Name : JAY PATEL

```
import tkinter as tk
import abc
import math
import unittest

# Implementing the Interpreter Pattern.
# Expression 'Abstract' Class will be inherited by TerminalExpression and NonTerminalExpression
Class.
# TerminalExpression Class will be all numbers in the Expression.
# NonTerminalExpression Classes are Add, Subtract, Multiply, Divide, Log, and Sin.
# the interpret method of Abstract class is overwritten in every Child Class.

class Expression:
    @abc.abstractmethod
    def interpret(self):
        pass

# this class stores only number, all other expression are interpreted in NonTerminal Class

class TerminalExpression(Expression):
    def __init__(self, number):
        self.__number = float(number)

    def interpret(self):
        return self.__number

# The NonTerminal Classes returns the object of TerminalExpression Class after performing
respective calculations.

class AddExpression(Expression):
    def __init__(self, firstExpression, secondExpression):
        self.__firstExpression = firstExpression
        self.__secondExpression = secondExpression

    def interpret(self):
        return self.__firstExpression.interpret() + self.__secondExpression.interpret()

class SubtractExpression(Expression):
    def __init__(self, firstExpression, secondExpression):
        self.__firstExpression = firstExpression
        self.__secondExpression = secondExpression

    def interpret(self):
        return self.__secondExpression.interpret() - self.__firstExpression.interpret()

class MultiplyExpression(Expression):
    def __init__(self, firstExpression, secondExpression):
        self.__firstExpression = firstExpression
        self.__secondExpression = secondExpression

    def interpret(self):
        return self.__firstExpression.interpret() * self.__secondExpression.interpret()

class DivideExpression(Expression):
    def __init__(self, firstExpression, secondExpression):
        self.__firstExpression = firstExpression
        self.__secondExpression = secondExpression

    def interpret(self):
        return self.__secondExpression.interpret() / self.__firstExpression.interpret()

class SinFunction(Expression):
    def __init__(self, firstExpression):
        self.__firstExpression = firstExpression

    def interpret(self):
        return math.sin(self.__firstExpression.interpret())

class LogFunction(Expression):
    def __init__(self,firstExpression):
        self.__firstExpression = firstExpression

    def interpret(self):
        return math.log(self.__firstExpression.interpret(),2)
```

```
# isOperator will checks whether any operator is encountered or not
# getExpressionObject returns the object of one of NonTerminal Class as per requirement

class ParserUtil:
    def isOperator(self, symbol):
        return (symbol == "+" or symbol == "-" or symbol == "*" or symbol is "/" or symbol == "sin"
 or symbol == "lg")

    def getExpressionObject(self,firstExpression, secondExpression, symbol):
        if symbol == "+":
            return AddExpression(firstExpression, secondExpression)
        elif symbol == "-":
            return SubtractExpression(firstExpression, secondExpression)
        elif symbol == "*":
            return MultiplyExpression(firstExpression, secondExpression)
        elif symbol == "/":
            return DivideExpression(firstExpression, secondExpression)
        elif symbol == "sin":
            return SinFunction(firstExpression)
        elif symbol == "lg":
            return LogFunction(firstExpression)

# Expression Parser is class where all postfix Expression is Evaluated.
# by calling the getExpressionObject function of ParserUtil Class it gets the required NonTerminal
Class
# and after it calling the interpret method it creates the TerminalObject of Result and stores in
List.
# note that as the log and sin function has only one argument, it pops only one item from List.

class ExpressionParser:

    def parser(self, str):
        self.list = []
        tokenList = str.split(" ")
        for symbol in tokenList:
            if ParserUtil.isOperator(self, symbol):
                if (symbol == 'sin' or symbol == 'lg'):
                    firstExpression = self.list.pop()
                    secondExpression = None
                else:
                    firstExpression = self.list.pop()
                    secondExpression = self.list.pop()
                self.operator = ParserUtil.getExpressionObject(self, firstExpression,
secondExpression, symbol)
                self.result = self.operator.interpret()
                resultExpression = TerminalExpression(self.result)
                self.list.append(resultExpression)
            else:
                numberExpression = TerminalExpression(symbol)
                self.list.append(numberExpression)
        result = self.list.pop()
        return result

