign LC.***

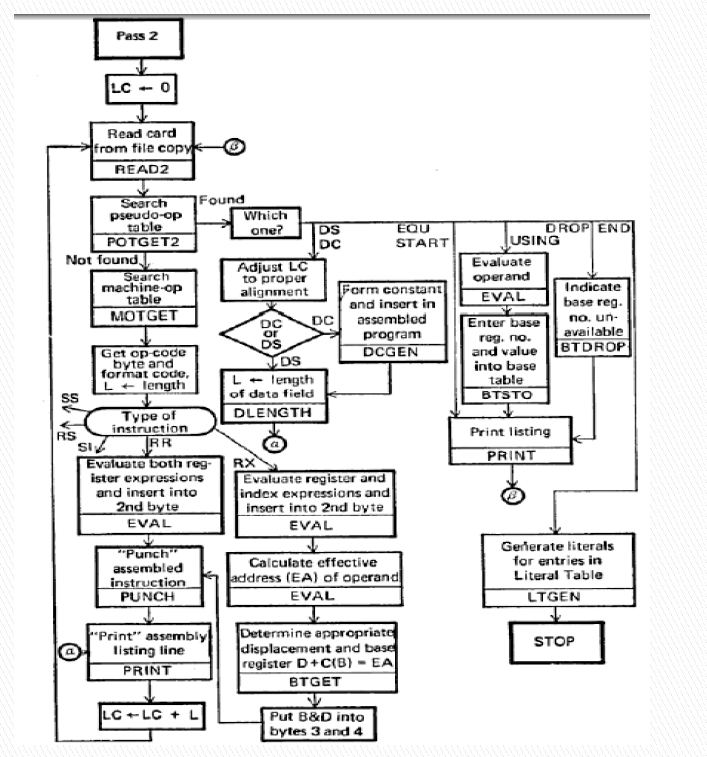
**2 Pass Assembler:**

**Pass 1:**

****

**2 Pass Assembler:**

**Pass 2:**

****

CODE:

***asm = []***

***f=["JOHN START 0","USING \*,15","L 1,FIVE","A 1,FOUR","ST 1,TEMP","FOUR DC F'4'","FIVE DC F'5'","TEMP DS 1F","END"]***

