Exp:11: Intermediate Code Generation: Quadruple, Triple and Indirect Triple

AIM: To design a code to intermediate code generation Quadruple, Triple and Indirect Triple

LANGUAGE USED: Python 3

ALGORITHM/PROCEDURE: -

Infix expression to Prefix:

- First, reverse the given infix expression.
- Scan the characters one by one.
- If the character is an operand, copy it to the prefix notation output.
- If the character is a closing parenthesis, then push it to the stack.
- If the character is an opening parenthesis, pop the elements in the stack until we find the corresponding closing parenthesis.
- If the character scanned is an operator
  - If the operator has precedence greater than or equal to the top of the stack, push the operator to the stack.
  - If the operator has precedence lesser than the top of the stack, pop the operator and output it to the prefix notation output and then check the above condition again with the new top of the stack.
- After all the characters are scanned, reverse the prefix notation output.

Infix expression to Postfix:

- Scan the given infix expression from left to right.
- If the scanned character is an operand, then output it.
- Else,

  If the precedence of the scanned operator is greater than the precedence of the operator in the top of the stack (or the stack is empty or if the stack contains a '('), push it.

  Else, Pop all operators from the stack whose precedence is greater than or equal to that of the scanned operator. Then, push the scanned operator to the top of the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

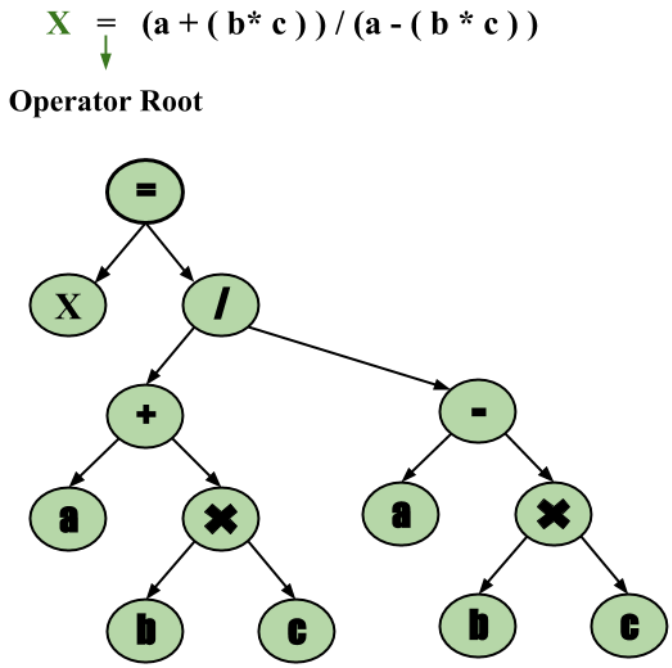
- If the scanned character is '(', push it to the stack.
- If the scanned character is ')', pop the stack and and output characters until '(' is encountered, and discard both the parenthesis.
- Repeat steps 2-5 until infix expression is scanned completely.

- Print the output.
- Pop and output from the stack.

Three address code generation:

A statement involving no more than three references (two for operands and one for the result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of form x = y op z, here x, y, z will have the address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement.
Eg: The three-address code for the expression a + b * c + d:
T 1 = b * c
T 2 = a + T 1
T 3 = T 2 + d
T1, T2, T3 are temporary variables.

SPACE TREE DIAGRAM/ EXPLANATION:



SOURCE 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() # pop '('
        else:
            while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:
                output += stack.pop()
            stack.append(ch)
    while stack:
            output += stack.pop()
    print(f'POSTFIX: {output}')
    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() # 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 )
    print(f'PREFIX: {exp_stack[-1]}')
    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 == '=':
```

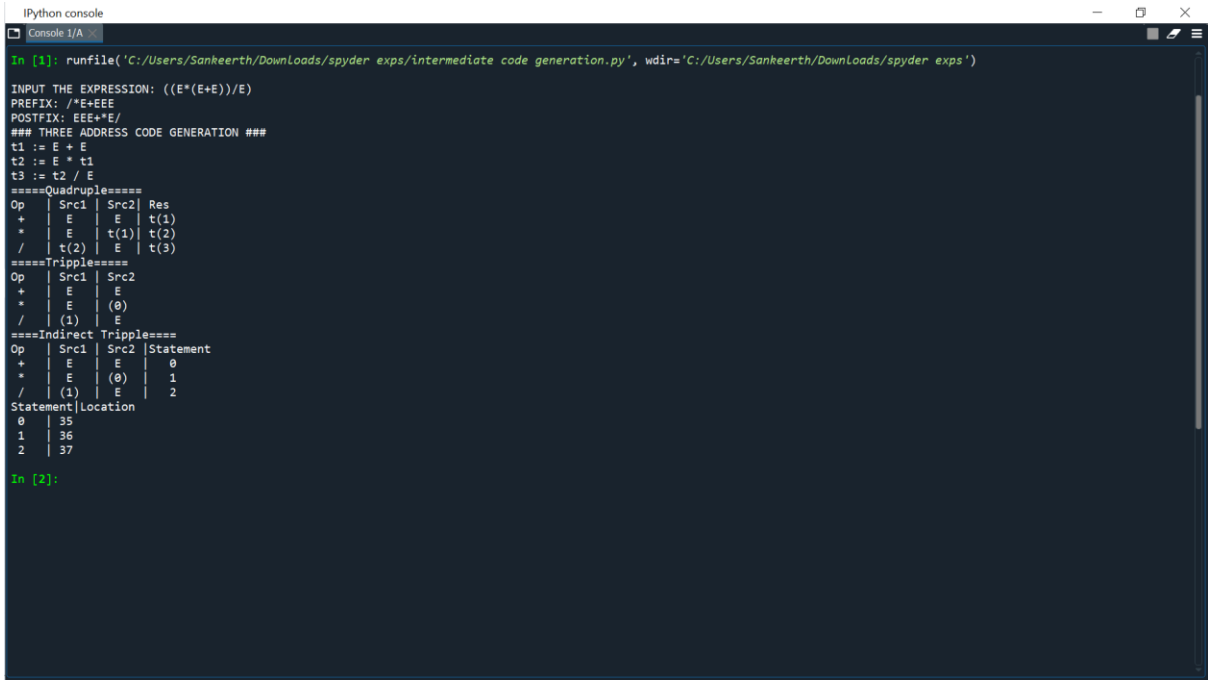
```
            op2 = stack.pop()
            op1 = stack.pop()
            print("{0:^4s} | {1:^4s} | {2:^4s} | {3:^5d}".format(i, op1, op2, c))
            c = c+1
        else:
            op1 = stack.pop()
            if stack != []:
                op2 = stack.pop()
                print("{0:^4s} | {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:

INPUT THE EXPRESSION: ((E(E+E))/E)*
*PREFIX: /*E+EEE*
*POSTFIX: EEE+*E/*
THREE ADDRESS CODE GENERATION
t1 := E + E
*t2 := E * t1*
t3 := t2 / E
=====Quadruple=====
Op | Src1 | Src2| Res
+ | E | E | t(1)
** | E | t(1)| t(2)*
/ | t(2) | E | t(3)
=====Tripple=====
Op | Src1 | Src2

+ | E | E
** | E | (0)*
/ | (1) | E
====Indirect Tripple====
Op | Src1 | Src2 |Statement
+ | E | E | 0
** | E | (0) | 1*
/ | (1) | E | 2
Statement|Location
0 | 35
1 | 36
2 | 37



RESULT: Therefore, we successfully implemented intermediate code generation (three address code – Quadruple, Triple, Indirect Triple).