IMPERATIVE PROGRAMMING 3 TT2019

SHEET 2

GABRIEL MOISE

Question 1

```
class FilterIterator[T] (test : T => Boolean, it : Iterator[T]) extends Iterator[T] {
  private var _next : Option[T] = None
  advance
```

// Advances the iterator until we get to an element that satisfies test: if _next = None, there is no next element, otherwise _next = Some(element)

```
  private def advance () = {
    _next = None
    var found = false
    while (it.hasNext && ! found)
    {
      var current = it.next
      if (test(current)) {_next = Some(current) ; found = true}
    }
  }

  override def hasNext () : Boolean =
    {
      _next match {
        case None => false
        case _    => true
      }
    }

  override def next() : T =
    {
      _next match {
        case None => throw new NoSuchElementException("There is no next element")
        case Some(x) => {advance ; return x} // we only advance at a next instruction
      }
    }
}
```

Question 2

```
trait Command[T] {
  def execute(target: T): Option[Change]
}
trait Change {
  def undo()
}
// (a)
trait Account {
  private var money = 0

  // Abs : money is a non-negative integer that reflects the balance of the current account

  /** deposit a non-negative amount of money from the account*/
  // Pre : x >= 0
  // Post : money = money + x && return (x>=0)
  def deposit (x : Int) : Boolean

  /** withdraw a non-negative amount of money from the account*/
  // Pre : x > 0 && x <= money
  // Post : money = money - x && return (x>=0) && (money >= x)
  def withdraw (x : Int) : Boolean

  /** see the balance of the account */
  // Post : return money
  def balance : Int
}
// (b)
```

```
class DepositCommand (amount : Int) extends Command[Account] {
  override def execute (target : Account) : Option[Change] =
    {
      if (target.deposit(amount))
        Some (new Change {def undo = target.withdraw(amount)} )
      else None
    }
}

class WithdrawCommand (amount : Int) extends Command[Account] {
  override def execute (target : Account) : Option[Change] =
    {
      if (target.withdraw(amount))
        Some (new Change {def undo = target.deposit(amount)} )
      else None
    }
}
// (c)
class BasicAccount (_balance : Int) extends Account {
  private var money = _balance

  def deposit (x : Int) : Boolean =
    {
      if (x >= 0) {money += x ; return true}
      else return false
    }

  def withdraw (x : Int) : Boolean =
    {
      if ((x >= 0) && (money >= x)) {money -= x ; return true}
      else return false
    }

  def balance : Int = money
}
```

Question 3

```
trait PriorityQueue {
  def isEmpty: Boolean // Determine if queue is empty
  def insert(e:Int) // Place the element e in the queue
  def remove(e:Int) // Remove (one copy of) element e (if present)
                    // ...(this operation is needed to undo insert)
  def delMin(): Int // Remove and return the smallest element
}

class InsertCommand (e : Int) extends Command[PriorityQueue] {
  override def execute (target : PriorityQueue) : Option[Change] =
  {
    target.insert(e)
    Some(new Change {def undo = target.remove(e)})
  }
}

class DelMinCommand extends Command[PriorityQueue] {
  override def execute (target : PriorityQueue) : Option[Change] =
  {
    if (target.isEmpty) return None
    var min = target.delMin
    Some(new Change {def undo = target.insert(min)})
  }
}
```

Question 4

// (a)

```
class AndThenCommand[T] (first : Command[T], second : Command[T]) extends Command[T] {
  override def execute (target : T) : Option[Change] =
  {
    var ch1 = new Change {def undo = {}}
    first.execute(target) match {
      case None => None
      case (Some(ch)) => ch1 = new Change {def undo = ch.undo}
    }

    second.execute(target) match {
      case None => {ch1.undo ; None}
      case (Some (ch2)) => Some(new Change {def undo = {ch2.undo ; ch1.undo}})
    }
  }
}
```

// (b) and (c) are in the object Tester at the end

Question 5

// (a)

```
class WhileCommand[T](test:T => Boolean, cmd:Command[T]) extends Command[T] {
  override def execute (target : T) : Option[Change] = {
    var change = new Change {def undo = {}}
    if (! test(target)) return Some(change)
    cmd.execute(target) match {
      case None => return None
      case Some(ch) => change = new Change {def undo = ch.undo}
    }
    var nrUndo = 1 // how many times we need to apply cmd.undo to undo the whole thing
    while (test(target))
    {
      cmd.execute(target) match {
        case None => {for (i <- 0 until nrUndo) change.undo ; return None}
        case Some(ch) => nrUndo += 1
      }
    }
    Some(new Change {def undo = {for (i <- 0 until nrUndo) change.undo}})
  }
}
```

