07-Apr-22

# **Experiment 11 - Intermediate code generation- Quadruple, Triple, Indirect Triple**

Dhawal Patil

RA1911003010575

CSE A2

Aim:

A program to implement Intermediate code generation – Quadruple, Triple, Indirect Triple.

Algorithm:

The algorithm takes a sequence of three-address statements as input. For each three address

statements of the form a:= b op c perform the various actions. These are as follows:

1. Invoke a function getreg to find out the location L where the result of computation b op c should be stored.
2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction MOV y' , L to place a copy of y in L.
3. Generate the instruction OP z' , L where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptors.
4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those register will no longer contain y or z.

Code:

OPERATORS = set(['+', '-', '\*', '/', '(', ')'])

PRI = {'+': 1, '-': 1, '\*': 2, '/': 2}

def infix\_to\_postfix(formula):

stack = []

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()

else:

while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:

output += stack.pop()

stack.append(ch)

while stack:

output += stack.pop()

return output

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()

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)

while op\_stack:

op = op\_stack.pop()

a = exp\_stack.pop()

b = exp\_stack.pop()

exp\_stack.append(op+b+a)

return exp\_stack[-1]

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)

def Quadruple(pos):

stack = []

op = []

x = 1

for i in pos:

if i not in OPERATORS:

stack.append(i)

elif i == '-':

op1 = stack.pop()

stack.append("t(%s)" % x)

print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(

i, op1, "(-)", " t(%s)" % x))

x = x+1

if stack != []:

op2 = stack.pop()

op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(

"+", op1, op2, " t(%s)" % x))

stack.append("t(%s)" % x)

x = x+1

elif i == '=':

op2 = stack.pop()

op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i, op2, "(-)", op1))

else:

op1 = stack.pop()

op2 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(

i, op2, op1, " t(%s)" % x))

stack.append("t(%s)" % x)

x = x+1

def Triple(pos):

stack = []

op = []

x = 0

for i in pos:

if i not in OPERATORS:

stack.append(i)

elif i == '-':

op1 = stack.pop()

stack.append("(%s)" % x)

print("{0:^4s} | {1:^4s} | {2:^4s}".format(i, op1, "(-)"))

x = x+1

if stack != []:

op2 = stack.pop()

op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s}".format("+", op1, op2))

stack.append("(%s)" % x)

x = x+1

elif i == '=':

op2 = stack.pop()

op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s}".format(i, op1, op2))

else:

op1 = stack.pop()

if stack != []:

op2 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s}".format(i, op2, op1))

stack.append("(%s)" % x)

x = x+1

def IndirectTriple(pos):

stack = []

op = []

x = 0

c = 0

for i in pos:

if i not in OPERATORS:

stack.append(i)

elif i == '-':

op1 = stack.pop()

stack.append("(%s)" % x)

print("{0:^4s} | {1:^4s} | {2:^4s} | {3:^5d}".format(i, op1, "(-)", c))

x = x+1

if stack != []:

op2 = stack.pop()

op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s} | {3:^5d}".format(

"+", op1, op2, c))

stack.append("(%s)" % x)

x = x+1

c = c+1

elif i == '=':

op2 = stack.pop()

op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s} | {3:^5d}".format(i, op1, op2, c))

c = c+1

else:

op1 = stack.pop()

if stack != []:

op2 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s} | {3:^5d}".format(

i, op2, op1, c))

stack.append("(%s)" % x)

x = x+1

c = c+1

z = 35

print("Statement|Location")

for i in range(0, c):

print("{0:^4d} |{1:^4d}".format(i, z))

z = z+1

print("=====Quadruple=====")

print("Op | Src1 | Src2| Res")

Quadruple(pos)

print("=====Tripple=====")

