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

IT University of Copenhagen

Spring 2017

The exam consists of 12 tasks to be solved within 4 hours. The BSc students answer only the first 10 questions. The Master students answer only the first 11 questions. The MSc students answer all 12 questions. You can use any function from the course (textbook, exercises) in the solutions, as well as standard library functions. 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 `exam2017.scala` found in the zip file at `http://itu.dk/people/wasowski/6e5cd/684.zip`. Write your name and your ITU email in the top of the `exam2017.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 in the same file.

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 `exam2017.scala`. For completeness, this is the preamble of this Scala file:

⟨*scala*⟩≡

```
// Your name and ITU email: ____
package adpro.exam2017

import scala.language.higherKinds
import fpinscala.monoids.Monoid
import fpinscala.monads.Functor
import fpinscala.state.State
import fpinscala.laziness.{Stream,Empty,Cons}
import fpinscala.laziness.Stream._
import fpinscala.parallelism.Par._
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 the function `hasKey` for association lists. Given an association list `l` and a key `k`, it should return true if and only if there exist an element with that key on the list.

⟨*scala*⟩+≡

```
object Q1 {

  def hasKey[K,V] (l: List[(K,V)]) (k: K) :Boolean = ???
```

Task 2. Implement a `reduceByKey` operation, which produces an association list (with unique keys) combining values with the same key component using the reduction operator `ope`. Keep your solution simple, do not optimize for efficiency.

You may use the function `hasKey` from Task 1 (even if you failed to implement it).

⟨*scala*⟩+≡

```
  def reduceByKey[K,V] (l :List[(K,V)]) (ope: (V,V) => V) :List[(K,V)] = ???
```

Task 3. Consider the following function transforming an association list of identifiers paired with tokenized sentences, into list of identifiers paired with single tokens (a problem extracted from our sentiment analysis project). Rewrite this function using for-yield expression instead of `map` and `flatMap`.

⟨*scala*⟩+≡

```
  def separate (l :List[(Int,List[String])]) :List[(Int,String)] =
    l flatMap { idws => idws._2 map { w => (idws._1,w) } }

  def separateViaFor (l :List[(Int,List[String])]) :List[(Int,String)] = ???

} // Q1
```

Question 2. Strange Trees

Task 4. Consider the following binary tree type that embeds lists of elements of type `A`, instead of simple elements in the internal nodes.

⟨*scala*⟩+≡

```
object Q2 {

  trait TreeOfLists[+A]
  case object LeafOfLists  extends TreeOfLists[Nothing]
  case class BranchOfLists[+A] (
    data: List[A],
    left: TreeOfLists[A],
    right: TreeOfLists[A]
  ) extends TreeOfLists[A]
```

Generalize these types to use *any* generic collection `C[_]` instead of `List[_]`

⟨*scala*⟩+≡

```
  // trait TreeOfCollections[...]
  // case class LeafOfCollections ...
  // case class BranchOfCollections ...
```

Task 5. For the `TreeOfLists` we use a special `map` function, that takes a transformer function for elements of type `A` and applies it to all `A`s in all lists in all nodes of a tree:

⟨*scala*⟩+≡

```
  def map[A,B] (t: TreeOfLists[A]) (f: A => B) :TreeOfLists[B] = t match {
    case LeafOfLists => LeafOfLists
    case BranchOfLists (data,left,right) =>
        BranchOfLists (data map f, map (left) (f), map (right) (f))
  }
```

We want to generalize this `map` function from Lists to any collection `C[_]`, so to use `TreeOfCollections` instead of `TreeOfLists`. This requires assuming that an instance of `Functor[C]` exists (textbook Chapter 11, also in the exam `.zip` file). Write the generalized `map` function below.

⟨*scala*⟩+≡

```
  // def map[...] (t: TreeOfCollections[...]) (f: A => B) ...

} // Q2
```

Question 3. Parameter Passing Semantics

Task 6. Consider the following two functions:

⟨*scala*⟩+≡

```
object Q3 {

  def p (n: Int): Int = { println (n.toString); n }

  def f (a: Int, b: Int): Int = if (a > 10) a else b
```

Note that the argument of `p` is call-by-value; We will be varying the argument passing semantics of `f`. What is printed by the following expression: `p ( f( p(42), p(7) ) )` ?

A. Assuming that `f` is call-by-value in both arguments?

B. Assuming that `f` is call-by-name in both arguments?

C. Assuming that `f` follows the lazy evaluation (call-by-need) strategy for both arguments? (note: not available directly in Scala).

⟨*scala*⟩+≡

```
  // Answer the questions in comments here

  // A. ...

  // B. ...

  // C. ...

