EXP 11
28.04.21

Intermediate code generation – Quadruple, Triple, Indirect triple

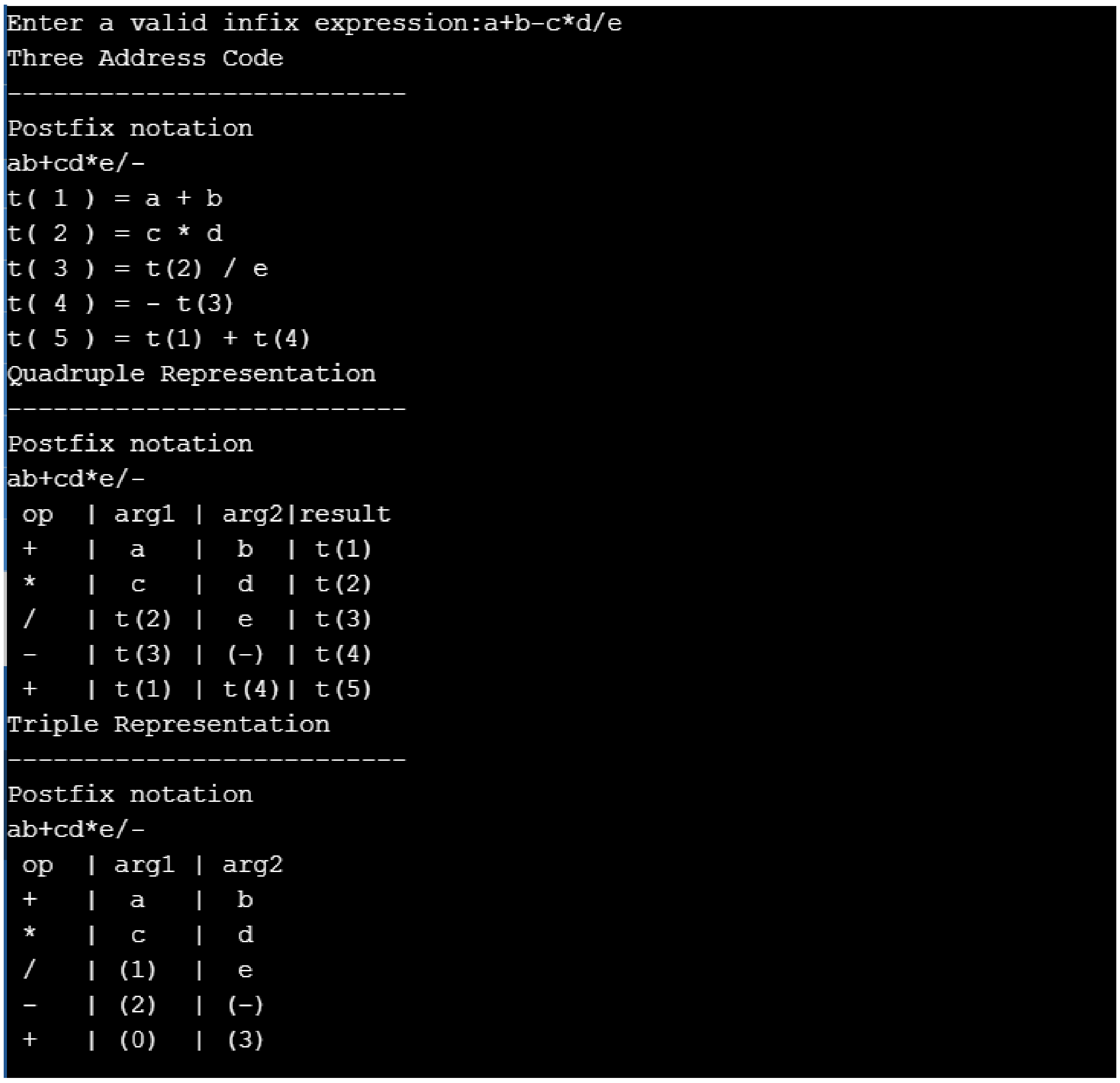
AIM: To write a code to perform intermediate code generation Quadruple, Triple, Indirect triple