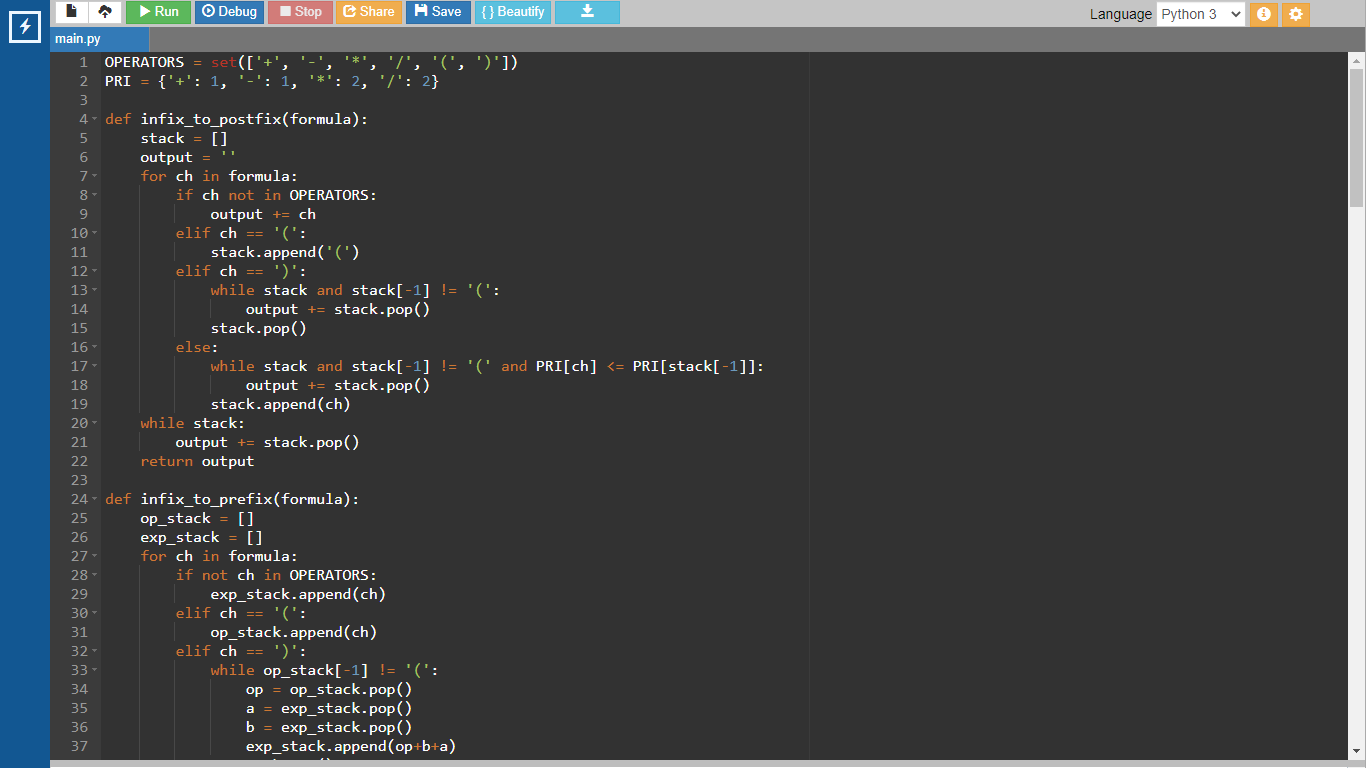
print("Op | Src1 | Src2")