***for i in f:***

***asm.append(i.split())***

***oprands = ["START", "USING", "DC", "DS", "END", "A", "L", "ST"]***

***Poprands = ["START", "USING", "L", "END"]***

***mot = []***

***pot = []***

***st = []***

***lt = []***

***bt = []***

***lc = 0***

***def newLC(op, opc):***

***if op in Poprands:***

***return 0***

***else:***

***return 2 if opc[0] == "H" else 4***

***def updateTables(label, oprand, opcode, lc):***

***if oprand not in Poprands or oprand == "L":***

***mot.append((oprand, "", "01", "RX"))***

***else:***

***pot.append((oprand, ""))***

***if "'" in opcode:***

***lt.append((opcode[1:], str(lc), str(newLC(oprand, opcode))))***

***if label != " ":***

***st.append((label+"b"\*(8-len(label)), str(lc), str(len(label))))***

***if oprand == "USING":***

***\_,num = opcode.split(",")***

***for i in range(16):***

***if int(num) == i:***

***bt.append((str(i), "Y", "00"))***

***else:***

***bt.append((str(i), "N", "00"))***

***for i in asm:***

***if i[0] in oprands:***

***oprand = i[0]***

***label = " "***

***else:***

***label = i[0]***

***oprand = i[1]***

***opcode = i[-1]***

***if oprand == "END" or oprand == "START":***

***opcode = " "***

***lc += newLC(oprand, opcode)***

***updateTables(label, oprand, opcode, lc)***

***print(label, oprand, opcode, lc)***

***print()***

***print("\_" \* 20)***

***print("Pass1:")***

***print("="\*10)***

***print("MOT:")***

***for i in mot:***

***if i[0] in "DS DC":***

***continue***

***print(" ".join(i))***

***print("\_"\*10)***

***print("POT:")***

***for i in pot:***

***print(" ".join(i))***

***print("\_"\*10)***

***print("LT:")***

***for i in lt:***

***print(" ".join(i))***

***print("\_"\*10)***

***print("ST:")***

***for i in st:***

***print(" ".join(i))***

***print("\_" \* 20)***

***print("Pass2:")***

***print("="\*10)***

***print("MOT:")***

***for i in mot:***

***print(" ".join(i))***

***print("\_"\*10)***

***print("POT:")***

***for i in pot:***

***print(" ".join(i))***

***print("\_"\*10)***

***print("LT:")***

***for i in lt:***

***print(" ".join(i))***

***print("\_"\*10)***

***print("ST:")***

***for i in st:***

***print(" ".join(i))***

***print("\_"\*10)***

***print("BT:")***

***for i in bt:***

***print(" ".join(i))***

***3.Display MDT, MNT ,ALA***

***4. Display expanded code***

**2 Pass MACRO**

**CODE:**

def loader(ins):

mdt=[]

mnt=[]

ala=[]

esource=[]

i=0

temp=0

mindex=[]

mi=0

while(i<len(ins)):

if ins[i]=="MACRO":

dummy=[]

i=i+1

mnt.append(ins[i][:ins[i].index(" ")])

mindex.append(mi)

mi=mi+1

kj=[]

kj=ins[i][ins[i].index(" ")+1:].split(",")

ala.append(kj)

i=i+1

while(ins[i]!="MEND"):

kj1=ins[i].split(" ")

for ff in kj1[1].split(","):

# print(ff)

if(ff in kj):

#print(ff)

ins[i]=ins[i].replace(ff,str(kj.index(ff)+1))

# print(ins[i])

dummy.append(ins[i])

i=i+1

dummy.append("MEND")

mdt.append(dummy)

i=i+1

temp=i

else:

i=i+1

print("-- PASS 1 RESULTS ---")

print("-- MDT --")

for a in mdt:

print(a)

print("-- MNT --")

for b in zip(mnt,mindex):

print(b)

print("--- ALA ---")

print(ala)

print("-- PASS 2--")

j=0

''' for com in ins[:temp]:

esource.append(com)'''

for com in ins[temp:]:

com1=com.split(" ")

if com1[0] in mnt:

cc=com1[1].split(",")

j=mnt.index(com1[0])

for f in mdt[j]:

f1=f.split(" ")

if(len(f1)>1):

f2=f1[1].split(",")

for ii in f2:

f=f.replace(ii,str(cc[int(ii)-1]))

esource.append(f)

else:

esource.append(com)

for res in esource:

print(res)

n=int(input("Enter no of ins:"))

ins=[]

i=0

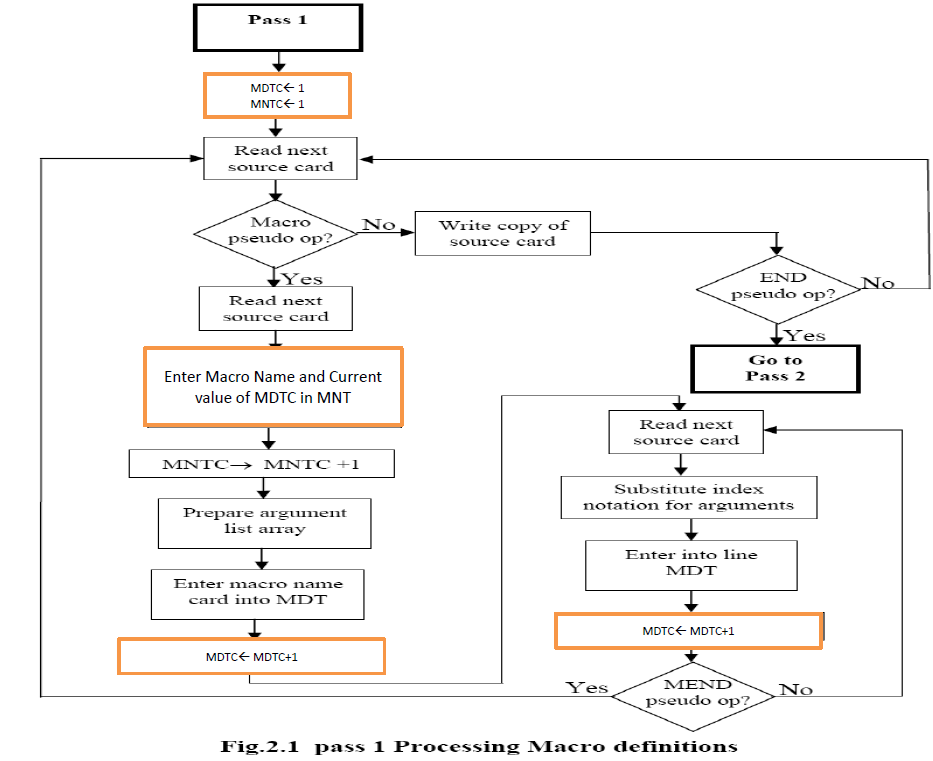
while(i<n):