# Implementing Observer Pattern, where Cell class will be both Observer and Subject at a time.
# AddObserver will add the dependent cell of particular cell into the Observers of that cell
# notifyObserver will call update method on all the dependent cell of current Cell Object.

class Observable:
    def __init__(self):
        self.observers = []
        self.changed = 0

    def addObserver(self, observer):
        if observer not in self.observers:
            self.observers.append(observer)

    def deleteObserver(self, observer):
        self.observers.remove(observer)

    def notifyObserver(self):
        for observer in self.observers:
            observer.update()

# Observer Class is Abstract as it has only one update method which is called when the Subject has
Changed.
```

```
class Observer:
    @abc.abstractmethod
    def update(self):
        pass

#this function returns the Alphabetical letter which is used name the individual cell of the
SpreadSheet.
def cellname(i, j):
    return f'{chr(ord("A") + j)}'

# Cell Class plays as Originator in Memento Pattern and also has a State in State Pattern
# Cell Class has all the attributes and function required for the individual Cell to operate.

class Cell(Observer, Observable):

    def __init__(self,i,j,siblings,currentState,careTaker,parent):
        super().__init__()
        self.row = i
        self.col = j
        self.formula = '0'
        self.value = 0
        self.currentcellState = currentState
        self.siblings =siblings
        self.name = cellname(i,j)
        self.var = tk.StringVar()
        self.careTaker = careTaker
        entry = self.widget = tk.Entry(parent,textvariable = self.var)
        entry.bind('<Return>',self.update_cell)


    # update_cell is called whenever the user presses the Enter key.
    # careTaker will take the snapshot and then update_cell method will evaluate postfix expression
 in update() call.
    # notifyObserver() will update the dependency of the current cell.

    def update_cell(self, event):
        self.careTaker.save(self.createMemento())
        self.formula = self.var.get()
        self.update()
        self.notifyObserver()

    # the main function of update method is to evaluate the postfix expression and add the
Obsersers,
    # which are used in the postfix Expression and replace the cell varaiable with actual value of
Cell.
    def update(self):
        variable_replace = self.formula.split(" ")
        count = 0
        for variable in variable_replace:
            if(variable[0] == '$'):
                cell_variable = variable[1]
                self.siblings[cell_variable].addObserver(self)
                variable_replace[count] = str(self.siblings[cell_variable].value)
                count += 1
        seperator = " "
        self.formula_1 = seperator.join(variable_replace)
        self.value = self.calculate(self.formula_1)
        self.notifyObserver()

        # below code checks the state of the Spreadsheet and display the value or formula as per
State.
        if(isinstance(self.currentcellState, FormulaViewState)):
            self.var.set(self.formula)
        elif(isinstance(self.currentcellState, ValueViewState)):
            self.var.set(self.value)

    # calculate function only evalutes the postfix Expression and returns the result(number) after
solving.
    def calculate(self, formula):
        expressionParser = ExpressionParser()
        result = expressionParser.parser(formula)
        return  result.interpret()

    # will return the Memento Object when the CareTaker will call the createMemento function,(
```

```
Memento Pattern).
    def createMemento(self):
        return Memento([self.name,self.value,self.formula])

# Memento Class stores the current snapshot, but is Immutable, so can not modify the value of
attributes.
class Memento:
    def __init__(self,cellInfo):
        self.cellInfo = cellInfo

    def get_CellInfo(self):
        return self.cellInfo

# CareTaker is responsible for storing the Memento Objects which are used to restore the state of
the Cells.
class CareTaker:
    mementoList = []
    def save(self, mementoObject):
        self.mementoList.append(mementoObject)

    def undo(self):
        return self.mementoList.pop().get_CellInfo()

# SpreadSheet class is used as main Class where each cell in SpreadSheet is created.
# It also has CareTaker object which is passed to each cell when they are created.
# it has undo_cell function which calls the undo function of CareTaker and restore the previous
state of cell.