// (b) function defined at the end

Question 6

If we type "abc", move to the beginning of the line and then to the end, then we press space and then type "def", by pressing CTRL-Z the whole text will disappear. This happens because every time we type a new character, we use an insertion command. By looking in the insertCommand from UndoableEditor file, the lastChange is going to be the composition of the last change and this one (only if the last one was also of type AmalgInsertion, which in our case is), because this way the insertions act like a single command, when they are undone.

1) A less surprising implementation would treat each change as solitary and we will add a new change after each character insertion, like that:

```
override def insertCommand(ch: Char) = {
        var p = ed.point
        super.insertCommand(ch)
        lastChange = new Change {
                def undo() { ed.deleteChar(p) }
                def redo() { ed.insert(p,ch) }
        }
  }
```

2) But in many other editors, after we type abc, execute some commands and then Space + "def", when we undo, we want the " def" to be deleted and then "abc". Thus, a different version uses a variable last_key that remembers the last key pressed and does not use amalgamate if there was a command in between:

```
/** The controller class for a basic editor */
class Editor {
  ...
  /** Last key pressed */
  protected var last_key = -1 // can be accessed by UndoableEditor since it is protected
  ...
  /** Read keystrokes and execute commands */
  def commandLoop() {
    //activate(display)
    while (alive) {
      val key = display.getKey()
      Editor.keymap.find(key) match {
        case Some(cmd) => obey(cmd)
        case None => beep()
      }
      last_key = key // to update the last_key value
    }
    }
...
}
/** The controller class for an  editor with undoable commands */
class UndoableEditor extends Editor with UndoHistory {
    ...
    /** Command: Insert a character and record change */
    override def insertCommand(ch: Char) {
        super.insertCommand(ch)
        lastChange = new AmalgInsertion(ed.point-1, ch)
    }

    /** Record of insertion that can be amalgamated with adjacent, similar changes */
    class AmalgInsertion(val pos: Int, ch: Char) extends Change {
        /** The text inserted by all commands that have merged with this one */
        private val text = new Text(ch)

        def undo() { ed.deleteRange(pos, text.length) }
        def redo() { ed.insert(pos, text) }

        override def amalgamate(change: Change) : Boolean = {
                  if (! Display.printable.contains(last_key)) return false // other command was executed meanwhile (command that would have
impact on our implementation of undo), then we don't amalgamate
            change match {
                case other: AmalgInsertion =>
                  if (text.charAt(text.length-1) == '\n'
                        || other.pos != this.pos + this.text.length)
                     false
                  else {
                     text.insert(text.length, other.text)
                     true
                  }

                case _ => false
```

```
          }
        }
    }
    ...
}
```

Question 7

The test suite fails for negative values of x, because we get that sqrt(x*x) = sqrt((-x)*(-x)) = -x != x, so we do not get the desired result. We can solve this problem by replacing n with math.abs(n). Another problem that occurs is when n*n becomes too large for the Int type, so if we want to solve this problem, we put n into a value of type long and work with it instead:

```
import org.scalacheck._
import org.scalacheck.Prop._

object Question7 extends org.scalacheck.Properties("Question7") {

  property("is a root") =
      forAll { (n: Int) => {
        var aux : Long  = n
        (scala.math.sqrt(aux*aux) == math.abs(aux)) } }
}
```

Question 8

I checked a few more properties, but I could not find any bug in the Text class.

```
  property("insert somewhere") =
    forAll { (s1: String, s2: String, p: Int) =>
     val t = new Text(s1)
     val pos: Int = (if (p < 0) (s1.length / 2) else pos) % (s1.length + 1)
     t.insert(pos, s2)
     t.toString() == s1.take(pos) + s2 + s1.drop(pos)
  }


   property("delete range") =
     forAll {(s: String, pos:Int, cnt: Int) =>
       var start: Int = (if (pos < 0) 0 else pos) % math.max(1, s.length)
       var elems: Int = (if (cnt < 0) 0 else cnt) % math.max(1, s.length - beg)
```

```
      val t = new Text(s)
      t.deleteRange(start, elems)
      t.toString == s.take(start) + s.drop(start + elems)
  }
```

