Functional Programming with Scala

Homework 2

Instructor: Dr. Ionut Cardei

The purpose of this programming assignment is to:
1. Chapter 3: practice with the basic data structures of List and Tree, with pattern matching, HOFs.
2. Chapter 4: practice with exception handling, the Option and Either ADTs, .

Read the textbook chapters and the **lecture notes posted online** for material and examples not covered in the texbook.

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.
When in doubt, please contact the instructor at icardei@fau.edu.

Coding Requirements

For this assignment we can 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 h2.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 builds and runs correctly.

5. Convert the Word file to PDF.
6. Upload the PDF file with the answers and the Scala files on Canvas.

In this class when we say "method" and "function" we assume they are **pure**. Impure functions and non-RT code will be penalized.

Where required use the List,Tree, Option, and Either source code from the textbook Scala source project linked from the Lecture Notes or from the source zip file.

Problem 1. List with Cons

Put the required functions in an object called List in file **p1.scala**, in a package called h2.p1. Start with the List code given in class, i.e. the List ADT using the *Cons* constructor. In the same file add a module called p1 with a *main* method where you will write part of the solutions.

a) Write in module p1 a **polymorphic** method called *switchHead* that takes as argument a list *lst* and switches the 1st and 2nd elements from *lst*. If the list has fewer than two elements throw an exception using function call *sys.error*("too short"). Use pattern matching.
Example usage:

```
val lst = List(0, 1, 2, 3)
println(List.switchHead(lst))    // prints Cons(1,Cons(0,Cons(2,Cons(3,Nil))))
println(List.switchHead(List(0))) // raises java.lang.RuntimeException: too short
```

Write in method p1.main two examples of using this method.

b) Write in module p1 a **polymorphic** method (with type parameter *A*) called *get* that takes as argument a list *lst* and an integer *n*. The function returns the element from *lst* with index *n*, considering the first element is at index 0. If the index is invalid (<0 or >= size of *lst*) then *get* throws an exception using function call *sys.error*("wrong index"). Use pattern matching.
Example usage:

```
val lst = List(0, 1, 2, 3)
println(List.get(lst, 3))    // prints 3
println(List.get(5))         // raises java.lang.RuntimeException: wrong index
```

Write in method p1.main two examples of using this method.

c) Write in module p1 a **polymorphic** method (with type parameter *A*) called *set* that takes as argument a list *lst*, an integer *n*, and *x* of type *A*. The function returns a new list identical to *lst* except for the element from *lst* with index *n* that is replaced by *x*. The first list element has index 0. If the given index is invalid (*n*<0 or >= size of *lst*) then *set* throws an exception using function call *sys.error*("wrong index"). Use pattern matching.
Example usage:

```
val lst = List(0, 1, 2, 3)
println(List.set(lst, 2, 100))  // prints Cons(0, Cons(1, Cons(100, Cons(3, Nil)))
```

Write in method p1.main two examples of using this method.

d) Write in module p1 a **polymorphic** method (with type parameter *A*) called *take* that takes as argument a list *lst: List*[*A*] and an integer *n*. The function returns a new list with the first *n* elements from *lst*. If *n* is invalid (<0 or >= size of *lst*) then *takeN* throws an exception using function call *sys.error*("wrong index"). Use pattern matching. This function is quite similar to *drop*.
Example usage:

```
 val lst = List(0, 1, 2, 3)
 println(List.takeN(lst, 2))   // prints Cons(0,Cons(1,Nil))
```

Write in method p1.main an example of using this method.

e) Write in module p1 a **polymorphic** method with this signature:

```
  def mapsq[A](lst: List[A])(f: A => A) : List[A]
```

mapsq returns a new list where each element *x* from the original list *lst* is replaced by expression *f*(*f*(*x*)). Use the *map* function.

Write in *main* an expression with *mapsq* that obtains a list of elements x * 4 for each element x of List(1,2,3,4), i.e. the same list as List(4,8,12,16). Use the _ notation for the lambda expression.

Write in *main* an expression with *mapsq* that obtains a list of elements x + 6 for each element x of List(1,2,3,4), i.e. the same list as List(7,8,9,10). Use the _ notation for the lambda expression.

