Tuan Nguen

**Question 1:**

class FilterIterator[T] (test: T => Boolean, var it: Iterator[T]) extends Iterator[T]{

it = it.filter(test) // Returns an iterator for elements that satisfy "test"

override def hasNext: Boolean = it.hasNext

override def next: T = it.next

}

object Test{

def test(x: Int): Boolean = x >= 9

def main(args: Array[String]) = {

val data = new Array[Int](10)

for(i <- 0 to 9) data(i) = i

val it = data.iterator

val filterIter = new FilterIterator[Int](test, it)

while(filterIter.hasNext) println(filterIter.next)

}

}

**Question 2:**

trait Command[T]{

def execute(target: T): Option[Change]

}

trait Change{

def undo()

}

trait Account{

/\*\* DTI: \_balance >= 0 \*/

/\*\* Pre: x >= 0

\* Post: \_balance = \_balance0 + x \*/

def deposit(x: Int)

/\*\* Pre: x >= 0

\* Post: if(\_balance0 >= x) \_balance = \_balance0 - x \*/

def withdraw(x: Int)

/\*\* Post: Return \_balance \*/

def balance(): Int

}

class DepositCommand(amount: Int) extends Command[Account]{

def execute(acc: Account): Option[Change] = {

if(amount > 0){

acc.deposit(amount)

Some(new Change{ def undo() {acc.withdraw(amount)} })

}

None // If the deposit is not executed

}

}

class WithdrawCommand(amount: Int) extends Command[Account]{

def execute(acc: Account): Option[Change] = {

if(acc.balance >= amount && amount >= 0){

acc.withdraw(amount)

Some(new Change{ def undo() {acc.deposit(amount)} })

}

None // If the withdrawaw is not executed

}

}

class BasicAccount(private var \_balance: Int) extends Account{

def deposit(amount: Int) = {

assert(amount >= 0)

\_balance += amount

}

def withdraw(amount: Int) = {

assert(amount >= 0 && \_balance >= amount)

\_balance -= amount

}

def balance(): Int = \_balance

}

object Test{

def main(args: Array[String]) = {

val ac1 = new BasicAccount(50)

val d10 = new DepositCommand(10)

val w5 = new WithdrawCommand(5)

d10.execute(ac1)

println("Balance is:" + ac1.balance)

w5.execute(ac1)

println("Balance is:" + ac1.balance)

}

}

**Question 3:**

trait PriorityQueue {

/\*\* Determine if the queue is empty \*/

def isEmpty: Boolean

/\*\* Place the element e in the queue \*/

def insert(e: Int)

/\*\* Remove a copy of element e, if present \*/

def remove(e: Int)

/\*\* Remove and return the smallest element \*/

def delMin(): Int

}

class InsertCommand(x: Int) extends Command[PriorityQueue] {

def execute(queue: PriorityQueue): Option[Change] = {

queue.insert(x)

Some(new Change{ def undo() {queue.remove(x)} })

}

}

class DelMinCommand(x: Int) extends Command[PriorityQueue] {

def execute(queue: PriorityQueue): Option[Change] = {

if(queue.isEmpty) None

else {

val min = queue.delMin()

Some(new Change{ def undo(){queue.insert(min)} })

}

}

}

**Question 4:**

a)

class AndThenCommand[T](first: Command[T], second: Command[T]) extends Command[T] {

def execute(target: T): Option[Change] = {

var change1: Change = first.execute(target) match{

case None => return None

case Some(change) => change

}

var change2: Change = second.execute(target) match{

case None => {change1.undo(); return None}

case Some(change) => change

}

return Some(new Change{

def undo(){

change1.undo()

change2.undo()

}

})

