Functional Programming with Scala

Homework 3

Instructor: Dr. Ionut Cardei

The purpose of this programming assignment is to:
Chapter 5: practice with non-strict functions, memoization with lazy vals, and the Stream data type.

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

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.

This homework has 4 problems.
Total score: 100 points + extra credit.

Homework Formatting and Delivery

1. Start a new Word document called h3.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 the 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.

Follow strictly all requirements to get 100% credit. E.g. if a problem says: “use function foldRight” then your solution must use function foldRight to get full credit.

Problem 1. Theory

a) Explain how a lazy val can be used to memoize the value of an expression passed as by-name parameter.

b) Extra credit 3 points * Extra credit 3 points * Extra credit: 3 points
Describe how the memoization and evaluation mechanism for a lazy val could be implemented by the Scala compiler.

c) Consider method foldRight from the textbook Stream trait:

```
def foldRight[B](z: => B)(f: (A, => B) => B): B =
   this match {
     case Cons(h,t) => f(h(), t().foldRight(z)(f))
     case _ => z
   }
```

Explain under what condition(s) will expressions (or functions) using *foldRight* stop traversing the stream before reaching its end in contrast to traversing the entire stream unconditionally. Give a code example NOT covered in class with a function or expression using *foldRight* with incremental computation that stops mid-stream and another example NOT covered in class where the entire stream is always evaluated.

d) Trace the evaluation of expression Stream(1, 2, 3, 4).takeWhile(_ < 3).exists(_ == 3) in the same way it is done for a different expression in Listing 5.3 on page 72 (or shown here), from the textbook. Use the same level of detail.

Problem 2. Unit testing

Write the solution in a file called p2.scala, in an object called p2.
a) Consider the function from Homework 2:

```
def testFunction2[A,B,C](testname: String, a: A, b: B, expected: C)(f: (A,B) => C)
     : Either[String, String] = {
   val actual = f(a, b)
   if (actual == expected) Right(testname + " passed")
   else Left(testname + " failed")
 }
```

Write a similar function called *testExpression* that tests expressions **passed by name** against an expected value and that returns *Right* or *Left* in the same way as testFunction2. Here are some examples how to use it:

```
def plus1(x: Int) = x + 1
def plus1_buggy(x: Int) = x - 1    // buggy by design

println(testExpression("divide", 24 / 2, 12))  // Right("divide OK passed")
println(testExpression("plus1", plus1(1), 2))  // Right("plus1 OK passed")

println(testExpression("plus1_buggy", plus1_buggy(1), 2))
// Left("plus1_buggy failed")
```

b) Write a polymorphic function called *tryOrElse* that has two non-strict parameters of type *A*, called *expr* and *defval*. Evaluating expression expr may throw an exception. If no exception is thrown, *tryOrElse* returns the result from evaluating *expr*. If *expr* throws an exception then *tryOrElse* returns the result from evaluating *defval*. Make sure *expr* is evaluated exactly once and *defval* is forced only in case *expr* fails with an exception.

Here are some examples:

```
val grade = tryOrElse("100".toInt, 0)
// grade == 100

val gradeBadFormat = tryOrElse("1X00".toInt, 0)
// gradeBadFormat == 0

val grades = List[Int]()
val gpa = tryOrElse(grades.sum / grades.length, -1.0)
// gpa == -1
```

Write in p2's main() function 3 test cases for *tryOrElse* using the *testExpression* unit test function from part a) that are different from the examples above.

c) Use the formal definition to prove that *tryOrElse* is a non-strict function.

Problem 3. Stream practice

Write the solution in a file called p3.scala, in an object called p3. Use (import) the version of the Stream types given in the textbook and discussed in class.
a) Write an expression using the *from* and *find* functions to find the first number that divides both 6 and 8.

b) Write an expression using the *from*, *filter*, *take*, and *map* functions that returns a stream with the first 10 square integers greater than a variable *n* initialized to 1000.

c) Use functions *from*, *foldRight*, *map*, and *take* to write an expression that returns a comma-separated string with the first 10 square numbers: “1,4,9,16,25,36,49,64,81,100,” .

d) Consider this stream declaration:

```
val st2 = Stream(1.1, 1, 1.15, 1.11, 0.85, 1.15, 1.23)
```

Write an expression using Stream’s methods and *st2* that computes the absolute value of the difference between successive elements of *st2* and returns *Some*(*d*) where *d* is the first such difference that exceeds 0.2 or returns *None* if no such difference exceeds that threshold. Do not write a custom recursive function. In our case with *st2*, the expression should return *Some*(0.26).
Hint: *drop* and *zipWith*.