} // Q3
```

Question 4. State

Task 7. We consider a simple finite state automaton that models a coffee maker. The machine has two inputs: you can insert a Coin, and you can press Brew. The machine has a counter estimating the amount of coffee bean portions available in storage, and another integer value counting the accumulated coins. We model the inputs using the following types:

⟨*scala*⟩+≡

```
object Q4 {

  sealed trait Input
  case object Coin extends Input
  case object Brew extends Input
```

The machine has two discrete states: a ready state (accepting coins only) and a busy state (aka not ready), in which one can press the Brew button to make coffee. The following type captures the entire state of the automaton:

⟨*scala*⟩+≡

```
  case class MachineState (ready: Boolean, coffee: Int, coins: Int)
```

The transition rules of the automaton are:

- Ignore all the input if it contains no coffee beans (no state change)
- Inserting a coin into a machine causes it to become busy (if there are still coffee bean portions left). It also increases the number of coins accumulated.
- Pressing Brew on a busy machine will cause it to deliver a cup of coffee (which changes the state by taking one coffee portion away) and return to the ready state.
- Pressing Brew on a machine which is not busy (a ready machine) has no effect
- Inserting two or more coins in a row is possible, but you will only get one coffee anyway. The machine is a bit simplistic: the new coin is just gladly consumed.

Implement a function step that given an input and a state computes a new state, as per the above rules:

⟨*scala*⟩+≡

```
  def step (i: Input) (s: MachineState) :MachineState =  ???
```

Task 8. The method `simulateMachine` (type below) should execute the machine based on the list of inputs and return the number of coffee bean portions and coins accumulated at the end. For example, if initially the machine has 42 coffee portions and 10 coins, and a total of 4 coffees are successfully sold for one coin each, then the output should be `(38, 14)`.

Use the `State` monad type from the course to implement this function.[1] You shall use the function `step` from the previous task (even if you have not implemented it).

⟨*scala*⟩+≡

```
    def simulateMachine (initial: MachineState) (inputs: List[Input]) :(Int,Int) =  ???

} // Q4
```

[1] Implementations of basic types (here `State`) for this and the following tasks are included in the exam `.zip` file for reference

Question 5. Lazy Streams with Buffering

Task 9. Imagine that we want to model as a lazy stream a data access activity that is relatively slow. With this data access it is expensive to force one element (say downloading from a remote location) but it is relatively faster to take elements in batches of `k` (`k` is a small natural number), as the cost of transmission is dominated by the cost of establishing the connection.

To treat such data efficiently in a lazy stream we can model this as a stream of type `Stream[List[A]]`, where each list in the stream has `k` elements fetched. Now forcing one element of the string will load (buffer) the entire list of `k` elements. However, it is inconvenient to program with streams of lists, if the rest of the program logic needs to process the elements of type `A` one-by-one.

Implement a function `flatten` that converts a `Stream[List[A]]` to a `Stream[A]` in the obvious way (as if the lists where 'concatenated into a single stream'). Do so carefully, to avoid forcing the next list in the stream if not necessary. It is fine to force the head of the stream always (as often done earlier in the course), but do not force deeper elements. The textbook implementation of lazy streams is included in the exam archive for your convenience.

⟨*scala*⟩+≡

```
object Q5 {

  def flatten[A] (s: =>Stream[List[A]]) :Stream[A] = ???

} // Q5
```

Question 6. Parallel HOFs

Task 10. Recall the function exists from the standard list interface:

```
def exists[A] (l: List[A]) (p: A =>Boolean): Boolean
```

The function checks whether at least a single element of a list `l` satisfies the predicate satisfy the predicate `p`.

Implement a simple parallel version of exists that would work well on lists shorter than the number of available threads, for predicates that take longer time to compute. This new function, `parExists`, should check in parallel whether each and single element of the list `l` satisfies the predicate `p`, and then should combine the results so that the outcome is equivalent to the sequential exists.

The resulting type should be `Par[Boolean]` (so a computation that returns Boolean, if scheduled on a parallel computing resource; as defined in the course). The implementation of the `Par` type, in the textbook version, has been included in the exam `.zip` file for your convenience.

⟨*scala*⟩+≡

```
object Q6 {

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

} // Q6
```

Question 7. Finger Trees (MSc and Master students only)

Task 11. The Hinze/Paterson paper on finger trees introduces a logarithmic time concatenation operation. A simpler concatenation method for deques can be implemented in linear time (in the size of the deque), by iteratively adding all the elements of the second (the right) deque to the right of the first (the left) deque. Implement this operation.

If you need, you can rely on other elements of the `FingerTree` implementation than quoted here. The entire implementation is included in the `.zip` file for your convenience.

⟨*scala*⟩+≡

```
object Q7 {

  //  def reduceL[A,Z] (opl: (Z,A) => Z) (z: Z, t: FingerTree[A]) :Z = ??? // assume that this is implemented
  //  def reduceR[A,Z] (opr: (A,Z) => Z) (t: FingerTree[A], z: Z) :Z = ??? // assume that this is implemented

  //  trait FingerTree[+A] {
  //  def addL[B >:A] (b: B) :FingerTree[B] = ??? // assume that this is implemented as in the paper
  //  def addR[B >:A] (b: B) :FingerTree[B] = ??? // assume that this is implemented as in the paper
  // }

  // Implement this:

  def concatenate[A, B >: A] (left: FingerTree[A]) (right: FingerTree[B]) :FingerTree[B] = ???

} // Q7
```

Question 8. Lenses (MSc students only)

Task 12. Implement a lens `nullOption` between the concrete type `T` and the abstract view type `Option[T]`. This lens should translate a value `c` of type `T` into `Some(c)` and a `null` value into `None`. Use the Monocle interface for instantiation lenses (like in the course exercise).

⟨*scala*⟩+≡

```
object Q8 {

  // def nullOption[T] = Lens[...]
```

Then respond to the three following questions:

A. Does this lens obey the put-get law, and why?

B. Does this lens obey the get-put law, and why?

C. Does this lens obey the put-put law, and why?

⟨*scala*⟩+≡

```
  // Answer the questions below:

  // A. ...

  // B. ...

  // C. ...

} // Q8
```