Name: Suraj Jorwekar (41)

Pratima Nagare (67)

Title : Code Generation

CODE:

#!/usr/bin/python

import sys

class Stack:

def \_\_init\_\_(self):

self.items = []

return

def isEmpty(self):

return self.items == []

def push(self, item):

self.items.append(item)

def pop(self):

return self.items.pop()

def peek(self):

return self.items[len(self.items)-1]

def size(self):

return len(self.items)

def printStack(self):

for i in range(len(self.items)):

sys.stdout.write(str(self.items[i])+" ")

return ""

class Lex:

delim = [" ",",",";","{","}","(",")","$"]

operator = ["+","-","/","\*","=","&&","or","&","|","!","=="]

Relational = ["<",">","<=",">="]

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"]

lexeme\_list = []

identifier = []

invalid\_identifier = []

invalid\_operator = []

output = []

comment = 0

temp=-1

count=0

quad=1

CURRENT=""

MAIN\_TABLE = {}

LINENO = 0

flag=0

STATES = 35

COLUMN = 21

ids={}

QUAddrDescRUPLES = []

ACTUAL\_VALUE = ""

ACTION = "None"

SYMBOLS = []

nextUse = [["a","y","2"],["b","y","1"],["c","y","2"],["d","y","4"],["t","y","3"],["u","y","5"],["v","y","5"]]

AddrDesc = [["a","-1"],["b","-1"],["c","-1"],["d","-1"],["t","-1"],["u","-1"],["v","-1"]]

RegDesc = ["-1","-1","-1"]

NO\_SYM = [2,2,1,1,4,7,3,3,3,1,3,3,1,3,1]

#0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

TOKENS = ['id',';','=','{','}','(',')','relop','while','+','-','\*','/','$','S','A','I','C','E','T','F']

RULES = ["S->AS","S->IS","S->A","S->I","A->id=E;","I->while(C){S}","C->id relop id","E->E+T","E->E-T","E->T","T->T\*F","T->T/F","T->F","F->(E)","F->id"]

'''

TOKENS = ['id','+','\*','(',')','$','E','T','F']

RULES = ['E->E+T','E->T','T->T\*F','T->F','F->(E)','F->id']

NO\_SYM = [3,1,3,1,3,1]

'''

'''

actionTable = [

['s5','-','-','s4','-','-',1,2,3],

['-','s6','-','-','-','acc',-1,-1,-1],

['-','r2','s7','-','r2','r2',-1,-1,-1],

['-','r4','r4','-','r4','r4',-1,-1,-1],

['s5','-','-','s4','-','-',8,2,3],

['-','r6','r6','-','r6','r6',-1,-1,-1],

['s5','-','-','s4','-','-',-1,9,3],

['s5','-','-','s4','-','-',-1,-1,10],

['-','s6','-','-','s11','-',-1,-1,-1],

['-','r1','s7','-','r1','r1',-1,-1,-1],

['-','r3','r3','-','r3','r3',-1,-1,-1],

['-','r5','r5','-','r5','r5',-1,-1,-1]

]

'''

actionTable = [

['s4','0','0','0','0','s9','0','0','s5','0','0','0','0','0','1','2','3','0','6','7','8'],

['0','0','0','0','0','0','0','0','0','0','0','0','0','acc','0','0','0','0','0','0','0'],

['s11','0','0','0','r3','0','0','0','s5','0','0','0','0','r3','10','2','3','0','0','0','0'],

['s11','0','0','0','r4','0','0','0','s5','0','0','0','0','r4','12','2','3','0','0','0','0'],

['0','r15','s13','0','0','0','r15','s14','0','r15','r15','r15','r15','0','0','0','0','0','0','0','0'],

['0','0','0','0','0','s15','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'],

['0','0','0','0','0','0','0','0','0','s16','s17','0','0','0','0','0','0','0','0','0','0'],

['0','r10','0','0','0','0','r10','0','0','r10','r10','s18','s19','0','0','0','0','0','0','0','0'],

['0','r13','0','0','0','0','r13','0','0','r13','r13','r13','r13','0','0','0','0','0','0','0','0'],

['s21','0','0','0','0','s9','0','0','0','0','0','0','0','0','0','0','0','0','20','7','8'],

['0','0','0','0','r1','0','0','0','0','0','0','0','0','r1','0','0','0','0','0','0','0'],

['0','0','s13','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'],

['0','0','0','0','r2','0','0','0','0','0','0','0','0','r2','0','0','0','0','0','0','0'],