Problem 2. Using the standard library List (::)

Put the required functions in an object called List in file **p2.scala**, in a package called h2.p2. Use the ***scala.collection.immutable.List*** class and **not** the List class detailed in the course. Hence, your solutions must rely on the **::** constructor to stand for *Cons*, if necessary. In the same file add a module called *p2* with a *main* method where you will write part of the solutions.

a) Write in method *p2.main* a statement that uses *List.foldRight* to print the sum of all **odd** numbers from List(13, -1, 0, 2).

b) Write in module *p2* a **polymorphic** method function called *toStrings* that takes a parameter *lst* of type List[A] and returns a List[String] where each element *x* of *lst* is replaced by *x.toString*. Use the *map* method.
Example usage:

```
   val numbers = List(13, -1, 0, 2)
   println(toStrings(numbers))   // prints List(13, -1, 0, 2), a list of String
```

Write in method *p2.main* an example of using this method with Double, different from the previous one.

c) Write in module *p2* a **polymorphic** method function called *compute* **using foldLeft**, that takes in the first parameter list a List[A], in the second parameter list a binary operator *f* of type (A, A) => A, and that applies *f* to all elements of this sequence, going left to right.
So, *compute*(List(a, b, c, d))(f) == f(f(f(a, b), c) , d)

Example usage:

```
 val numbers = List(13, -1, 0, 2)
println(compute(numbers)(_ + _))   // prints 14
```

Hint: The standard List trait has methods *head* and *tail* that work as expected.
Write in method *p2.main* an example of using this method different from the previous one.

d) Write in module *p2* a monomorphic method with this signature:

```
def splitSum(lst: List[Int])(p: Int => Boolean) : (Int, Int)
```

splitSum has two parameter lists to assist with type inference. It returns a tuple where the first element is the sum of all elements *x* from *lst* for which *p*(*x*) is true and the second element is the sum of all elements *x* from *lst* where for which *p*(*x*) is not true, i.e. expression $(\sum_{x \in lst \wedge p(x)} x, \sum_{x \in lst \wedge !p(x)} x)$
Write in method *p2.main* an example on a List[Int] that prints the tuple with the sum of even numbers and the sum of odd numbers from that list.

e) Extra credit: 3 points
Write in module p2 a **polymorphic** method with this signature:

```
def split[A](xs: List[A])(p: A => Boolean) : (List[A], List[A])
```

that returns a partition of list *xs* based on predicate *p*: the first element of the returned tuple is a list with elements *x* from *xs* where p(x) is true and the second element of the returned tuple is a list with elements *x* from *xs* where p(x) is false. The order of elements in these lists must be preserved. Use one of the *fold* methods.

Example usage:

```
val lst = List(0, 1, 2, 3)
println(split(lst)(_ < 2))     // prints (List(0, 1), List(2, 3))
```

Write in method *p2.main* an example of using this method different from the previous one.

f) Extra credit: 3 points
Write in module p2 a **polymorphic** method with this signature:

```
def filterFoldRight[A, B](xs: List[A], z: B)(p: A => Boolean)(f: (A, B) => B) : B
```

using *pattern matching* that applies the binary operator *f* in a right-to-left order only to elements *x* of list *xs* for which *p*(*x*) is true.

Write a second version called *filterFoldRight_1* that does the same thing using the *foldRight* function.

Write in method *p2.main* an example of using each of this function.

Problem 3. Tree

Put the required functions in an object called List in file **p3.scala**, in a package called h2.p3. Start with the *Tree* ADT code from the textbook. In the same file add a module called *p3* with a *main* method where you will write part of the solutions.

a) Write an expression in method *p3.main* that computes the product of all leaves in a tree of *Int*s using the *Tree.fold* method. If Scala type inference produces unreasonable compilation errors just don't use _ (underscore) for your lambda expressions.

b) Consider this tree variable:

```
val tree: Tree[Int] = Branch(Branch(Leaf(2), Leaf(3)), Branch(Leaf(4), Leaf(5)))
```

Write an expression in method *p3.main* with the *Tree.map* function that computes the tree with the same structure of variable *tree* but with each leaf value being formatted to a string. The result must be the same as the following Tree[String] :

```
Branch(Branch(Leaf("leaf 2"),Leaf("leaf 3")),Branch(Leaf("leaf 4"),Leaf("leaf 5")))
```

c) Write in module Tree (file p3.scala) a **polymorphic** method with this signature:

```
def toList[A](t: Tree[A]) : List[A]
```

that returns a list (i.e. standard library List) with all elements from the leaves in tree *t*. Use the *Tree.fold* method given from the textbook.
Hint: the List.++ method appends two lists.