The Tester object that contains testing for most questions and the makeTransaction and threshold functions:

```
object Tester {
  def makeTransaction[T] (commands : List[Command[T]]) : Command[T] =
    new Command[T]
    {
      def execute (target : T) : Option[Change] =
      {
        if (commands.isEmpty) Some (new Change {def undo = {}})
        var list = commands
        var listChanges = List[Change]()
        while (! list.isEmpty)
        {
          var current = list.head
          current.execute(target) match {
            case None =>
            {
              while (! listChanges.isEmpty)
              {
                var currentChange = listChanges.head
                currentChange.undo
                listChanges = listChanges.tail
              }
              return None
            }
            case Some(ch) => listChanges = ch :: listChanges
          }
          list = list.tail
        }
        var change = new Change
        {
          def undo =
            {
              while (! listChanges.isEmpty)
              {
                var currentChange = listChanges.head
                currentChange.undo
                listChanges = listChanges.tail
              }
            }
        }
        return Some(change)
      }
    }

  def treshold (limit : Int) : (PriorityQueue => Boolean) = {
    (queue  => testQueue(queue,limit))
  }

  def testQueue (queue : PriorityQueue, limit : Int) : Boolean = {
    var min = queue.delMin
    if (min < limit) return true
    queue.insert(min)
    return false
  }

  def main(args : Array[String]) = {
    /** Tests for Question 1 */
    println("-----------------------")
    println("Printing for Question 1")
    println("-----------------------")
    // Test 1.1 - normal situation
    var string1 = "Gabriel"
    var vowel : (Char => Boolean) = (x => (x == 'a') || (x == 'e') || (x == 'i') || (x == 'o') || (x == 'u'))
  var fit1 = new FilterIterator[Char] (vowel,string1.iterator)

  print("1.1. The vowels in " + string1 + " are: ")

  while (fit1.hasNext) print(fit1.next + " ")

  println()

  // Test 1.2 - empty String

  var string2 = ""

  var gapString : (Char => Boolean) = (x => ('g' <= x) && (x <= 'p'))

  var fit2 = new FilterIterator[Char] (gapString,string2.iterator)

  print("1.2. Empty test should print nothing: ")

  while (fit2.hasNext) print(fit2.next + " ")

  println()

  // Test 1.3 - elements that verify the test at the beginning and at the end

  var string3 = "abcda"

  var letter_a : (Char => Boolean) = (x => x == 'a')

  var fit3 = new FilterIterator[Char] (letter_a,string3.iterator)

  print("1.3. There should be two a's here: ")

  while (fit3.hasNext) print(fit3.next + " ")

  println()

  /** Tests for Question 2 */

  println("-----------------------")

  println("Printing for Question 2")

  println("-----------------------")

  // Test 2.1 - test from the task

  println("---------------")

  println("Test 2.1 - task")

  println("---------------")

  val ac1 = new BasicAccount(50)

  val d10 = new DepositCommand(10)

  val w5 = new WithdrawCommand(5)

  d10.execute(ac1)

  println("Balance is: "+ac1.balance + " (60)") // Should print 60

  w5.execute(ac1)

  println("Balance is: "+ac1.balance + " (55)") // Should print 55

  // Test 2.2 - test with overdrawing the balance
```

```
println("-------------------")

println("Test 2.2 - overdraw")

println("-------------------")

val ac2 = new BasicAccount(100)

val d100 = new DepositCommand(100)

val w50 = new WithdrawCommand(40)

w50.execute(ac2)

println("Balance is: "+ac2.balance + " (60)") // Should print 60

w50.execute(ac2)

println("Balance is: "+ac2.balance + " (20)") // Should print 20

w50.execute(ac2)

println("Balance is: "+ac2.balance + " (20)") // Should print 20 (overdraw)

// Test 2.3 - with undoing

println("------------------")

println("Test 2.3 - undoing")

println("------------------")

val ac3 = new BasicAccount(100)

val d60 = new DepositCommand(60)

val w30 = new WithdrawCommand(30)

println("Balance is: "+ac3.balance + " (100)") // Should print 100

d60.execute(ac3) match {

  case Some(ch) => ch.undo

  case _ => {}

}

println("Balance is: "+ac3.balance + " (100)") // Should print 100 (Deposits 60 and then uses undo)

w30.execute(ac3)

println("Balance is: "+ac3.balance + " (70)") // Should print 70

var change = w30.execute(ac3)

println("Balance is: "+ac3.balance + " (40)") // Should print 40

change match {

  case Some(ch) => ch.undo

  case _ => {}

}

println("Balance is: "+ac3.balance + " (70)") // Should print 70
```

```
// Test 2.4 - with illegal commands (negative amounts of money)

println("-----------------------------")

println("Test 2.4 - negative operations")

println("------------------------------")

val ac4 = new BasicAccount(100)

println("Balance is: "+ac4.balance + " (100)") // Should print 100

val invalidDeposit = new DepositCommand(-20)

val invalidWithdraw = new WithdrawCommand(-100)

invalidDeposit.execute(ac4)

println("Balance is: "+ac4.balance + " (100)") // Should print 100

invalidWithdraw.execute(ac4)

println("Balance is: "+ac4.balance + " (100)") // Should print 100

/** Tests for Question 4 */

println("-----------------------")

println("Printing for Question 4")

println("-----------------------")

// Test 4.1 - test from the task

println("---------------")

println("Test 4.1 - task")

println("---------------")

val ac5 = new BasicAccount(50)

val t1 = makeTransaction(List(d10,d10,w5,d10,w5))

val c1 = t1.execute(ac5)

println("Balance is: "+ac5.balance + " (70)") // Should print 70

c1.get.undo()

println("Balance is: "+ac5.balance + " (50)") // Should print 50

// Test 4.2 - test with invalid command

println("--------------------------")

println("Test 4.2 - invalid command")

println("--------------------------")

val ac6 = new BasicAccount(30)

val w60 = new WithdrawCommand(60)

val t2 = makeTransaction(List(d10,d10,w5,d10,w60,w5))

val c2 = t2.execute(ac6)
```

```
    assert (c2 == None)
    println("Balance is:"+ac6.balance + " (30)") // Should print 30 - nothing is done
    /** Tests for Question 4 */
    println("-----------------------")
    println("Printing for Question 5")
    println("-----------------------")
    // Test 5.1 - test that ends correctly + can undo its effect
    println("-------------------------")
    println("Test 5.1 - correct + undo")
    println("-------------------------")
    val ac7 = new BasicAccount(100)
    val w10 = new WithdrawCommand(10)
    val test : (Account => Boolean) = (x => x.balance >= 30)
    val tw10 = new WhileCommand(test,w10)
    val c3 = tw10.execute(ac7)
    println("Balance is: "+ac7.balance + " (20)") // Should print 20
    c3.get.undo()
    println("Balance is: "+ac7.balance + " (100)") // Should print 100
    // Test 5.2 - test that ends incorrectly because of invalid operation
    println("--------------------")
    println("Test 5.2 - incorrect")
    println("--------------------")
    val ac8 = new BasicAccount(100)
    val tw60 = new WhileCommand(test,w60)
    assert (tw60.execute(ac8) == None)
    println("Balance is: "+ac8.balance + " (100)") // Should print 100 - nothing is done
  }
}
```

