Functional Programming with Scala

Homework 1

Instructor: Dr. Ionut Cardei

The purpose of this programming assignment is to:
1. Chapter 1: test understanding of referential transparency, pureness, and the substitution model
2. Chapter 2: practice with basic FP concepts in Scala, such as pure functions, importing classes, higher and polymorphic functions, tail recursion, lambda expressions, traits, objects, and classes.

Make sure you follow exactly all requirements listed in this document to get full credit.

Academic Honesty

For every assignment, including this, you have to work alone. Your submission must be your original work, that was not shared with anybody else. It is not enough to rename identifiers and move code around. Your solution for non-trivial answers (e.g. one liners) must be substantially different from any other.
Note that it is easy for the grader to find code or answers taken from online sources or to find solutions that are too similar.
Also, do not copy text (like examples, definitions) from notes/textbook or other sources, unless explicitly required to do so. Use your own words; paraphrase.
When in doubt, please contact the instructor at icardei@fau.edu.

Coding Requirements

For this assignment we use Intellij IDEA to write the code. Emacs also accepted.

Don't forget to read the **Important Notes** at the end of this file.

Homework 1 has 5 problems.
Total score: 100 points + extra credit.

Homework Formatting and Delivery

1. Start a new Word document called h1.docx. At the top write a title and your name.
2. Write the text answers for the problems in the given order (e.g. 1, 2, 3,...). Include Scala code from your source files where applicable. Do not reorder solutions in the document.
3. Although syntax highlighting is not required, it is very appreciated by the grader, as it makes your code more readable, hence more likely to prevent grading mistakes. Paste directly from IDEA or check out http://hilite.me.
4. Make sure the code build and runs correctly.
5. Convert the Word file to PDF.
6. Upload the PDF file with the answers and the Scala files on Canvas.

Problem 1. FP Concepts

a) Referential transparency.
a.1 Explain informally and **with your own words** what referential transparency means.
It is silly to say this, but do not copy/paste the definition from the book/note/other sources.
a.2 Give an example of a non-trivial expression that is referential transparent. Explain why it is so using the formal textbook definition. (Just to be sure, do not use the examples from the textbook/notes. Not going to repeat this again...)
a.3 Give an example of a non-trivial expression that is **not** referential transparent. Explain why it is so using the formal textbook definition.

b) Function purity.
b.1 Explain informally and **with your own words** what a pure function is.
b.2 Give an non-trivial example of a pure function written in Scala. Explain why it is so using the formal textbook definition.
b.3 Give an example of a non-trivial **impure** function written in Scala. Explain why it is so using the formal textbook definition.

c) Explain why writing pure functions helps us get these advantages:
c.1 composable code and modularity
c.2 simplified testing
(mention the implications of separation of concerns, made possible by function purity)

Problem 2. The substitution model

a)
a.1 Explain with your own words how the substitution model is used for equational reasoning.
a.2 Consider this short program:

```
object p1 {

 // format a multiline string with product UPC and name
 def addProduct(upc: Int, prodName: String, prevProds: String) : String = {
   val line = "UPC:%04d Product:%20s".format(upc, prodName)
   prevProds + "\n" + line
 }

 def main(args: Array[String]): Unit = {
   val prod0 = "Apple 2"
   val prod1 = "IBM PC"
   val productLines0 = addProduct(325, prod0, "")
   val productLines1 = addProduct(103, prod1, productLines0)

   // non FP code to display the result:
   println("Products: \n" + productLines1 + "\n")
 }
```

```
}
```

Use the substitution model to derive the value for variable productLines1.
a.3 Explain why function addProduct is a pure function.