ins.append(input("Enter ins:"))

i=i+1

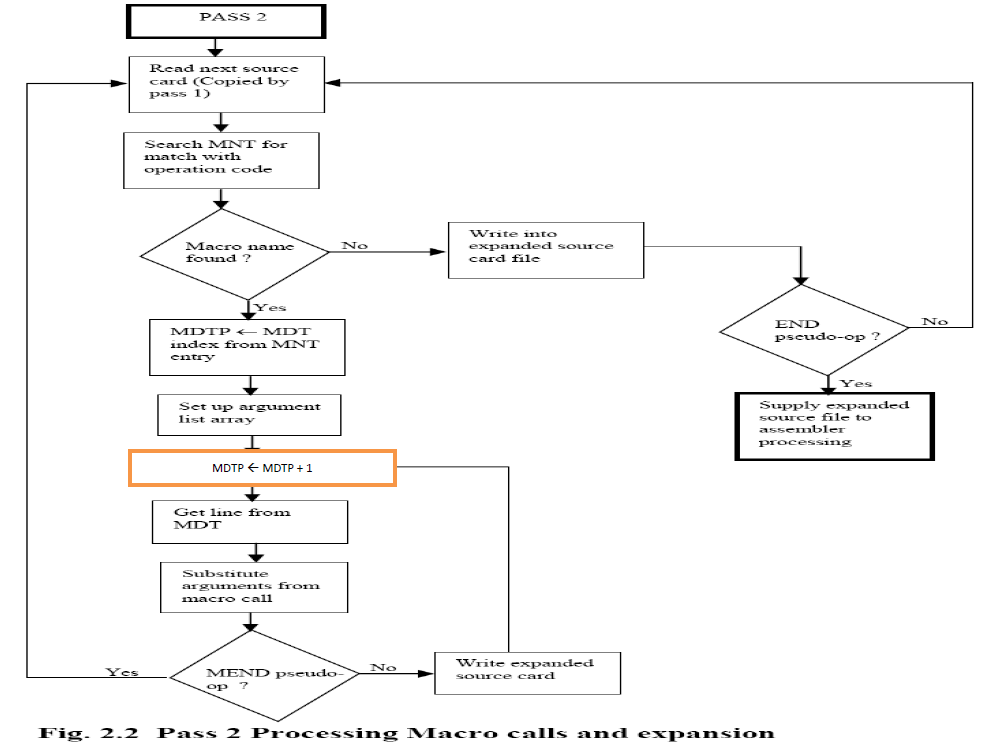
loader(ins)

**PASS1**

****

**2 Pass MACRO**

**PASS 2**

****

***5. Compute first set***

**ALGO**

Compute FIRST(X) as follows:

a) if X is a terminal, then FIRST(X)={X}

b) if X→ε is a production, then add ε to FIRST(X)

c) if X is a non-terminal and X→Y1Y2...Yn is a production, add FIRST(Yi) to FIRST(X) if the preceding Yjs contain ε in their FIRSTs

***6. Compute follow set.***

**ALGO**

**Compute FOLLOW as follows:**

#### Algorithm for calculating Follow set:

* if α is a start symbol, then FOLLOW() = $
* if α is a non-terminal and has a production α → AB, then FIRST(B) is in FOLLOW(A) except ℇ.
* if α is a non-terminal and has a production α → AB, where B ℇ, then FOLLOW(A) is in FOLLOW(α).

