IMPERATIVE PROGRAMMING 3 TT2019

SHEET 3

GABRIEL MOISE

Question 1

```
/** The Dictionary class - Dictionary.scala*/
class Dictionary(fname: String) {
  private val words = new scala.collection.mutable.HashSet[String]
  initDict

  /** Check if a word is in the dictionary */
  def isWord(w: String) = words.contains(w)

  /** Initialize the dictionary */
  private def initDict = {
    val allWords = scala.io.Source.fromFile(fname).getLines
    def include(w: String) = w.forall(_.isLower)
    for (w <- allWords; if include(w)) words += w
  }
}

/** WordPathsApp.scala */
import scala.swing._
import scala.swing.event._

object WordPathsApp extends SimpleSwingApplication {
  def top = new MainFrame {
    object SourceText extends TextField {columns = 10}
    object TargetText extends TextField {columns = 10}
    object SearchButton extends Button {text = "Search"}
    var solutionLabel = new Label {text = "No solution"}
    contents = new FlowPanel {
      contents += new Label{text = "Start with: "}
      contents += SourceText
```

```
    contents += new Label{text = " End with: "}
    contents += TargetText
    contents += SearchButton
    contents += solutionLabel
  }

  /** the class that finds the path */
  val finder = new PathFinder(new Dictionary("knuth_words"))

  /** a thread that finds the specified path */
  class Worker extends Thread {
    override def run {
      val answer = finder.find_path(SourceText.text, TargetText.text)
      println(answer)
      val message = if (answer == Nil) "No solution" else
      answer.toString
      Swing.onEDT {solutionLabel.text = message}
    }
  }
  listenTo(SearchButton)
  reactions += {
    case ButtonClicked(SearchButton) => (new Worker).start
  }
 }
}

/** PathFinder.scala */
/** A class that finds paths in a dictionary */
class PathFinder(dict: Dictionary) {
 private val path = new scala.collection.mutable.HashMap[String,List[String]]
 private val queue = new scala.collection.mutable.Queue[String]

 /** Find a path from source to target */
 def find_path(source: String, target: String): List[String] = {
```

```
  if (!dict.isWord(source) || !dict.isWord(target) || source.length != target.length)
    return Nil
  path.clear
  queue.clear
  path.update(source, List(source))
  queue.enqueue(source)
  while (!queue.isEmpty && !path.contains(target)) {
    val word = queue.dequeue
    val len = word.length
    for (i <- 0 until len; c <- 'a' to 'z') {
      if (word.charAt(i) != c) {
        val new_word = word.take(i) + c + word.drop(i + 1)
        if (!path.contains(new_word) && dict.isWord(new_word)) {
          path.update(new_word, new_word :: path(word))
          queue.enqueue(new_word)
        }
      }
    }
  }
  if (path.contains(target)) path(target).reverse
  else Nil
 }
}
```

Question 2

```
class MySet[T] (elements : Set[T]) extends Set[T] {
 // I use a pre-defined List as my concrete implementation
 private val mySet : List[T] = elements.toList
 def contains (key : T) : Boolean = {
  for (i <- this ; if (key == i)) return true
  false
 }
 def iterator : Iterator[T] = mySet.iterator
 def + (elem : T) : MySet[T] = new MySet (mySet.toSet + elem)
 def - (elem : T) : MySet[T] = new MySet (mySet.toSet - elem)
```

```
  override def empty : MySet[T] = new MySet(Set())
}
```

Question 3

(a)

```
trait PartialOrder[T] {
  def <=(that: T): Boolean // checks this <= that. Partial order on T.
  def lub(that: T): T // returns the least upper bound of this and that.
}


class MySet[T] (elements : Set[T]) extends Set[T] with PartialOrder[MySet[T]]{
  ...
  // We can use for (v <- this/that) because we already created the method iterator
  def <= (that : MySet[T]) : Boolean = {
    for (v <- this) if (! that.contains(v)) return false
    true
  }
  def lub (that : MySet[T]) : MySet[T] = {
    var result = this
    for (v <- that) result = result + v
    result
  }
}
```

(b)

```
class UpSet [T <: PartialOrder[T]](_elements : Set[T]) {
  /** We create the set using a "minimal" standard set */
  private var elements = minimal(_elements)

  /** Check if the value is in the set */
  def contains (x : T) : Boolean = {
    for (v <- elements ; if (v <= x)) return true
    false
  }

  /** Intersect two upsets */
```

```
  def intersection (that : UpSet[T]) : UpSet[T] = {
    var result = Set[T] ()
    for (a <- elements ; b <- that.elements) result = result + (a lub b)
    new UpSet(result)
  }

  /** Private function that finds the minimal subset */
  private def minimal (data : Set[T]) : Set[T] = {
    var result : Set[T] = data
    for (a <- data) {
      var ok = true
      for (b <- data ; if ((a != b) && (b <= a))) ok = false
      if (! ok) result = result - a
    }
    result
  }
}
```