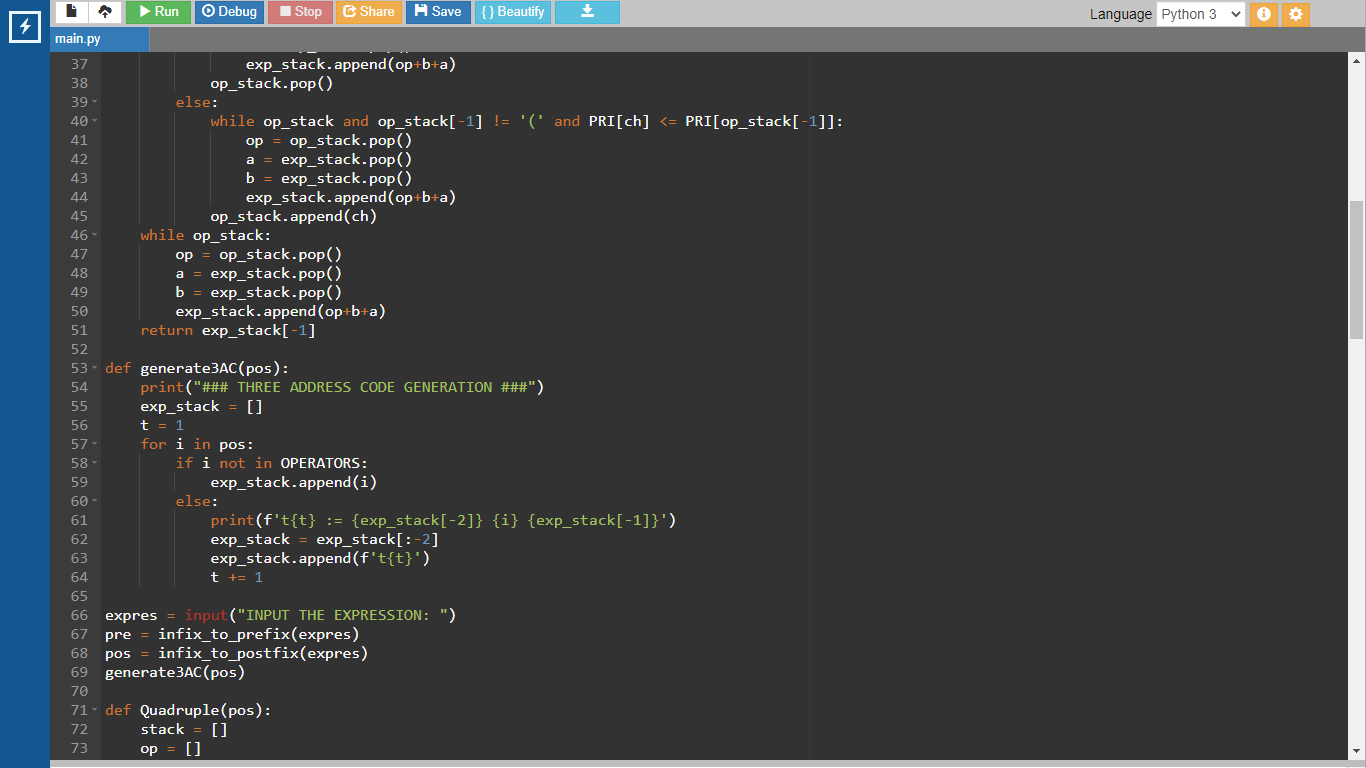
Triple(pos)

