**Final Exam: Advanced Programming (BSc-Level, Master-Level, MSc-Level)**

IT University of Copenhagen, Spring 2018: 28 May, 15:00

The exam consists of 12 tasks to be solved within 4 hours. The BSc students solve only the first 9 tasks. The Master students solve only the first 10 tasks. The MSc students solve all 12 tasks. You can use any function from the course (textbook, exercises) in the solutions, as well as standard library functions, unless stated otherwise in the question. You can access any written or electronic materials, also online, but you are not allowed to communicate with anybody during the exam.

By submitting, you declare to have solved the problems alone, without communicating with anybody.

**Submission.** Solve the tasks in the file `exam2018.scala` found in the zip file linked from LearnIt under "Exam Bundle" (right under the last assignment). Write your name and your ITU email in the top of the `exam2018.scala` file. Submit this file and only this file to **learnIT**. Do not convert it to any other format than `.scala`. Do not reorder the answers, and do not remove question numbers from the file. When free text answers are expected, write them as comments.

The only accepted file format is `.scala`.

**Criteria.** The answers will be graded manually. We will focus on the correctness of ideas and the use of the course concepts. We will be permissive on minor issues such as semicolons, other punctuation, small deviations in function names, switching between curried and uncurried parameters, etc. We will not check whether the type inference succeeds, if a human reader could infer types.

We **do not** recommend solving questions to the point when they compile and pass tests. Dependency problems and other technical issues can take a lot of time, so only do this, once you are done with drafting all answers.

Nevertheless, if you do compile, you can use the `build.sbt` file provided in the `.zip` archive linked above. It has the necessary library dependencies configured. The file also contains the course libraries that the solutions depend on.

**About this document.** This document is prepared as a literate program: it is a Scala file (the `monospaced` fragments are Scala) interleaved with comments in English (task descriptions) in Roman font. All Scala snippets together form a legal Scala program, after removing the questions. This is the same Scala program you will find in `exam2018.scala`. For completeness, this is the preamble of this Scala file:

⟨*scala*⟩≡

```
// Name: _____________
// ITU email: ________
package adpro.exam2018

import fpinscala.monoids.Monoid
import fpinscala.monads.Monad
import fpinscala.monads.Functor
import fpinscala.laziness.{Stream,Empty,Cons}
import fpinscala.laziness.Stream._
import fpinscala.parallelism.Par._
import scala.language.higherKinds
import adpro.data._
import adpro.data.FingerTree._
import monocle.Lens
```

## Question 1. Association Lists

**Task 1.** An association list is a `List[(K,V)]` storing pairs of values. The first element of each pair is a key (of type `K`), and the second element of each pair is a value (of type `V`).

Implement a `groupByKey` operation for association lists. Given an association list `l`, which might have multiple entries for each key `k`, it should produce an association list that has exactly one entry for each key `k`, paired with a list of all the values paired with `k` in the original list.

**Example:** `groupByKey (List(1->1,1->1,1->2,2->3))` should produce `List(1->List(1,1,2), 2->List(3))` (or any permutation of this list of lists).

**Hint:** For full points avoid using a map data structure from the standard library (Scala's `Map`, etc.). Association lists are the simplest representation for maps. Using hash maps and tree maps to implement association lists is a bit like using a car to move a bike—although it may improve speed, it defeats the purpose. Still, use a map, if no other solution comes to your mind.

⟨*scala*⟩+≡

```
object Q1 {

  def groupByKey[K,V] (l :List[(K,V)]) :List[(K,List[V])] = ???

}
```

## Question 2. Weaving Either

**Task 2.** Recall that `Either[A,B]` represents an error (`A`) or a result of a computation (`B`). Consequently, a value of type `List[Either[A,B]]` represents a list of computation results.

Write a function `f` that converts a list of type `List[Either[A,B]]` to `Either[List[A],List[B]]`, where the first component represents a list of all errors, if there were any, and the second component represents a list of all good results (only if all the computations were successful). The relative order of `A`s (or `B`s, depending what is returned) should be preserved from the input list.

⟨*scala*⟩+≡

```
object Q2 {

  def f[A,B] (results: List[Either[A,B]]) :Either[List[A],List[B]] = ???

}
```

### Question 3. Monads and Either

**Task 3.** Implement a monad instance for `Either[String,A]`. This is equivalent to implementing the monad instance for the type: `T[B] = Either[String,B]`. The instance should be right-biased, so should realize similar behavior to `Option`. For instance, `flatMap` and `map` should operate on instances of `Right`.

You can use any function, including `flatMap`, from the standard library implementation of `Either` (https://www.scala-lang.org/api/current/scala/util/Either.html) or from the book implementation of `Either`. Of course, stay within our course assumptions (pure, typed, etc.).