['s21','0','0','0','0','s9','0','0','0','0','0','0','0','0','0','0','0','0','22','7','8'],

['s23','0','0','0','0','0','r7','0','0','0','0','0','0','0','0','0','0','0','0','0','0'],

['s25','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','24','0','0','0'],

['s21','0','0','0','0','s9','0','0','0','0','0','0','0','0','0','0','0','0','0','26','8'],

['s21','0','0','0','0','s9','0','0','0','0','0','0','0','0','0','0','0','0','0','27','8'],

['s21','0','0','0','0','s9','0','0','0','0','0','0','0','0','0','0','0','0','0','0','28'],

['s21','0','0','0','0','s9','0','0','0','0','0','0','0','0','0','0','0','0','0','0','29'],

['0','0','0','0','0','0','s30','0','0','s16','s17','0','0','0','0','0','0','0','0','0','0'],

['0','r15','0','0','0','0','r15','0','0','r15','r15','r15','r15','0','0','0','0','0','0','0','0'],

['0','s31','0','0','0','0','0','0','0','s16','s17','0','0','0','0','0','0','0','0','0','0'],

['0','0','0','0','0','0','r7','0','0','0','0','0','0','0','0','0','0','0','0','0','0'],

['0','0','0','0','0','0','s32','0','0','0','0','0','0','0','0','0','0','0','0','0','0'],

['0','0','0','0','0','0','0','s14','0','0','0','0','0','0','0','0','0','0','0','0','0'],

['0','r8','0','0','0','0','r8','0','0','r8','r8','s18','s19','0','0','0','0','0','0','0','0'],

['0','r9','0','0','0','0','r9','0','0','r9','r9','s18','s19','0','0','0','0','0','0','0','0'],

['0','r11','0','0','0','0','r11','0','0','r11','r11','r11','r11','0','0','0','0','0','0','0','0'],

['0','r12','0','0','0','0','r12','0','0','r12','r12','r12','r12','0','0','0','0','0','0','0','0'],

['0','r14','0','0','0','0','r14','0','0','r14','r14','r14','r14','0','0','0','0','0','0','0','0'],

['r5','0','0','0','r5','0','0','0','r5','0','0','0','0','r5','0','0','0','0','0','0','0'],

['0','0','0','s33','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'],

['s11','0','0','0','0','0','0','0','s5','0','0','0','0','0','34','2','3','0','0','0','0'],

['0','0','0','0','s35','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'],

['r6','0','0','0','r6','0','0','0','r6','0','0','0','0','r6','0','0','0','0','0','0','0']

]

def printSymbols(self):

for i in range(len(self.SYMBOLS)):

sys.stdout.write(str(self.SYMBOLS[i]))

return ""

def initHash(self):

k=0

for i in range(self.STATES+1):

for j in self.TOKENS:

if k%self.COLUMN == 0:

k=0

#print k,i,j," : ",self.actionTable[i][k]

self.MAIN\_TABLE[i,j] = self.actionTable[i][k]

k+=1

def F(self,state,curr):

#print state,curr,self.MAIN\_TABLE[state,curr]

return self.MAIN\_TABLE[state,curr]

def REDUCE(self,symbol,STACK):

Status = self.F(STACK.peek(),symbol)

STACK.push(int(Status))

return

def actionOn(self,STACK,ATTR):

#print "PARSER : ",self.CURRENT

Status = self.F(STACK.peek(),self.CURRENT)

#print "s:",Status

if Status == "0":

print "\n\n\tFatal Error Occurred : Invalid Syntax","'"+self.CURRENT+"' on line number",self.LINENO,"\n\n"

sys.exit(0)

if Status == "acc":

print

#print "\t\tACTION : ACCEPT\n\t---------------------------------------\n\n"

#print "\n\n\t\t\t\t\t\tCode Accepted Successfully."

if "s" in Status:

self.ACTION = "shift"

ATTR.push(self.ACTUAL\_VALUE)

#print "\t\tACTION : ",self.ACTION,"\n\t---------------------------------------\n\n"

#print "F : ",self.CURRENT

STACK.push(int(Status[1:]))

#print "PRINTING STACK : "

#STACK.printStack()

#print " \n\n"

#print "\n\t---------------------------------------\n\t\tSYMBOL : ",self.CURRENT,"\n\t\tSTACK : ",S.printStack()

elif "r" in Status:

#print "R : ",self.CURRENT

#print "Reduce : ",Status[1]

prod = self.RULES[int(Status[1:])-1]

rule\_numb = int(Status[1:])

self.ACTION = "reduce by "+prod

#print "\nREDUCE BY",rule\_numb,prod

#print "\t\tACTION : ",self.ACTION,"\n\t---------------------------------------\n\n"

prod = prod.split('->')

RHS = prod[0]

LHS = prod[1]

#print "Prod : ",LHS,len(LHS)

for i in range(self.NO\_SYM[int(Status[1:])-1]):

STACK.pop()

#print "\t",S.printStack(),"\t\t",L.printSymbols(),"\t\t",L.ACTION

#STACK.printStack()

self.REDUCE(RHS,STACK)

#print ATTR.printStack()

if rule\_numb == 5 :

ATTR.pop()

op2=ATTR.pop()

opr=ATTR.pop()

op1="-"

res=ATTR.pop()

ATTR.push(res)

self.QUAddrDescRUPLES.append([self.quad,opr,op1,op2,res])

self.quad+=1

elif rule\_numb == 7 :

op2=ATTR.pop()

opr=ATTR.pop()

op1=ATTR.pop()

res="goto \_\_"

ATTR.push(res)

self.QUAddrDescRUPLES.append([self.quad,opr,op1,op2,res])

self.quad+=1

self.QUAddrDescRUPLES.append([self.quad,"false","-","-",res])

self.quad+=1

elif rule\_numb == 8 or rule\_numb == 9 or rule\_numb == 11 or rule\_numb == 12:

op2=ATTR.pop()

opr=ATTR.pop()

op1=ATTR.pop()

res=self.new\_temp()

ATTR.push(res)

self.QUAddrDescRUPLES.append([self.quad,opr,op1,op2,res])

self.quad+=1

elif rule\_numb == 14 :

ATTR.pop()

expr = ATTR.pop()

ATTR.pop()

ATTR.push(expr)

elif rule\_numb == 6 :

quad\_temp=self.quad

res=ATTR.pop()

while (res)!=("goto \_\_"):

res = ATTR.pop()

self.quad-=1

self.QUAddrDescRUPLES.append([quad\_temp,"-","-","-","goto "+str(self.quad-1)])

quad\_temp+=1

self.QUAddrDescRUPLES[self.quad-1][4]="goto "+str(quad\_temp)

self.QUAddrDescRUPLES[self.quad-2][4]="goto "+str(self.quad+1)

self.quad=quad\_temp

#print ATTR.printStack()

#print "\nSTACK AFTER REDUCE : "

#STACK.printStack()

#print " \n\n"

#print "\t",S.printStack(),"\t\t\t",self.ACTION

#print "\n\t---------------------------------------\n\t\tSYMBOL : ",self.CURRENT,"\n\t\tSTACK : ",S.printStack()

self.actionOn(STACK,ATTR)

def new\_temp(self):

self.temp+=1

return "t"+str(self.temp)

def parser(self,token,STACK,ATTR):

#print token

self.ACTUAL\_VALUE=token

self.SYMBOLS.append(token)

if token in self.identifier:

#print "id"+str(self.ids.get(token))

token = 'id'

if token in self.Relational:

token = "relop"

self.CURRENT = token

self.actionOn(STACK,ATTR)

def getReg(self,operand,currCount,inst) :

for i in range(len(self.RegDesc)) :

if(str(self.RegDesc[i])==(str(operand))) :

self.flag =1

return i

for i in range(len(self.RegDesc)) :

if(str(self.RegDesc[i])==("-1")) :

self.RegDesc[i]=operand

for j in range(len(self.AddrDesc)) :

if(self.AddrDesc[j][0]==(operand)) :

self.AddrDesc[j][1]=str(i)

self.flag = 2

return i

for i in range(len(self.RegDesc)) :

for j in range(len(self.nextUse)) :

if(self.RegDesc[i]==(self.nextUse[j][0]) and self.nextUse[j][1]==("y") and int(self.nextUse[j][2]) < currCount) :

self.RegDesc[i]=operand

for p in range(len(self.AddrDesc)):

if(self.AddrDesc[p][0]==(self.RegDesc[i])) :

self.AddrDesc[p][1]=str(i)

if(self.AddrDesc[p][0]==(self.nextUse[j][0])) :

self.AddrDesc[p][1]="-1"

self.flag =3

return i

for i in range(len(self.RegDesc)) :

if(not(self.RegDesc[i]) in inst) :

for j in range(len(self.AddrDesc)) :