**CODE:**

import re

firsts = {}

follows = {}

prods = {}

n = int(input("Enter number of production rules: "))

class Prod:

ter = None

def \_\_init\_\_(self, p) -> None:

\_, p = p.split("->")

p = p.split("|")

self.rules = p

self.ter = \_

def \_\_str\_\_(self) -> str:

return f"{self.ter} {self.rules}"

def CheckFirst(prod: Prod):

first = set()

for rule in prod.rules:

if not rule[0].isupper():

first.add(rule[0])

elif '@' == rule:

first.add('@')

else:

for idx, Y in enumerate(rule):

if Y == "'":

continue

if idx+1 < len(rule) and rule[idx+1] == "'":

Y+= "'"

if Y.isupper():

firstY = CheckFirst(prods[Y])

for i in firstY:

first.add(i)

if not "@" in firstY:

break

if idx != len(rule) - 1:

first.remove("@")

return first

def checkFollow(prod: Prod):

if prod.ter == 'S':

return set(["$"])

follow = set()

regex = r"[A-B]"

for rule in prod.rules:

matches = re.finditer(regex, rule, re.MULTILINE)

for i in matches:

ter = i[0]

# A -> aBb

if i.span()[0] != 0 and i.span()[1] != len(rule):

temp = firsts[ter]

for f in temp:

if "@" == f:

continue

follow.add(f)

# A -> aB || A -> aBb

elif i.span()[0] != 0:

temp = checkFollow(prods[ter])

for f in temp:

follow.add(f)

return follow

for i in range(n):

prod = input("Enter prod rule: ")

prods[prod.split('->')[0]]=(Prod(prod))

for ter, prod in prods.items():

firsts[ter] = CheckFirst(prod)

follows[ter] = checkFollow(prod)

print(f"\_"\*15)

for ter, rule in firsts.items():

print(f"{ter} :")

print(rule)

print(f"\_"\*15)

for ter, rule in follows.items():

print(f"{ter} :")

print(rule)

***7. Optimization techniques- Dead code elimination, Algebraic simplification, common subexpression elimination***

DEADCODE THEORY:

How does this Lexical analyzer work?

The lexical analyzer created by Lex behaves in concert with a parser in the

following manner. When activated by the parser, the lexical analyzer begins reading

its remaining input , one character at a time, until it has found the longest prefix of the

input that is matched by one of the regular expressions p. Then it executes the

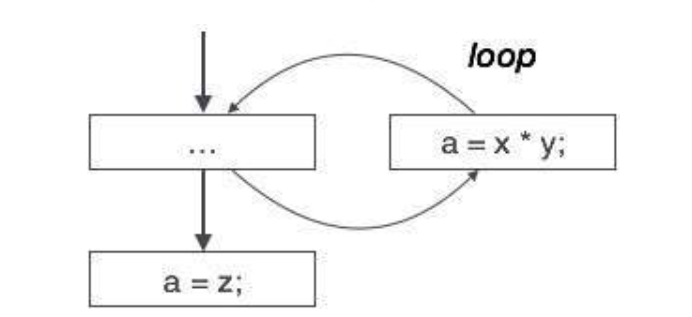
corresponding action. Typically the action will return control to the parser. However,

if it does not, then the lexical analyzer proceeds to find more lexemes, until an action

causes control to return to the parser. The repeated search for lexemes until an

explicit return allows the lexical analyzer to process white space and comments

conveniently.



CODE:

def cpo(code):

print("YOUR CODE WAS:")

print("------------------------")

for m in code:

print(m)

d={}

for i in code:

i=i.split("=")

if(i[1].isnumeric() or i[1].replace('.', '', 1).isdigit()):

d[i[0]]=i[1]

ans=[]

for i in code:

i=i.split("=")

for j in i[1].split(" "):

if(j in d.keys()):

i[1]=i[1].replace(j,d[j])

s=""

s=str(i[0])+"="+str(eval(i[1]))

ans.append(s)

print("OPTIMIZED CODE:")

print("------------------")

for c in ans:

print(c)

#------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

def cf(code):

print("YOUR CODE WAS:")

print("------------------------")

print(code)

o=["=","\*","+","-","/","\*\*"]

a=[]

ans=code.split(" ")

for i in ans:

if not i.isalpha() and i not in o:

a.append(eval(i))

else:

a.append(i)

a=list(map(str,a))

print("OPTIMIZED CODE:")

print("------------------")

print("".join(a))