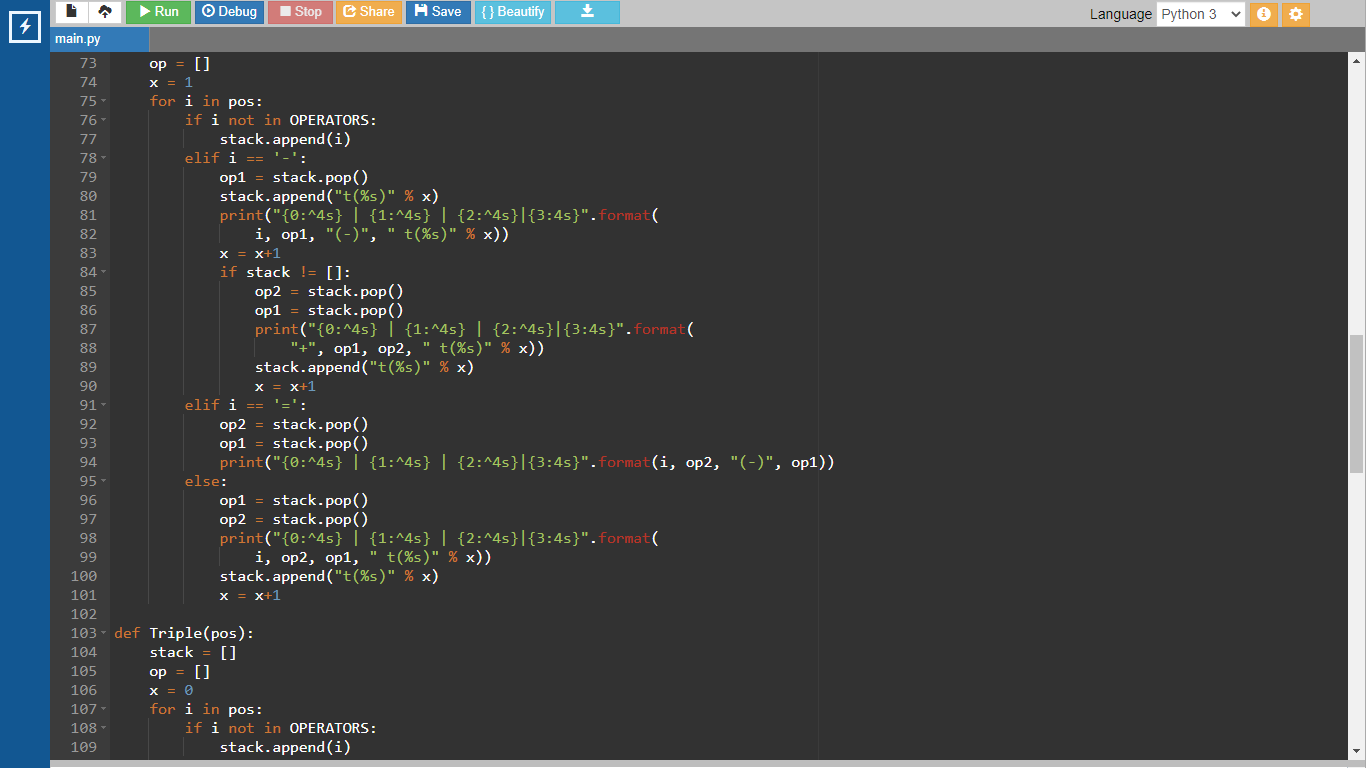
print("====Indirect Tripple====")