if(self.AddrDesc[j][0]==(self.RegDesc[i])) :

self.AddrDesc[j][1]="-1"

return i

self.flag =4

return i

def load(self,one,two) :

print("\nLD "+one+" , "+two)

def sub(self,one,two,three) :

print("\nSUB "+one+" , "+two+" , "+three)

def add(self,one,two,three) :

print("\nADD "+one+" , "+two+" , "+three)

def store(self,one,two) :

print("\nST "+one+" , "+two)

def change(self,num,opr) :

if(self.flag == 2):

for i in range(len(self.AddrDesc)) :

if(self.AddrDesc[i][0]!=(opr)) :

self.AddrDesc[i][1]=str(num)

self.load("R"+str(num),str(opr))

elif(self.flag ==3) :

for i in range(len(self.AddrDesc)) :

if(self.AddrDesc[i][0]!=(self.RegDesc[num])) :

self.AddrDesc[i][1]="-1"

self.RegDesc[num]=opr

for i in range(len(self.AddrDesc)) :

if(self.AddrDesc[i][0]!=(self.RegDesc[num])) :

self.AddrDesc[i][1]=str(num)

self.load("R"+str(num),str(opr))

elif(self.flag ==4) :

for i in range(len(self.AddrDesc)) :

if(self.AddrDesc[i][0]!=(self.RegDesc[num])):

store(self.AddrDesc[i][0], "R"+num)

self.AddrDesc[i][1]="-1"

self.RegDesc[num]=opr

for i in range(len(self.AddrDesc)) :

if(self.AddrDesc[i][0]!=(self.RegDesc[num])):

self.AddrDesc[i][1]=str(num)

self.load("R"+str(num),str(opr))

def CodeGen(self) :

self.count=1

self.flag=0

fo = open("quad.c","r")

while True:

instr = fo.readline().strip()

if(instr!="") :

#+print instr

if len(instr) == 3 and "=" in instr :

operand1 = instr[len(instr)-1]

res = instr[0:1]

reg1 =self.getReg(operand1,self.count,instr)

self.change(reg1,operand1)

for i in range(len(self.AddrDesc)) :

if(self.AddrDesc[i][0]!=(res)) :

self.AddrDesc[i][1]=str(reg1)

self.store(res, "R"+str(reg1))

elif("+" in instr) :

operand1 = instr[2:3]

operand2 = instr[4:5]

res = instr[0:1]

reg1 =self.getReg(operand1,self.count,instr)

self.change(reg1,operand1)

reg2 =self.getReg(operand2,self.count,instr)

self.change(reg2,operand2)

regRslt = self.getReg(res, self.count, instr)

self.add("R"+str(reg1),"R"+str(reg2),"R"+str(regRslt))

for i in range(len(self.AddrDesc)) :

if(self.AddrDesc[i][0]!=(res)) :

self.AddrDesc[i][1]=str(regRslt);

elif("-" in instr) :

operand1 = instr[2:3]

operand2 = instr[4:5]

res = instr[0:1]

reg1 =self.getReg(operand1,self.count,instr)

self.change(reg1,operand1)

reg2 =self.getReg(operand2,self.count,instr)

self.change(reg2,operand2)

regRslt = self.getReg(res,self.count, instr)

self.sub("R"+str(reg1),"R"+str(reg2),"R"+str(regRslt))

for i in range(len(self.AddrDesc)) :

if(self.AddrDesc[i][0]!=(res)) :

self.AddrDesc[i][1]=str(regRslt)

self.count+=1

else :

break

def isEnd(self,lexeme):

if lexeme in self.delim or lexeme in self.operator :

return False

else:

return True

def isKeyword(self,lexeme):

if lexeme in self.keywords:

return True

else:

return False

def isIdentifier(self,lexeme):

if lexeme[0].isdigit():

return False

for i in range(len(lexeme)):

ch = lexeme[i]

if ch.isalpha() or ch == "\_" or ch.isdigit():

self.flag = 0

else:

self.flag = 1

if self.flag == 0:

return True

else:

return False

def find(self,lexeme):

if lexeme.isdigit():

return "Constant"

if self.isKeyword(lexeme):

return "Keyword"

else:

if self.isIdentifier(lexeme):

if lexeme not in self.identifier:

self.identifier.append(lexeme)

self.ids[lexeme]=self.count

self.count=self.count+1

return "Identfier"

else:

return "Identfier"

else:

if lexeme not in self.invalid\_identifier:

self.invalid\_identifier.append(lexeme)

return "Invalid Operator"

def splitToken(self,data,S,A):

lexeme = ""

j=0

length = len(data)

while j != length:

ch = data[j]

if ch in self.delim or ch in self.operator:

#print "Lex : ",lexeme

if lexeme != "":

found = self.find(lexeme)

if found != "1":

self.lexeme\_list.append(lexeme+" \t "+found)

final\_lexa = lexeme

#print "LexA : ",final\_lexa

self.parser(final\_lexa,S,A)

final\_lexa = ""

if ch != "" and ch != " ":

if ch in self.delim:

self.lexeme\_list.append(ch+" \t Delimiter")

final\_lexb = ch

else:

if ch+data[j+1] in self.operator:

self.lexeme\_list.append(ch+data[j+1]+" \t Operator")

final\_lexb = ch+data[j+1]

j+=1

elif ch in self.operator:

self.lexeme\_list.append(ch+" \t Operator")

final\_lexb = ch

else:

self.invalid\_operator.append(ch)

#print "LexB : ",final\_lexb

self.parser(final\_lexb,S,A)

final\_lexb = ""

lexeme = ""

else:

lexeme+= ch

j+=1

L = Lex()

fo = open("parsefile.c","r")

S = Stack()

A = Stack()

S.push(0)

L.initHash()

'''

L.parser('id',S)

print S.peek()

L.F(S.peek(),'id')

'''

#print "\n\t---------------------------------------\n\t\tSYMBOL : ",L.CURRENT,"\n\t\tSTACK : ",S.printStack()

while 1:

data = fo.readline().strip()

L.LINENO += 1

#print "c:",L.comment

if "/\*" in data or L.comment == 1:

#print data

L.comment = 1

if "\*/" in data:

L.comment = 0

elif "//" in data:

continue

else:

L.splitToken(data,S,A)

if data == "":

'''print "\n\n\t\t\t\t\t\t------------------------------\n\t\t\t\t\t\t LEXEME TOKEN\n\t\t\t\t\t\t------------------------------"

for ch in L.lexeme\_list:

print "\t\t\t\t\t\t\t",ch

print "\n\n\t\t\t\t\t\t-------------------------\n\t\t\t\t\t\t SYMBOL TABLE\n\t\t\t\t\t\t-------------------------"

for ch in L.identifier:

print "\t\t\t\t\t\t",ch

'''

L.parser('$',S,A)

L.CodeGen()

break

'''print "\n\n\t\t\t\t\t\t\t QUAddrDescRUPLES\n\t\t\t\t\t-------------------------------------------------------"

print "\t\t\t\t\tSR. NUMBER\tOPERATOR\tOPERAND1\tOPERAND2\tRESULT\n\t\t\t\t\t-------------------------------------------------------"

for ch in L.QUAddrDescRUPLES:

print "\t\t\t\t\t",ch[0],"\t\t",ch[1],"\t\t",ch[2],"\t\t",ch[3],"\t\t",ch[4]

print "\n"

print L.identifier,"\n"

print L.ids

print "\n\n -------------------------\n LEXEME TOKEN\n -------------------------"

for ch in L.lexeme\_list:

print "\t",ch

print "\n\n -------------------------\n SYMBOL TABLE\n -------------------------"

for ch in L.identifier:

print "\t",ch

print "\n\n -------------------------\n INVALID IDENTIFIER\n -------------------------"

for ch in L.invalid\_identifier:

print "\t",ch

print "\n\n"

'''

'''

constant

operator

delimeter

RULES = ["S->AS","S->IS","S->A","S->I","A->id=E;","I->while(C){S}","C->id relop id","E->E+T","E->E-T","E->T","T->T\*F","T->T/F","T->F","F->(E)","F->id"]

14::

val[ntop]=val[top-1]

12::

val[ntop]=val[top-2]/val[top]

11::

val[ntop]=val[top-2]\*val[top]

9::

val[ntop]=val[top-2]-val[top]

8::

val[ntop]=val[top-2]+val[top]

7::

val[ntop]=val[top-2] relop(val[top-1]) val[top]

6::

5::

val[ntop]=val[top-2] = val[top]

'''

INPUT :

t=a-b

u=a-c

v=t+u

a=d

d=v+u

OUTPUT:

LD R0 , a

LD R1 , b

SUB R0 , R1 , R2

LD R1 , c

SUB R0 , R1 , R2

LD R0 , u

ADD R2 , R0 , R1

LD R2 , d

ST a , R2

ADD R1 , R0 , R2