Imperative Programming Part 3
Problem Sheet 2

Peter Jeavons[*]

Trinity Term 2019

1. **[Programming]** Write a "filter iterator" wrapper class which provides an iterator over all the elements of a given iterator that pass a given test. All other members returned by the original iterator should be ignored. You may use the following model as a guide to how the class should look.

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

   In order to verify that your new class is doing the correct thing you should test it with a suitable set of examples. (You may find it helpful to note that in Scala, for any `String`, `s`, the expression `s.iterator` returns an iterator for the characters in `s`.)

2. **[Programming]** The `Command` trait defined below represents undoable commands that act on objects of type `T`. A command may fail and return `None`, or it may succeed and return `Some(ch)`, where `ch` is an object implementing the trait `Change`, which provides a method by which the effect of the command may be undone. In this question, you are asked to define classes that implement several forms of `Command`, together with appropriate `Change` classes that allow them to be undone.

```
trait Command[T] {
   def execute(target: T): Option[Change]
}
trait Change {
   def undo()
}
```

   (a) Write a trait `Account` to describe an abstract data type of simplified bank account; it should provide methods to deposit a positive amount of money, and withdraw a positive amount of money, and to discover the balance in the account.

   You may assume for simplicity that amounts of money are represented by integer values. You may also assume that depositing an equal amount suffices to undo the effect of an earlier withdrawal, and vice versa. The balance of the account must never be allowed to become negative.

   Include suitable comments in the `Account` trait so that it fully specifies an abstract data type (and pins down any missing or ambiguous elements in the above description).

   (b) Define appropriate sub-classes of `Command[Account]` so that for any positive integer `amount`, `new DepositCommand(amount)` creates a `Command` object that can be used to deposit `amount`

[*]Based on earlier material by Mike Spivey, Joe Pitt-Francis and Milos Nikolic.

in a specified account, and `new WithdrawCommand(amount)` creates a `Command` object that can be used to withdraw `amount` from a specified account, failing if doing so would cause the account to become overdrawn.

(c) Write a simple implementation of the `Account` trait in a class `BasicAccount` with a class parameter specifying the initial balance.
Run some suitable tests on your answers above. For example:

```
val ac1 = new BasicAccount(50)
val d10 = new DepositCommand(10)
val w5 = new WithdrawCommand(5)

d10.execute(ac1)
println("Balance is:"+ac1.balance) // Should print 60
w5.execute(ac1)
println("Balance is:"+ac1.balance) // Should print 55
```

3. The following trait defines an unusual type of priority queue over integers

```
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
}
```

With the same conventions as in Question 2, define command classes so that `InsertCommand(x)` represents a `Command[PriorityQueue]` object that inserts the integer `x` into a specified priority queue, and `DelMinCommand` represents a `Command[PriorityQueue]` object that removes (but does not return) the smallest element from the queue.

4. **[Programming]** In this question you will build general tools for combining the `Command` objects used in Questions 2 and 3.

(a) Define a sub-class of `Command[T]` called `AndThenCommand[T]` so that, for any objects `first` and `second` of type `Command[T]`, `AndThenCommand(first, second)` represents a command that executes the first command on a specified target object, and if that succeeds then executes the second command. If either command fails, then the target object should be left in the state in which it started.

(b) Hence or otherwise, define a *factory method* `makeTransaction` so that if `commands` is a list of type `List[Command[T]]`, then `makeTransaction[T](commands)` returns a command that executes all the commands in sequence on a specified target object, succeeding exactly if they all succeed, and otherwise leaving the target in the state in which it began.

(c) Run some suitable tests on your answers above. For example, using the `BasicAccount` class defined in Question 2:

```
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 c1 = t.execute(ac1)
println("Balance is:"+ac1.balance) // Should print 70

c1.get.undo()
println("Balance is:"+ac1.balance) // Should print 50
```

5. In this question you will build another general tool for manipulating the `Command` objects used in Questions 2 and 3.

   (a) Define a class `WhileCommand[T]` with heading

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

   The effect of executing this command should be to execute the command `cmd` repeatedly, stopping when the `test` applied to the target of the command no longer returns `true`. If any attempted execution of `cmd` fails, the whole `WhileCommand` should fail and leave the target state unchanged. Otherwise, the `WhileCommand` succeeds and returns a `Change` that is able to undo its entire effect.

   (b) Define a function `threshold` so that

   ```
   new WhileCommand[PriorityQueue](threshold(limit), new DelMinCommand())
   ```

   creates a `Command` object that removes all elements of a specified priority queue that are smaller than `limit`.

6. In the *Ewoks* editor try opening a blank file, typing `abc`, then pressing `Ctrl-A` to move to the start of the line and `Ctrl-E` to move to the end again. Now type a space and then `def`, and press `Ctrl-Z`.

   What happens? Explain how the editor implements this behaviour, suggest an alternative that may be less surprising to the user, and implement it.

7. Run the following test program and report the outcome. If it fails, explain why, and find some suitable way to correct the error.

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

object Q_Sqrt extends org.scalacheck.Properties("Sqrt") {

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

}
```

8. Extend the following ScalaCheck test suite for the `Text` class in the Editor so that it checks some additional properties. For example, check that an insertion in the middle of the text has the desired results. Can you find any errors in the `Text` class?

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

object Q_Text extends org.scalacheck.Properties("Text") {

   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 }
}
```