#---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

def spcc(code):

print("YOUR CODE WAS:")

print("------------------------")

for m in code:

print(m)

s=[]

e=[]

k=[]

j={}

ans=[]

for i in code:

i=i.split("=")

if(i[1] not in e):

e.append(i[1])

s.append(i[0])

else:

k.append(i[0])

rep=e.index(i[1])

j[i[0]]=s[rep]

for i in e:

for u in k:

if u in i.split(" "):

i=i.replace(u,j[u])

ans.append(i)

z=zip(s,ans)

print("OPTIMIZED CODE:")

print("------------------")

for x,y in z:

print(str(x)+"="+str(y))

#-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

n=int(input("Number of lines:"))

code=[]

print("-- MENU----")

print("1.CONSTANT FOLDING")

print("2.CONSTANT PROPAGATION")

print("3.COMMON SUBEXPRSSION ELIMINATION")

print("----------------------------------------------")

ch=int(input("Enter your choice:"))

if(ch==1):

code=input("ENTER A CODE LINE:")

cf(code)

if(ch==2):

for t in range(n):

code.append(input("Enter line:"))

cpo(code)

if(ch==3):

for t in range(n):

code.append(input("Enter line:"))

spcc(code)

***8. Handwritten lexical analyzer***

**Algorithm:for lexical analysis**

1) Specify the grammar with the help of regular expression

2) Create transition table for combined NFA

3) read input character

4) Search the NFA for the input sequence.

5) On finding accepting state

i. if token is id or num search the symbol table

1. if symbol found return symbol id

2. else enter the symbol in symbol table and return its id.

ii. Else return token

6) Repeat steps 3 to 5 for all input characters.

CODE:

def lex(code):

keyword=["int","char","float","string","cout","cin"]

op=["+","-","\*","/","(",")","[","]","{","}",">>","<<",",",";","="]

num=["0","1","2","3","4","5","6","7","8","9"]

print("YOUR CODE IS:")

for command in code:

print(command)

print("----------------------------------------------------------------")

for command in code:

command=command.split(" ")

for j in command:

if j in keyword:

print("Keyword:",j)

elif j in op:

print("Operator:",j)

elif j.isdigit():

print("Number:",j)

else:

print("Identifier:",j)

co=int(input("ENTER NUMBER OF LINES:"))

code=[]

for u in range(co):

c=input("ENTER LINE:")

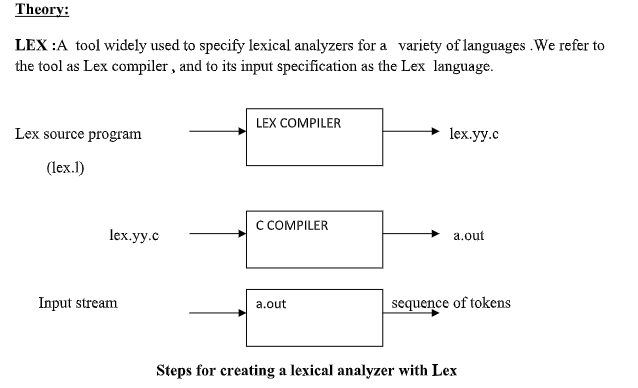
code.append(c)

print("--------------------------------------------------------------------")

lex(code)

***9. Lex tool***

THEORY



How does this Lexical analyzer work?

The lexical analyzer created by Lex behaves in concert with a parser in the

following manner. When activated by the parser, the lexical analyzer begins reading

its remaining input , one character at a time, until it has found the longest prefix of the

input that is matched by one of the regular expressions p. Then it executes the

corresponding action. Typically the action will return control to the parser. However,

if it does not, then the lexical analyzer proeeds to find more lexemes, until an action

causes control to return to the parser. The repeated search for lexemes until an

explicit return allows the lexical analyzer to process white space and comments

conveniently.

CODE:

%{

#define NUM 1

%}

%%

[0-9]+\.[0-9]+|[0-9]+ {return NUM;}

int|char|bool|void {return 2;}

[a-zA-Z]+[a-zA-Z0-9]\* {return 3;}

%%

#include<stdio.h>

int main(int argc, char \*argv[])

{ int val;

while(val=yylex()){

switch(val){

case 1:

printf("\n%s-Number", yytext);

break;

case 2:

printf("\n%s-Keyword", yytext);

break;

case 3:

printf("\n%s-Identifier", yytext);

break;

}

} return 0;