Example usage:

```
val tree: Tree[Int] = Branch(Branch(Leaf(2), Leaf(3)), Branch(Leaf(4), Leaf(5)))
println(Tree.toList(tree))   // prints List(2, 3, 4, 5)
```

Write in method *p3.main* an example of using this method different from the previous one.

Problem 4. Option/Either

Write the solution in a file **p4.scala**, in a package called h2.p4. Use (or import) the Option and Either code given in the course. In the same file add a module called *p4* with a *main* method where you will write part of the solutions.

a) Unit testing is vital for building robust code. Write in module *p4* a simple unit test function that has this signature:

```
def testFunction2[A,B,C](testname: String, a: A, b: B, expected: C)(f: (A,B) => C)
        : Either[String, String]
```

This function returns Right(testname + " passed") if f(a,b) == expected and Left(testname + " failed") otherwise. Using the *Either* ADT allows the programmer to quickly separate failed vs. passed tests, e.g. using pattern matching.

The purpose of this function is to automate testing for functions with two arguments, as in the next example:

```
def sample_add(x: Int, y: Int): Int = x + y    // function we want to test
```

```
// we set up a "test vector" with values 10, 20, 30, where 30 is the expected return value:
val add_test_result = testFunction2("sample_add", 10, 20, 10 + 20)(sample_add)
println(add_test_result)
// a Right value means the test passed: prints Right(sample_add passed)
```

b) Write in module p4 a function *sumO* that takes as arguments *x* and *y* (two Option[Int] objects) and returns the sum of the values in *x* and *y* in a Some constructor or *None*, if at least one of *x* and *y* is *None*. Use the *map2* function to get credit.

Write a new version of this function called *sumO_1* using *flatMap* and *map*, to get credit.

Write a new version of this function called *sumO_2* using a *for comprehension*, to get credit.

Write code in *p4.main* that uses the *testFunction2* function to test *sumO*, *sumO_1*, and *sumO_2*. At least one of the tests must use a *None* value for parameter *a* or parameter *b*.

c) Consider the *lift* function given in the textbook that converts an A => B function into an Option[A] => Option[B] function. Write in module p4 a function called *lift2* that converts an (A,B)=>C function into an (Option[A], Option[B]) => Option[C] function. Don't do pattern matching in this function. Instead, use the utility functions from the Option object or a for comprehension.

Example usage:

```
  def sample_add(x: Int, y: Int): Int = x + y    // function we want to test

  val add_lifted = lift2(sample_add)
  println(add_lifted(Some(10), Some(20)))  // prints Some(30)
```

Problem 5. Validation with Option/Either

Write the solution in a file **p5.scala**, in a package called h2.p5. Use (or import) the Option and Either code given in the course. In the same file add a module called *p5* with a *main* method where you will write part of the solutions.

Consider the following Student class:

```
sealed case class Student(name: String, id: Int, grades: List[Double])
```

Write a function called *mkStudent* that parses those Strings parameters and produces a new Student class as a *Right* value if input parsing is successful or a *Left*(*exception*) if *name* is an empty or string parsing failed for *id* or any of the strings in the *grades* list. Use a **for comprehension**. The function's signature is:

```
def mkStudent(name: String, idstr: String, gradesstr: List[String]): Either[Exception, Student]
```

Example usage with a *Right* value:

```
val studentE = mkStudent("Jane", "1234", List("90.0", "95", "73", "78"))
println(studentE)
// prints Right(Student(Jane,1234,List(90.0, 95.0, 73.0, 78.0)))
```

Example usage with a *Left* value, when parsing failed:

```scala
val studentF = mkStudent("Jose", "5678", List("90.0", "X95", "73", "78"))
// prints Left(java.lang.NumberFormatException: For input string: "X95")
println(studentF)
```

Write in *p5.main* different 2 examples with *mkStudent* that produce (and print) a *Right* value and a *Left* value.

Grading:

Each problem is worth 20 points. 6 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 **h2.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.