⟨*scala*⟩+≡

```
object Q3 {

  type T[B] = Either[String,B]
  implicit val eitherStringIsMonad :Monad[T] = ???
```

**Task 4.** Generalize your instance to work for any error type `A` (instead of `String`) by creating an implicit factory of instances of `Monad[Either[A,B]]` for any `A`. The definition has been started below. Replace the `???`.

⟨*scala*⟩+≡

```
  implicit def eitherIsMonad[A] = {
    type T[B] = Either[A,B]
    ???
  }

} // Q3
```

### Question 4. Streams and State

**Task 5.** Recall that we can take a `State` object, as defined in the course, and translate its transition function to a stream of generated actions. Consider the following automaton:

```
State[(Int,Int),Int] { x => (x._1, (x._2,x._1+x._2)) }
```

Answer the following questions:

A. What is the stream generated by this automaton if starting with with `(1,1)` as the initial state? Include an example prefix of 5 elements in your answer.

B. What is the state space (set of possible state values) of the automaton encoded by this `State` object with `(1,1)` as the initial state?

⟨*scala*⟩+≡

```
object Q4 {

   // Write the answers in English below.

   // A. ...

   // B. ...

}
```

### Question 5. Par

**Task 6.** Recall a function `forall` on `List`: `def forall[A] (l: List[A]) (p: A =>Boolean): Boolean`

It checks whether all elements of a list `l` satisfy the predicate `p`. Implement a function `parForall` that does the same in parallel: for a given list of `A`s and a predicate `p` it checks in parallel whether all elements satisfy the predicate.

⟨*scala*⟩+≡

```
object Q5 {

  def parForall[A] (as: List[A]) (p: A => Boolean): Par[Boolean] = ???

}
```

### Question 6. Types and Functions

**Task 7.** Recall that for a binary function `f: (A,B) => C` the function `f.curried` is of type `A => B => C`. Assume the following type declarations:

⟨*scala*⟩+≡

```
object Q6 {

  def apply[F[_],A,B](fab: F[A => B])(fa: F[A]): F[B] = ???
  def unit[F[_],A](a: => A): F[A] = ???

  val f: (Int,Int) => Int = _ + _
  def a :List[Int] = ???
```

Let `val x = apply (apply (unit (f.curried)) (a)) (a)`

The type inference fails for `x` in Scala, but the expression is actually well typed. What is the type of `x`? Include enough argument in your answer, so that we know it is not a guess.

⟨*scala*⟩+≡

```
  // Answer below in a comment:

  // ...

} // Q6
```

### Question 7. Map2

**Task 8.** Assume that a `map2` function is implemented (available) for lists:

⟨*scala*⟩+≡

```
object Q7 {

  def map2[A,B,C] (a :List[A], b: List[B]) (f: (A,B) => C): List[C] = ???
```

Implement `map3` for lists using `map2` (and just `map2`, so no standard library functions):

⟨*scala*⟩+≡

```
  def map3[A,B,C,D] (a :List[A], b: List[B], c: List[C]) (f: (A,B,C) => D) :List[D] = ???
```

**Task 9.** Solve the same tasks as Task 8 for arbitrary monad type constructor `M[_]` instead of `List`. Your solution should enforce that an instance of `Monad` exists for `M[_]`, in order to know that `map2` is available.

⟨*scala*⟩+≡

```
  // def map3monad ...

} // Q7
```

### Question 8. Finger Trees (MSc and Master students only)

**Task 10.** Implement a simple `filter` function on Finger Trees. You can use any structures and functions from the book or from the standard library. Our implementation of `FingerTrees` in Scala is included in the exam zip file.

**Hint:** If you decide to compile (not required), remember that it is easy to confuse the `Empty` constructor between finger trees, streams, and the standard library—so import from the right package.

⟨*scala*⟩+≡

```
object Q8 {

  def filter[A] (t: FingerTree[A]) (p: A => Boolean): FingerTree[A] = ???

}
```

**Question 9. Lenses (MSc students only)**

**Task 11.** Implement a `Lens` viewing an instance of `Either[A,B]` as an instance of `Option[B]`. This lens needs to use a default value, to fill in for `A`, whenever putting back a None value.

⟨*scala*⟩+≡

```
object Q9 {

  def eitherOption[A,B] (default: => A): Lens[Either[A,B],Option[B]] = ???
```

**Task 12.** Respond to the following questions about your lense from Task 11:

A. Does the above lens satisfy the put-get low, and why?

B. Does the above lens satisfy the get-put low, and why?

C. Does the above lens satisfy the put-put low, and why?

If the answer is positive give an argument, if the answer is negative, give a counter example.

⟨*scala*⟩+≡

```
  // Answer the questions below:

  // A. ...

  // B. ...

  // C. ...

} // Q9
```