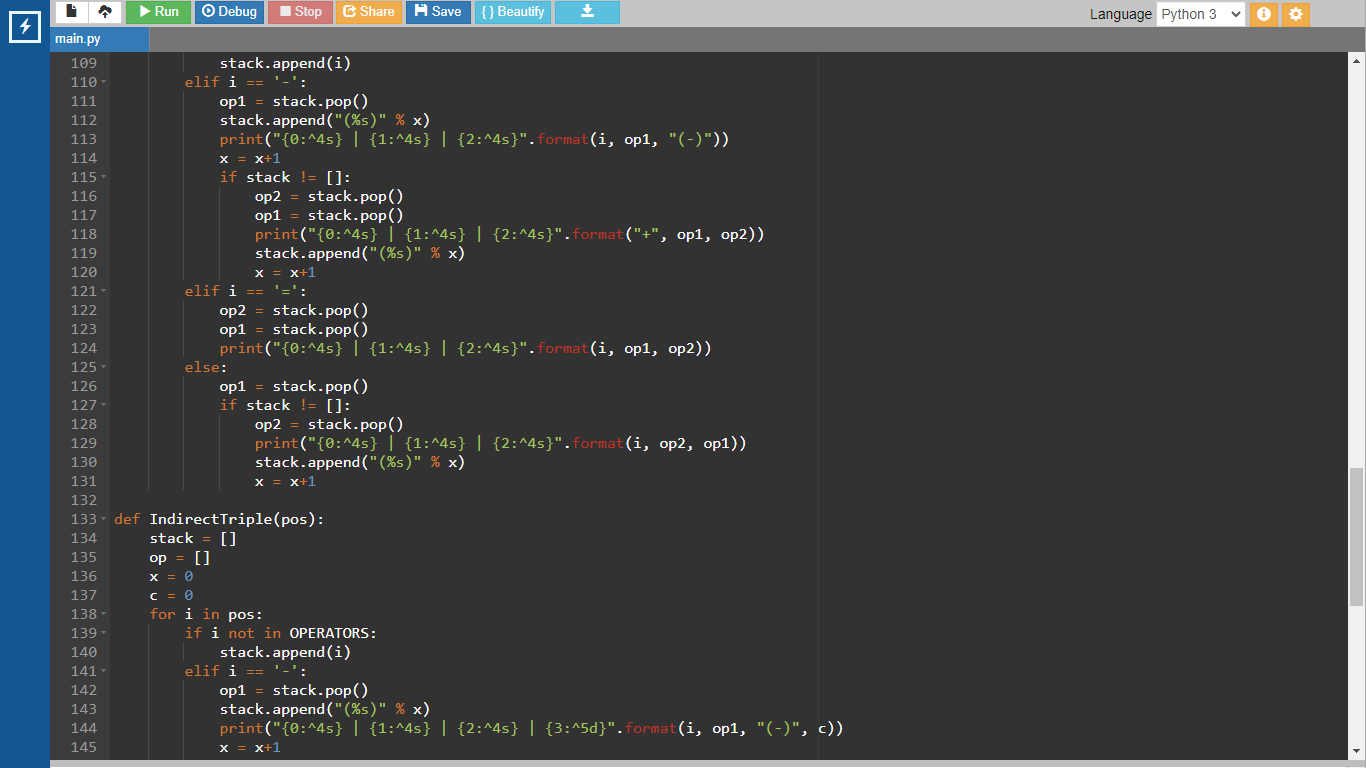
print("Op | Src1 | Src2 |Statement")