(c)

```
class UpSet [T <: PartialOrder[T]](_elements : Set[T]) extends PartialOrder[UpSet[T]]{
...
  /** Compare this with that */
  def <= (that : UpSet[T]) : Boolean = {
    for (v <- elements ; if (! that.contains(v))) return false
    true
  }
  /** Find the lub of two sets (reunion) */
  def lub (that : UpSet[T]) : UpSet[T] = {
    var result = elements
    for (v <- that.elements) result = result + v
    new UpSet(result) // automatically finds the minimal subset
  }
}
```

Question 4

```
// It can be observed from the specification that the class parameter of Bag is a function
class Bag[T](private val f : (T => Int)) {
  /** Add an element to the bag */
  def add (x : T) : Bag[T] = new Bag[T](v => {if (x == v) f(v) + 1 else f(v)})

  /** Remove an element if it is in the bag */
  def remove (x : T) : Bag[T] = new Bag[T] (v => {if (x == v) Math.max(0,f(v)-1) else f(v)})

  /** Count how many times an element appears in the bag */
  def count (x : T) : Int = f(x)

  /** Return the union of two bags */
  def union (that : Bag[T]) : Bag[T] = new Bag(v => f(v)+ that.count(v))
}
```

The Bag is represented by a function and functions should be contravariants in the domain (if A <: B then T[B] <: T[A]), i.e. we can use g : A -> C instead of f : B -> C as long as A contains B. Therefore, the bag should be contravariant. This change can be achieved by defining the generic type T as "-T"

```
class Bag [-T] (private val f : (T => Int)) {…}
```

Now, everything works as before, except the union function. Note that if we have f : B -> C and g : A -> C with A containing B, the union (f U g) should be defined on B -> C. Thus, the correct definition of union is:

```
def union [X <: T] (that : Bag[X]) : Bag[X] = new Bag[X] (v => f(v) + that.count(v))
```

Question 5

As it can be seen in the lecture notes, mutable objects/classes should not be covariant nor contravariant because of the exceptions that might occur in different situations. Suppose that A <: B and Set[A] <: Set[B] for mutable sets.

```
var a : Set[A] = new Set[A]
var b : Set[B] = new Set[B]
b = a // covariance allows this operation
b += new B // exception because b can now be added only objects of type A or its subtypes
```

Now, the contravariant situation: A <: B and Set[B] <: Set[A]

```
var a : Set[A] = new Set[A]
var b : Set[B] = new Set[B]
a = b // contravariance allows this operation
a += new B // works even though it shouldn't work, because B is not a subtype of A
```

However, this may provide more flexibility if it is handled more carefully. For general purposes, it is safer to treat mutable classes as invariants.

Question 6

A full set of documentation for the AutoSnail case study can be generated by running the command

scaladoc <path to sources>

In my case, I created a new folder “docs” in the same place where the “src” folder is and used:

scaladoc …/src/*

A lot of html files were created, including index.html which is the main entry to the documentation. In the AppFrame class I found the following methods inherited from scala.swing.UIElement :

def background : Color

def background_ = (c : Color) : Unit

def toolkit : ToolKit

and many others

Question 7

The Façade pattern is mainly used to simplify a set of complex classes by an easy-to-use interface. For example, the AppFrame implements the Façade pattern to ease the access to the GUI. It also achieves loose coupling between all the components. The responsibilities of the AppFrame is to aggregate all GUI elements into one window, also managing the events that happen and their order. Representation: Users <-> Façade class <-> Classes

Question 8

First of all, we have to create a new button that can be used to change the color of the sea:

```
private val ChangeColorButton = new Button {text = “Change sea color”}
```

and to create a reactor to this button in the AppFrame

```
listenTo (ChangeColorButton)
  reactions += {
        case ButtonClicked (ChangeColorButton) => {
              val default_color = viewer.background
              val option_color = ColorChooser.showDialog(this, “Choose see color”, default_color)
              option_color match {
                    case None => println(“No color chosen!”)
                    case Some(new_color) => Swing.onEDT {viewer.background = new_color ; viewer.repaint}
              }
        }
}
```

We also need to add the button to the design:

```
private val controls = new GridBagPanel {
    val elements = Array[Component](new MyLabel("From:"), fromText,
      …
      ChangeColorButton)
    …
  }
```