Code:

```
import StackClass


def T_A_C(exp):
    stack = []
    x = 1
    # an instance of the StackClass module
    obj = StackClass.Conversion(len(exp))
    # an instance that converts a given infix notation to post fix
    postfix = obj.infixToPostfix(exp)
    for i in postfix:
        if i in "abcdefghijklmnopqrstuvwxyz" or i in "0123456789":
            stack.append(i)
        elif i == '-':
            op1 = stack.pop()
            print("t(", x, ")", "=", i, op1)
            stack.append("t(%s)" % x)
            x = x + 1
            if stack != []:
                op2 = stack.pop()
                op1 = stack.pop()
                print("t(", x, ")", "=", op1, "+", op2)
                stack.append("t(%s)" % x)
                x = x + 1
        elif i == '=':
            op2 = stack.pop()
            op1 = stack.pop()
            print(op1, i, op2)

        else:
            op1 = stack.pop()
            if stack != []:
                op2 = stack.pop()
                print("t(", x, ")", "=", op2, i, op1)
                stack.append("t(%s)" % x)
                x = x + 1
```

```
def Quadruple(exp):
    stack = []
    op = []
    x = 1
    # an instance of the StackClass module
    obj = StackClass.Conversion(len(exp))
    # an instance that converts a given infix notation to post fix
    postfix = obj.infixToPostfix(exp)
    print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format('op', 'arg1', 'arg2', 'result'))
    for i in postfix:
        if i in "abcdefghijklmnopqrstuvwxyz" or i in "0123456789":
            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(exp):
    stack = []
    op = []
    x = 0
    # an instance of the StackClass module
    obj = StackClass.Conversion(len(exp))
    # an instance that converts a given infix notation to postfix
    postfix = obj.infixToPostfix(exp)
    print("{0:^4s} | {1:^4s} | {2:^4s}".format('op', 'arg1', 'arg2'))
    for i in postfix:
        if i in "abcdefghijklmnopqrstuvwxyz" or i in "0123456789":
            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


exp = input("Enter a valid infix expression:")
print("Three Address Code")
print("-------------------------")
T_A_C(exp)
print("Quadruple Representation")
print("-------------------------")
Quadruple(exp)
print("Triple Representation")
print("-------------------------")
Triple(exp)




# Class to convert the expression
class Conversion:
    # Constructor to initialize the class variables
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        # This array is used a stack
        self.array = []
        # Precedence setting
        self.output = []
        self.precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}

    # check if the stack is empty
    def isEmpty(self):
        return True if self.top == -1 else False

    # Return the value of the top of the stack
    def peek(self):
        return self.array[-1]

    # Pop the element from the stack
    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"

    # Push the element to the stack
    def push(self, op):
        self.top += 1
        self.array.append(op)

        # A utility function to check is the given character

    # is operand
    def isOperand(self, ch):
        return ch.isalpha()

    # Check if the precedence of operator is strictly
    # less than top of stack or not
    def notGreater(self, i):
        try:
            a = self.precedence[i]
            b = self.precedence[self.peek()]
            return True if a <= b else False
        except KeyError:
            return False

    # The main function that converts given infix expression
    # to postfix expression
    def infixToPostfix(self, exp):
        # Iterate over the expression for conversion
        for i in exp:
            # If the character is an operand,
            # add it to output
            if self.isOperand(i):
                self.output.append(i)
            # If the character is an '(', push it to stack
            elif i == '(':
                self.push(i)

            # If the scanned character is an ')', pop and
            # output from the stack until and '(' is found
            elif i == ')':
                while ((not self.isEmpty()) and self.peek() != '('):
                    a = self.pop()
                    self.output.append(a)
                if (not self.isEmpty() and self.peek() != '('):
                    return -1
                else:
                    self.pop()

            # An operator is encountered
            else:
                while (not self.isEmpty() and self.notGreater(i)):
                    self.output.append(self.pop())
                self.push(i)
                # pop all the operator from the stack
        while not self.isEmpty():
            self.output.append(self.pop())
        print("Postfix notation")
        print("".join(self.output))
        return "".join(self.output)
```

Output:



```
Enter a valid infix expression:a+b-c*d/e
Three Address Code
--------------------------
Postfix notation
ab+cd*e/-
t( 1 ) = a + b
t( 2 ) = c * d
t( 3 ) = t(2) / e
t( 4 ) = - t(3)
t( 5 ) = t(1) + t(4)
Quadruple Representation
--------------------------
Postfix notation
ab+cd*e/-
 op  | arg1 | arg2|result
 +   |  a   |  b  | t(1)
 *   |  c   |  d  | t(2)
 /   | t(2) |  e  | t(3)
 -   | t(3) | (-) | t(4)
 +   | t(1) | t(4)| t(5)
Triple Representation
--------------------------
Postfix notation
ab+cd*e/-
 op  | arg1 | arg2
 +   |  a   |  b
 *   |  c   |  d
 /   | (1)  |  e
 -   | (2)  | (-)
 +   | (0)  | (3)
```

Result:intermediate code generation Quadruple, Triple, Indirect triple was successfully implemented

RA1811003010303 RAGHU B