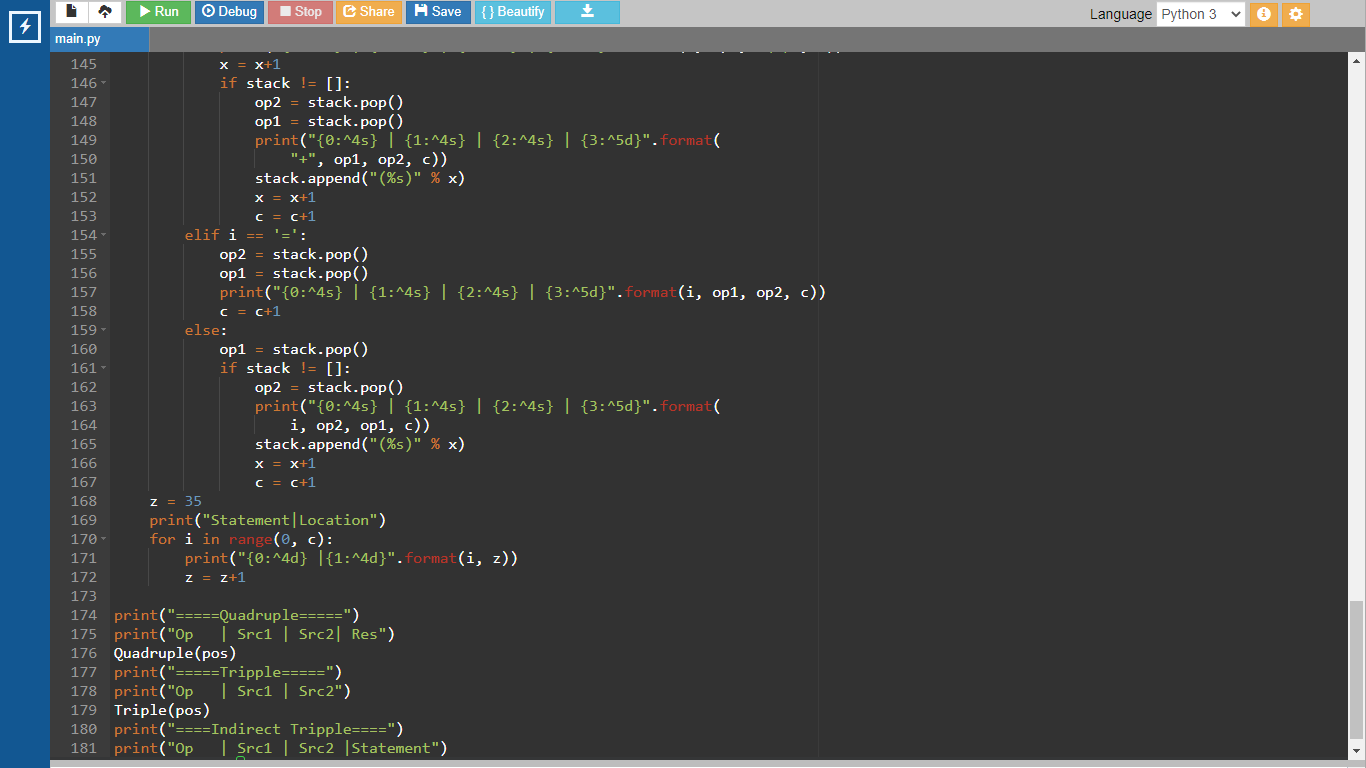
IndirectTriple(pos)