}

***10. ICG- 3AC and Quadruple representation***

CODE:

from tabulate import tabulate

def threeaddr(s):

l=s.split(" ")

l=l[2:]

op=['+','-','\*','/','^']

arg1=[]

arg2=[]

res=[]

oper=[]

n=(len(l))//2

if(l[n] not in op):

while(l[n] not in op):

n=n-1

p1=l[:n]

p2=l[n+1:]

ind=1

'''

oper.append('OPERATOR:')

arg1.append('ARGUMENT 1:')

arg2.append('ARGUMENT2:')

res.append('RESULT:')

'''

if(len(l)==3):

oper.append(l[n])

arg1.append(l[0])

arg2.append(l[2])

res.append("t"+str(ind))

oper.append("=")

arg1.append(s[0])

arg2.append("t"+str(ind))

res.append("t"+str(ind+1))

ans=[]

z1=zip(oper,arg1,arg2,res)

for a1,a2,a3,a4 in list(z1):

aq=[]

aq.append(a1)

aq.append(a2)

aq.append(a3)

aq.append(a4)

ans.append(aq)

print("QUADRAPLE TABLE:")

print(tabulate(ans, headers=["OPERATORS","ARG 1","ARG 2","RESULT"],tablefmt='orgtbl'))

else:

m=0

for i in p1:

if(i[0] in op and len(i)>1):

oper.append("unary"+i[0])

arg1.append(i[1])

arg2.append("nill")

res.append("t"+str(ind))

ind=ind+1

if(i in op and len(i)==1):

oper.append(i)

arg1.append(p1[m-1])

#print(p1.index(i)+1)

arg2.append(p1[m+1])

res.append("t"+str(ind))

my="t"+str(ind)

ind=ind+1

m=m+1

j=0

for i in p2:

if(i[0] in op and len(i)>1):

oper.append("unary"+i[0])

arg1.append(i[1])

arg2.append("nill")

res.append("t"+str(ind))

ind=ind+1

if(i in op and len(i)==1):

oper.append(i)

arg1.append(p2[j-1])

arg2.append(p2[j+1])

res.append("t"+str(ind))

you="t"+str(ind)

ind=ind+1

j=j+1

oper.append(l[n])

arg1.append(my)

arg2.append(you)

res.append("t"+str(ind))

oper.append("=")

arg1.append(s[0])

arg2.append("t"+str(ind))

res.append("t"+str(ind+1))

z=zip(oper,arg1,arg2,res)

ans=[]

for a1,a2,a3,a4 in list(z):

aq=[]

aq.append(a1)

aq.append(a2)

aq.append(a3)

aq.append(a4)

ans.append(aq)

print("QUADRAPLE TABLE:")

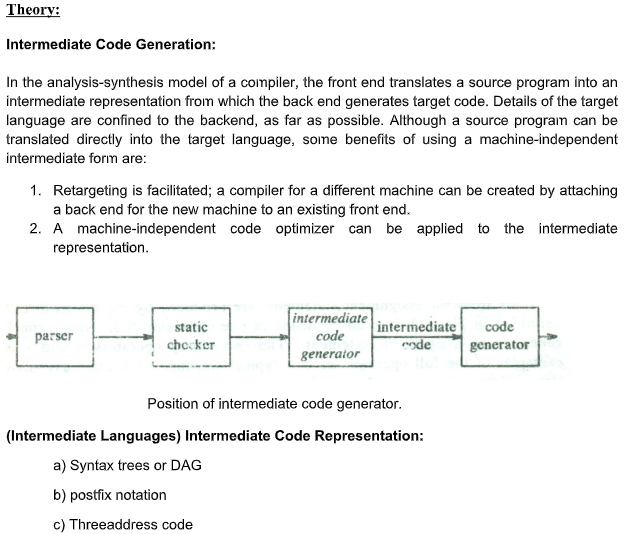
print(tabulate(ans, headers=["OPERATORS","ARG 1","ARG 2","RESULT"],tablefmt='orgtbl'))