}

}

b)

object Q4{

def makeTransaction[T](commands : List[Command[T]]) : Command[T] = {

return new Command[T]{

def execute(target: T): Option[Change] = {

var pass = true // A boolean to keep track if there is a command that fails(true if none fail)

var changes = List[Change]() //Keeps track of the changes we've made

for (x <- commands){

x.execute(target) match{

case Some(change) => changes = change :: changes

case None => pass = false //If a command fails, the whole transaction fails

}

}

var undoAll = new Change{ //Creates an undo list of all changes

def undo(){

for (x <- changes) x.undo()

}

}

if (!pass){

undoAll.undo()

return None

}

return Some(undoAll)

}

}

}

}

c)

def main(args: Array[String]) = {

val ac1 = new BasicAccount(50)

val d10 = new DepositCommand(10)

val w5 = new WithdrawCommand(5)

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

val q = new AndThenCommand(w5, w5)

val c1 = t.execute(ac1)

println(“Balance is:” + ac1.balance)

c1.get.undo()

println(“Balance is:” + ac1.balance)

}

**Question 5:**

a)

class WhileCommand[T](test: T => Boolean, cmd: Command[T]) extends Command[T] {

def execute(target: T): Option[Change] = {

var pass = true

var changes = List[Change]()

while(test(target) && pass){

cmd.execute(target) match{

case Some(change) => changes = change :: changes

case None => pass = false

}

}

var undoAll = new Change{

def undo(){

for(x <- changes) x.undo()

}

}

/\*\* If a command failed \*/

if(!pass){

undoAll.undo()

return None

}

return Some(undoAll)

}

}

b)

def threshold(limit: Int): (PriorityQueue => Boolean) = (target: PriorityQueue) => {

val min = target.delMin()

target.insert(min)

min < x

}

**Question 6:**

After pressing Ctrl-Z the editor deletes everything. This happens because the editor amalgamates the typed characters and since there is no other undoable command other than insertion it amalgamates all the text we’ve typed, so it appears as one change in the history.

A more natural behaviour is to delete just “def” when we press Ctrl-Z, which can be done if we end the amalgamation once space is pressed. This can simply be fixed if we change the

**if** (text.charAt(text.length-1) == '\n'

|| other.pos != **this**.pos + **this**.text.length)

to

**if** (text.charAt(text.length-1) == '\n'

|| text.charAt(text.length-1) == ' '

|| other.pos != **this**.pos + **this**.text.length)

**Question 7:**

/\*

! Sqrt.is a root: Falsified after 0 passed tests.

> ARG\_0: -1

> ARG\_0\_ORIGINAL: 216378256

Found 1 failing properties.

\*/

The error occurs because of the integer limit. As for finding a way to correct the error we can bound n to be between 0 and sqrt(2^31-1).

**Question 8:**

**object** Q\_Test **extends** org.scalacheck.Properties("Test"){

property("insert at start") =

forAll { (s: String) =>

**val** t = **new** Text(); t.insert(0, s)

t.toString() == s }

property("insert at end") =

forAll { (s1: String, s2: String) =>

**val** t = **new** Text(s1); t.insert(t.length, s2)

t.toString() == s1 + s2}

property("insert single char at start") =

forAll {(s: String, c: Char) =>

**val** t = **new** Text(s); t.insert(0, c)

t.toString() == c + s}

property("insert single char at end") =

forAll {(s: String, c: Char) =>

**val** t = **new** Text(s); t.insert(t.length, c)

t.toString() == s + c}

property("clear all") =

forAll {(s: String) =>

**val** t = **new** Text(s); t.clear()

t.toString() == ""}

property("delete single char at end") =

forAll {(s: String) =>

**val** t = **new** Text(s);

**if** (s.length == 0)return **true**

**else**{

t.deleteLast()

t.toString == s.substring(0, s.length - 1)

}

}

}

/\*

+ Test.insert at start: OK, passed 100 tests.

+ Test.insert at end: OK, passed 100 tests.

+ Test.insert single char at start: OK, passed 100 tests.

+ Test.insert single char at end: OK, passed 100 tests.

+ Test.clear all: OK, passed 100 tests.

+ Test.delete single char at end: OK, passed 100 tests. \*/