b) Consider this code:

```
import java.time.LocalDate    // a date class in the local timezone

// University Registrar class.
class Registrar(val univName: String) {
  private var allStudents = List[Student]()   // a MUTABLE instance variable

  // add student to the list of students registered
  // Return the number of students already registered
  def register(s: Student) : Int = {
    allStudents = allStudents :+ s
    // operator :+ appends something at the end of a list and returns the new list
    allStudents.size
  }

  def hasRegistered(s: Student) : Boolean = {
    @annotation.tailrec
    def go(lst: List[Student]) : Boolean = {
      if (lst == Nil) false
      else {
        if (lst.head.name == s.name) true
        else go(lst.tail)
      }
    }

    go(this.allStudents)
  }
}

// A student class.
class Student(val name:String) {
  // putting 'val' in front of parameter makes name an immutable instance variable
  val startedAt : LocalDate = LocalDate.now()
}



object p2 {
  def main(args: Array[String]): Unit = {
    val univ = new Registrar("FAU")
    val alice = new Student("Alice")
    val bob = new Student("Bob")



    val count1 = univ.register(alice)
```

```
  val count2 = univ.register(bob)

  val didBobRegister = univ.hasRegistered(bob)

  println("%d students registered".format(count2))
  println("Did Bob register? %b.\n\n".format(didBobRegister))
 }
}
```

b.1 Is expression `univ.register(bob)` referential transparent ? Explain your answer.
b.2 Is expression `univ.hasRegistered(bob)` referential transparent ? Explain your answer.

b.3 for 3 points extra credit extra credit extra credit extra credit extra credit extra credit
Sketch a solution for the student registration part that is pure, i.e. preserves referential transparency. It does not have to be a complete, working program. Use the methodology explained in the textbook/lecture. No need to write Scala code in a separate file.

Problem 3. Recursion and HOFs

Write the code for this problem in a file called p3.scala, within an object called *p3*.
a) Consider this function written in a non-functional, imperative style:

```
// imperative, impure version:  don't do like this
def trueForAll_imperative(bools: List[Boolean]) : Boolean = {
  var lst = bools
  while (lst != Nil) {
    if (! lst.head) {
      return false          // CAUTION: non-functional, bad programming style to use return
    }
    else {
      lst = lst.tail
    }
  }
  true
}
```

Write a **tail recursive pure function** called *trueForAll* that returns the same values as *trueForAll_imperative*. Use the correct annotation.

b) Write a tail-recursive polymorphic function with this signature:

```
  def countWithProperty[T](xs: List[T], p: T => Boolean) : Int
```

that returns the number of elements *x* from list *xs* for which *p(x)* is true.

For example:

```
  val luckyNumbers = List(4, 8, 15, 16, 23, 42)
  val evenCount = countWithProperty(luckyNumbers, (x: Int) => x % 2 == 0)
```

The value of variable *evenCount* is 4.

c) Write a tail-recursive polymorphic function with this signature:

```
def scalarProduct(x: Array[Double], y: Array[Double]) : Double
```

that returns the scalar product of arrays x and y.

For example:

```
val xvec = Array(-1.0, -2, 3)
val yvec = Array(2.0, -3, 1)
val prod  = scalarProduct(xvec, yvec)     // prod == 7
```

d) Write a tail-recursive polymorphic function with this signature:

```
def countCommonElements[T](xs: Array[T], ys: Array[T], p: (T, T) => Boolean): Int
```

that returns the number of elements *x* in *xs* and *y* in *ys*, such that p(x, y) is true.
This can be used, for example, to count the number of elements in *xs* that are also in *ys*:

```
val as = Array(7, 3, 8, 4, 9, 3, 5)
val bs = Array(8, -4, 5, 1, 0, 2, 4)
val commonNumbers = countCommonElements(as, bs, (x: Int, y: Int) => x == y)
// commonNumbers ==  3, for 8, 5, 4 are both in as and bs
```

e) Write a *main* function in object p3 that illustrates how to use the above functions with anonymous functions. Do not use the example code given in parts a)-d).

Problem 4. Partial application, composition, currying.

Write the code for this problem in a file called p4.scala, within an object called *p4*. Add in p4 an empty method *main*.
a) Partial function application.
a.1 Write a function ***partial2*** that takes as parameters a value *a:A* and a function *f:(A,B,C)=>D*, of 3 arguments, and returns as result a function of two arguments that partially applies *f* to *a*. This is an extension of the textbook function ***partial1*** from 2 arguments to 3 arguments.

a.2 Consider this function of three arguments:

```
def formatStudentInfo(isGraduate: Boolean, year: Int, name: String) = {
  val grad = if (isGraduate) "graduate" else "undergraduate"
  "%s student %s enrolled in year %d".format(grad, name, year)
}
```