# print(a1,a2,a3,a4)

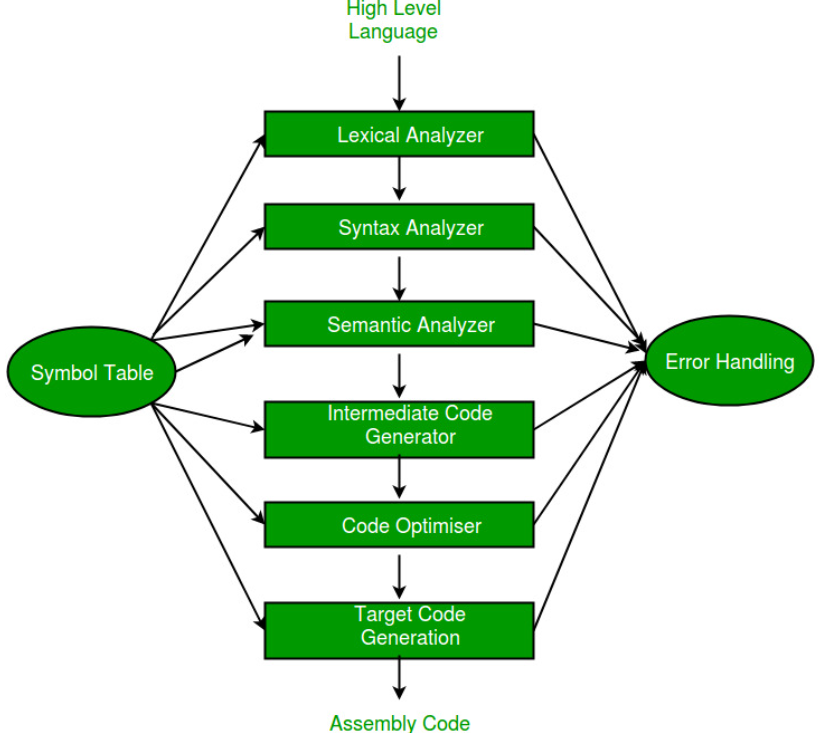
s=input("Enter code:")

threeaddr(s)

THEORY



***11. Code generation***

******

Algo:

Code-Generation Algorithm:

Code-Generation algorithm takes as a input a sequence of three address statements constituting a basic block. For each three address statement of the form X = Y op Z we perform the following actions:

1 Invoke a function getreg to determine the location L where the result of the

computation Y op Z should be stored. L will be a register or memory location.

2 Consult the address descriptor for y to determine y’, the current location of y.

Prefer the register for y’ if the value of y is currently both in memory and a

register. If the value of y is not already in L, generate the instruction MOV y’,

L to place a copy of y in L.

3 Generate the instruction OP z’, L where z’ is a current location of z. Prefer

register entry of z. update the address descriptor of x t indicate that x is in

location L. If L is a register, update its descriptors to indicate that it contains

the value of x, and remove x from all other register descriptors.

4 If the current values of y and/or z have no next uses, are not live on exit from

the block, and are in registers, alter the register descriptor to indicate that,

after execution of x = y op z, those registers no longer will contain y and/or z,

respectively.

**getReg** : Code generator uses *getReg* function to determine the status of available registers and the location of name values. *getReg* works as follows:

* If variable Y is already in register R, it uses that register.
* Else if some register R is available, it uses that register.
* Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions.

CODE:

def cg(code):

c=code[0]

d={}

c1=c.split("=")

op={'+':'ADD','-':'SUB'}

j=1

f=""

for i in c1[1]:

if i in op.keys():

f=i

continue

print("MOV"+ " " + "R"+str(j) + "," + i)

j=j+1

print(op[f]+" "+"R1"+","+"R2")

d[c1[0]]="R1"

for w in code[1:]:

ch=0

mm=0

r=[]

r=w.split("=")

if len(r[1])==1:

print("MOV"+" "+r[1]+","+d[r[1]])

else:

for t in r[1]:

if(r[1][0] in d.keys() and r[1][2] in d.keys()):

f=r[1][1]

m="R1"

ch=1

mm=1

break

if(t in op.keys()):

f=t

continue

if(t in d.keys()):

ch=1

if(r[1].index(t)==0):

ind=2

else:

ind=0

print("MOV"+" "+" "+"R2"+","+r[1][ind])

m=d[t]

if(ch==0 and mm!=1):

print("MOV"+" "+"R1"+","+r[1][0])

print("MOV"+" "+"R2"+","+r[1][2])

m="R1"

print(op[f]+" "+m+","+"R2")

d[r[0]]="R1"

code=[]

n=int(input("ENTER NUMBER OF LINES:"))

for y in range(n):

cc=input('ENTER LINE:')

code.append(cc)

cg(code)