The result it prints (in brackets there is the correct result):

```
 -----------------------
Printing for Question 1
-----------------------
1.1. The vowels in Gabriel are: a i e
1.2. Empty test should print nothing:
```

```
1.3. There should be two a's here: a a

-----------------------

Printing for Question 2

-----------------------

---------------

Test 2.1 - task

---------------

Balance is: 60 (60)

Balance is: 55 (55)

-------------------

Test 2.2 - overdraw

-------------------

Balance is: 60 (60)

Balance is: 20 (20)

Balance is: 20 (20)

------------------

Test 2.3 - undoing

------------------

Balance is: 100 (100)

Balance is: 100 (100)

Balance is: 70 (70)

Balance is: 40 (40)

Balance is: 70 (70)

-----------------------------

Test 2.4 - negative operations

------------------------------

Balance is: 100 (100)

Balance is: 100 (100)

Balance is: 100 (100)

-----------------------

Printing for Question 4

-----------------------
```

```
---------------
Test 4.1 - task
---------------
Balance is: 70 (70)
Balance is: 50 (50)
--------------------------
Test 4.2 - invalid command
--------------------------
Balance is:30 (30)
-----------------------
Printing for Question 5
-----------------------
-------------------------
Test 5.1 - correct + undo
-------------------------
Balance is: 20 (20)
Balance is: 100 (100)
--------------------
Test 5.2 - incorrect
--------------------
Balance is: 100 (100)
```