Write code in function *main* that shows how to obtain a function of 2 parameters using ***partial2*** that partially applies *formatStudentInfo* with parameter *isGraduate* set to *true*. Assign the new function to a val called *formatGraduateStd*. Then write in *main* code that uses *formatGraduateStd* with some sample student information and display the resulting string.
(Hint: if this sounds complicated, the problem wants to use *partial2* in the same way the lecture notes show how to use *partial1* with a function of 2 arguments called *firstNChars*.)

b) Composition.
b.1 Consider these functions that format HTML elements:

```
def htmlTag(elem: String, inner: String) = "<%s>%s</s>n\n".format(elem, inner, elem)

val mkBold = (inner: String) => htmlTag("B", inner)
```

Define in object *p4's main* method an immutable variable called *mkItalic* initialized with a lambda expression that is similar to *mkBold* but instead of returning a boldface <B>...</B> element (like *mkBold*) it returns an italic face element, <I>….</I>.

b.2 Define in object *p4's main* method an immutable variable called *mkBoldItalic* initialized with a lambda expression that uses the *compose* method from trait Function2 and the two lambda expressions, *mkBold* and *mkItalic* to format an italic bold font style with the composed element <I><B>….<B></I>.

b.3 Define in object *p4's main* method an immutable variable called *mkItalicBold* initialized with a lambda expression that uses the *andThen* method from trait Function2 and the two lambda expressions, *mkBold* and *mkItalic* to format a bold italic font style with the composed element <B><I>….</I></B>.

c) Currying.
c.1 Write a function ***curry2*** that takes as parameters a function *f:(A,B,C)=>D*, of 3 arguments, and returns as result a function of type *A* that returns a function of *B* and *C* returning *D*, i.e. *curry2* returns type A => ((B,C) => D).
This is an extension of the textbook function ***curry*** from argument functions with 2 parameters, i.e. from (*A*, *B*) => *C* to 3 parameters, *(A,B,C)=>D*.
In other words, if *f*'s type is (*A,B,C*) => *D*, then *curry2*(*f*) returns a function of type *A* => ((*B, C*) => *D*). The function returned from *curry2*(*f*) takes an *a:A* argument and returns a function that partially applies *f* to *a*.
Hint: use *curry*()'s source code given in the lecture notes as a template.

c.2 Use the *curry2* and *formatStudentInfo* functions to write in p4's *main* method a lambda expression stored in variable *mkStdInfo* that can be used to partially apply *formatStudentInfo* with a Boolean parameter, as shown next:

```
  val mkStdInfo = curry2(formatStudentInfo) : Boolean => (Int, String) => String

  val formatGradStdInfo = mkStdInfo(true)   // good for graduate student info

  println(formatGradStdInfo(2020, "Harry Potter"))
// displays: graduate student Harry Potter enrolled in year 2020
```

c.3 Use lambda expression *mkStdInfo* to write in p4's *main* method a lambda expression stored in variable *formatUndergradStdInfo* that can be used to format data for undergraduate students in the same way *formatGradStdInfo* is used for graduate students.

Problem 5. Traits, objects, and classes.

Write the code for this problem in a file called p5.scala, within an object called *p5*. Add in *p5* an empty method *main*.

a) Write a trait called *Shape* with two methods returning *Double*: *area* and *perimeter*.

b) Write a class called *Circle* with the appropriate instance variables that extends *Shape*.

c) Write a class called *Rectangle* with the appropriate instance variables that extends *Shape*.

d) Write code in *main* variables that uses these classes.

e) Initialize in *main* a variable (val) with an expression that returns an anonymous class implementing *Shape* that returns a perimeter of value 16.0 and an area 15.0.

Grading:

Each problem is worth 20 points. 3 points extra credit.

The points for programming problems are split into:

- code correctness: 85%
- programming style: 15%.

IMPORTANT NOTES:

- A submission that does not follow the instructions 100% (i.e. perfectly) will not get full credit.
- Submit the **h1.pdf** PDF file and the required .scala source files.
- Only submissions uploaded before the deadline will be graded.
- You have unlimited attempts to upload this assignment, but only the last one will be graded.