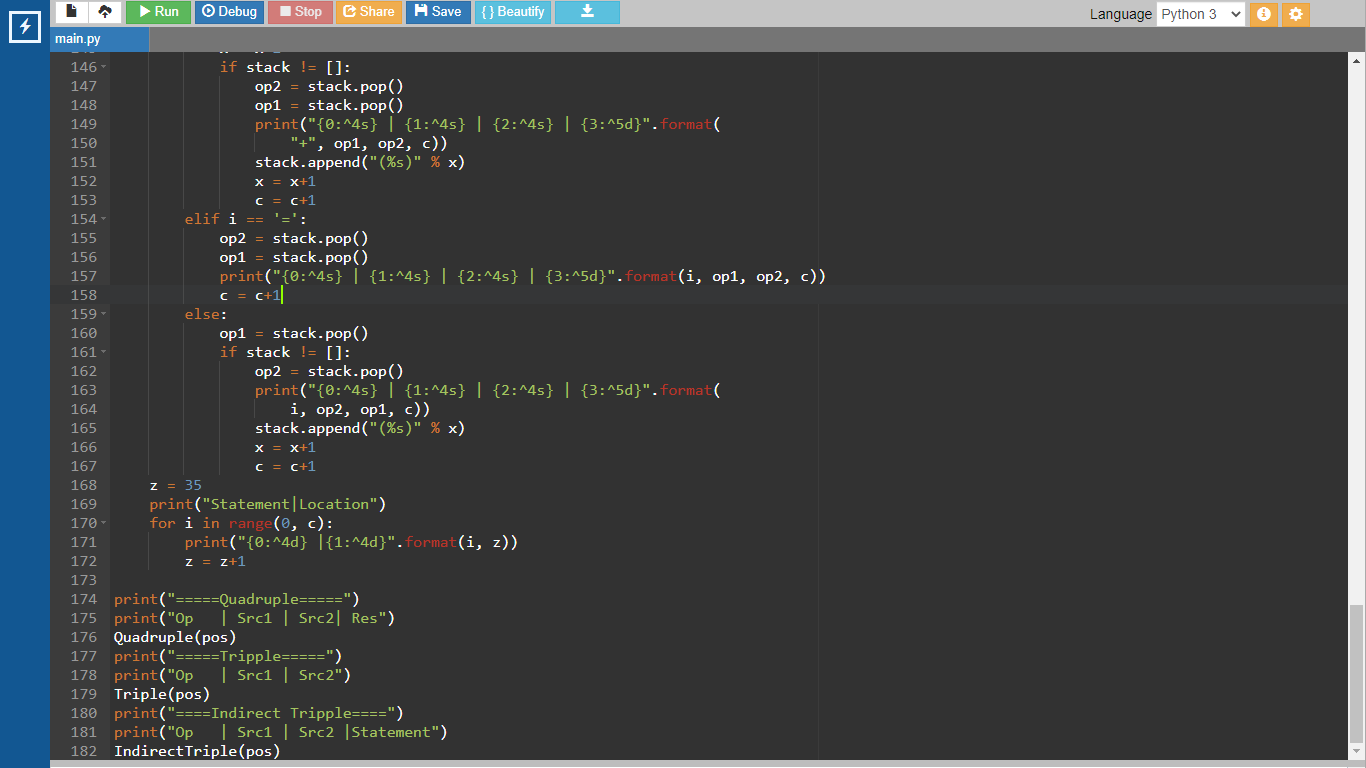




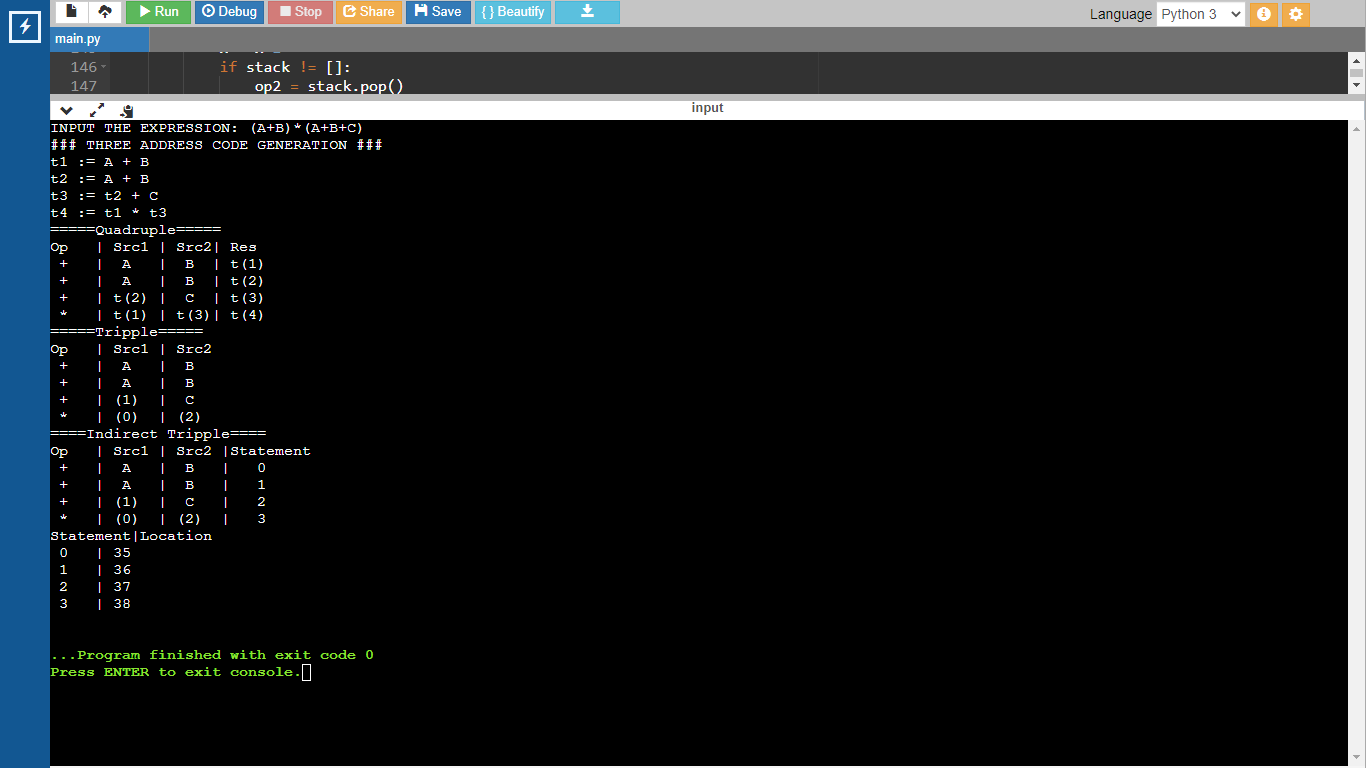








Output:



Result:

The program was successfully compiled and run.