class SpreadSheet(tk.Frame,Cell):
    def __init__(self,rows, cols, master=None):
        super().__init__(master)
        self.rows = rows
        self.cols = cols
        self.cells = {}
        self.currentViewState = FormulaViewState()
        self.careTaker = CareTaker()
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        self.cellframe = tk.Frame(self)
        self.cellframe.pack(side='top')

        # creates label as column name.
        for i in range(self.cols):
            label = tk.Label(self.cellframe, text="$" + chr(ord('A') + i))
            label.grid(row=0, column=i)

        # initialize each cell where careTaker and current state(FormulaViewState) are passed as
argument.
        for i in range(self.rows):
            for j in range(self.cols):
                cell = Cell(i,j,self.cells,self.currentViewState,self.careTaker,self.cellframe)
                self.cells[cell.name] = cell
                cell.widget.grid(row = i+1, column = j)

        self.btn_2 =  tk.Button(self.cellframe,text='Value View', width = 15, command=self.on_click
)
        btn_1 = tk.Button(self.cellframe, text='Undo', width = 15, command = self.undo_cell )
        self.btn_2.grid(row = 2, column = 6)
        btn_1.grid(row =2 , column = 4)

    # the careTaker object returns the Memento Object from Memento List which is used to restore
the state of cell.
    def undo_cell(self):
        cellObj = self.careTaker.undo()
        self.cells[cellObj[0]].formula = str(cellObj[2])
        self.cells[cellObj[0]].update()
        self.cells[cellObj[0]].notifyObserver()

    # on_click function the state of SpreadSheet changes from FormulaView to ValueView as per the
current State.
    def on_click(self):
        self.currentViewState = self.currentViewState.on_click()
        if isinstance(self.currentViewState, FormulaViewState):
```

```
            for cell in self.cells:
                self.cells[cell].var.set(self.cells[cell].formula)
                self.cells[cell].currentcellState = self.currentViewState
            self.btn_2['text'] = 'Value View'
        elif isinstance(self.currentViewState, ValueViewState):
            for cell in self.cells:
                self.cells[cell].var.set(self.cells[cell].value)
                self.cells[cell].currentcellState = self.currentViewState
            self.btn_2['text'] = 'Expression View'


class ViewState:
    @abc.abstractmethod
    def on_click(self):
        pass

class FormulaViewState(ViewState):
    def on_click(self):
            return ValueViewState()

class ValueViewState(ViewState):
    def on_click(self):
            return FormulaViewState()

# UNIT TESTING

# Testing Class for execution of NonTerminal Class in Postfix Expression
class TerminalExpressionClass(unittest.TestCase):
    def test_interpret(self):
        obj = TerminalExpression(56)
        self.assertEqual(56, obj.interpret())

class AddExpressionClass(unittest.TestCase):
    def test_addExpression(self):
        value1 = TerminalExpression(1)
        value2 = TerminalExpression(1)
        obj = AddExpression(value1,value2)
        self.assertEqual(2, obj.interpret())

class SubtractExpressionClass(unittest.TestCase):
    def test_subtractExpression(self):
        value1 = TerminalExpression(16)
        value2 = TerminalExpression(6)
        obj = SubtractExpression(value2,value1)
        self.assertEqual(10, obj.interpret())

class MultiplyExpressionClass(unittest.TestCase):
    def test_multiplyExpression(self):
        value1 = TerminalExpression(10)
        value2 = TerminalExpression(10)
        obj = MultiplyExpression(value1,value2)
        self.assertEqual(100,obj.interpret())

class DivideExpressionClass(unittest.TestCase):
    def test_multiplyExpression(self):
        value1 = TerminalExpression(10)
        value2 = TerminalExpression(2)
        obj = DivideExpression(value2,value1)
        self.assertEqual(5,obj.interpret())

class SinFunctionClass(unittest.TestCase):
    def test_sinExpression(self):
        value1 = TerminalExpression(45)
        obj = SinFunction(value1)
        self.assertEqual(0.8509035245341184,obj.interpret())

class LogFunctionClass(unittest.TestCase):
    def test_logExpression(self):
        value1 = TerminalExpression(16)
        obj = LogFunction(value1)
        self.assertEqual(4,obj.interpret())