e) Write a method in object p3 called *findIndex* that takes as arguments a Stream[A] and, in a separate argument list, a predicate *p: A => Boolean*, and that returns the index (starting from 0) in the stream *s* of the first element *a* for which *p(a)* is true. *findIndex* must use the methods of the Stream trait and should not be a custom recursive function.
Example usage:

```
println(findIndex(from(0) take 10)(_ == 30))  // None
println(findIndex(from(0) take 10)(_ >= 5))   // Some(5)
```

Write 3 tests using *testExpression* for testing this function. One should be for the boundary case, when the predicate fails to find any element.

f) Write a method in object p3 called *findPositions* that takes as arguments a Stream[A] and, in a separate argument list, a predicate *p: A => Boolean*, and that returns a Stream[Int] with the indices (starting from 0) in the stream *s* **for all** elements *a* for which *p(a)* is true. If none such values exist, the returned stream must be empty. *findPositions* must use the methods of the Stream trait and should not be a custom recursive function.
Example usage:

```
println(findPositions(Stream(1, 2, 1, 4, 0, 1, 3) take 20)(_ == 1) toList)
// List(0, 2, 5)

println(findPositions(Stream(1, 2, 1, 4, 0, 1, 3) take 20)(_ > 10) toList) //List()
```

Write 3 tests using *testExpression* for testing *findPositions*. One should be for the boundary case, when the predicate fails to find any element.

g) Extra credit 3 points * Extra credit 3 points * Extra credit: 3 points
Write a method in object p3 called *splitWith* that takes as arguments a Stream[A] and, in a separate argument list, a predicate *p: A => Boolean*, and that returns a tuple of Stream[A] objects, (*sTrue*, *sFalse*) so that stream *sTrue* gets the elements *a* for which *p(a)* is true and stream *sFalse* gets the elements *a* for which *p(a)* is false. s*plitWith* must use the methods of the Stream trait and should not be a custom recursive function.
Example usage:

```
val streams = splitWith(Stream(5, -3, 2, -1, 6, 0, 3, 2))(_ < 0)
println(streams._1.toList) // List(-3, -1)
```

```
println(streams._2.toList) // List(5, 2, 6, 0, 3, 2)
```

Problem 4. Infinite streams and corecursion

Write the solution in a file called p4.scala, in an object called p4. Use (import) the version of the Stream types given in the textbook and discussed in class.

a) Write a function called *list2stream* that takes as parameter a List[A] object reference and returns a Stream[A] with the objects from the list in the given order. Your function **must be corecursive**.
Example usage:

```
// converts the list to stream and back to a list:
println(list2stream(List(1,2,3,4)) toList) // List(1,2,3,4)
println(list2stream(List()) toList)        // List()
```

Write two unit tests in main() with *testExpression* for this function.

b) Write a function called *list2streamViaUnfold* that takes as parameter a List[A] object reference and returns a Stream[A] with the objects from the list in the given order. Your function **must use the unfold function**. (this function does the same thing as *list2stream* from part a)).
Example usage:

```
println(list2streamViaUnfold(List(1,2,3,4)) toList)  // List(1,2,3,4)
println(list2streamViaUnfold(List()) toList)         // List()
```

Write two unit tests in main() with *testExpression* for this function.

c) A linear congruential generator is an algorithm that creates a stream of pseudo-random numbers using the recursive formula: $X_{n+1} = (a\,X_n + c)\,\%\,m$.
Write a **corecursive** function called *rndStream* that takes as parameter the initial seed (X_0) and produces an infinite stream of pseudo-random numbers using this series with values a = 1103515245, m = 2^{31}, c = 12345, **and keeping only the 30 least significant bits for the values being returned**. Use the *Long* data type instead of *Int* where it is necessary to avoid overflow.
Example usage starting with seed = 0:

```
// List(0, 12345, 1406932606, 654583775, 1449466924,
// 229283573, 1109335178, 1051550459, 1293799192, 794471793)
println(rndStream(0) take 10 toList)
```

Check your random numbers against this list. Write a unit test in main() with *testExpression* for this function.

d) Write a corecursive function called *signs* that takes as parameter *n: Int* that and that returns an infinite stream +n, -n, +n, -n, ….
Example:

```
println(signs(1) take 5 toList)  // List(1, -1, 1, -1, 1)
```

e) Extra credit 3 points * Extra credit 3 points * Extra credit: 3 points
Write an expression using functions *foldRight*, *signs*, *take*, *from*, and *zipWith* that computes the sum $\sum_{i=1} \frac{(-1)^{i+1}}{i}$ for the first 500 terms. What does the sum of this series represent ?

Grading:

Each problem is worth 25 points + some 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 **h3.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.