# Checks the isOperator of ParseUtil class wheather it returns the appropriate symbol or not.
class ParserUtilClass(unittest.TestCase):
    def test_isOperator(self):
```

```
        obj = ParserUtil()
        self.assertEqual(True,obj.isOperator('+'))
        self.assertEqual(True, obj.isOperator('-'))
        self.assertEqual(True, obj.isOperator('*'))
        self.assertEqual(True, obj.isOperator('/'))
        self.assertEqual(True,obj.isOperator('sin'))
        self.assertEqual(True, obj.isOperator('lg'))

    # Checks the getExpressionObject weather it returns appropriate object corresponding to symbol
or not.
    def test_getExpressionObject(self):
        obj = ParserUtil()
        self.assertEqual(True,isinstance(obj.getExpressionObject(3,4,'+'),AddExpression))
        self.assertEqual(True, isinstance(obj.getExpressionObject(3, 4, '-'), SubtractExpression))
        self.assertEqual(True, isinstance(obj.getExpressionObject(3, 4, '*'), MultiplyExpression))
        self.assertEqual(True, isinstance(obj.getExpressionObject(3, 4, '/'), DivideExpression))
        self.assertEqual(False, isinstance(obj.getExpressionObject(3,4,'-'),AddExpression))
        self.assertEqual(True, isinstance(obj.getExpressionObject(4,4,'sin'),SinFunction))
        self.assertEqual(True, isinstance(obj.getExpressionObject(4, 4, 'lg'), LogFunction))

# tests the evaluation of postfix Expression
class ExpressionParserClass(unittest.TestCase):
    def test_parser(self):
        obj = ExpressionParser()
        result = obj.parser('5 5 +')
        self.assertEqual(10,result.interpret())
        result = obj.parser('50 50 -')
        self.assertEqual(0,result.interpret())
        obj.list = [1,2,3]
        self.assertEqual([1,2,3],obj.list)

# checks the updatation of value from evaluation of formula
class CellClass(unittest.TestCase):
    def test_formuala_value(self):
        frame = tk.Tk()
        obj = Cell(1,1,None,None,None,frame)
        obj.formula = '4 5 * 10 2 / +'
        obj.update()
        self.assertEqual(25,obj.value)

# checks the dependency updates of particular cell
class SpreadSheetClass(unittest.TestCase):
    def test_dependency(self):
        frame = tk.Tk()
        obj = SpreadSheet(1,9,frame)
        obj.create_widgets()
        obj.cells['A'].formula = '5 5 +'
        obj.cells['A'].update()
        obj.cells['B'].formula = '$A'
        obj.cells['B'].update()
        self.assertEqual(obj.cells['B'].value,obj.cells['A'].value)

# checks the saving of snapshot with help of CareTaker
class CareTakerClass(unittest.TestCase):
    def test_save(self):
        frame = tk.Tk()
        obj = SpreadSheet(1,9,frame)
        caretakerobj = CareTaker()
        obj.cells['A'].formula = '100'
        obj.cells['A'].update()
        caretakerobj.save(obj.cells['A'])
        self.assertEqual(obj.cells['A'],caretakerobj.mementoList[0])

    # Checks the undo operation from careTaker and restores its previous state.
    def test_undo(self):
        frame = tk.Tk()
        obj = SpreadSheet(1,9,frame)
        caretakerobj = CareTaker()
        cellA = obj.cells['A']
        cellA.formula = '5 5 +'
        cellA.update()
        caretakerobj.save(cellA)
        cellA.formula = '100 100 +'
        cellA.update()
        caretakerobj.save(cellA)
```

```
        self.assertEqual(cellA.value,200)
        caretakerobj.undo()
        self.assertEqual(cellA.value,10)

def main():
    root = tk.Tk()
    app = SpreadSheet(1,9,master=root)
    app.mainloop()

    unittest.main()

if __name__ == '__main__':
    main()
```