Final Exam: Advanced Programming

IT University of Copenhagen

Spring 2016

The exam consists of 10 tasks. You have 4 hours to solve them.

Open book exam. You can use any functions from the course (textbook, exercises) in your solutions. You can access any written or electronic materials, also online, but you are not allowed to communicate with anybody during the exam.

By handing the solutions you declare, that you have solved all the problems honestly yourself, without communicating with anybody.

Submission Choice 1. We recommend to solve the tasks in the Scala file available at `http://goo.gl/E38eq2`. Write your name and your ITU email in the top of the file. Then hand in the file by submitting it to **learnIT**. Don't convert it to any other format. Don't reorder the answers, and don't remove task numbers from the file.

Submission Choice 2. Alternatively, you can type answers into a text document containing your name and ITU email on the first page. The tasks must be solved in order and the answer to each task must be preceded with a task number. If you do not use a Scala file, then convert the text document to a PDF file before submitting.

Other formats than `.scala` and `.pdf` will not be approved.

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 small syntactic issues (semicolons, punctuation, small deviations in function names, switching between curried and uncurried parameters, unless the question is about currying, etc.). We will not check whether the type inference succeeded (if a human reader could infer types).

We **don't** recommend solving questions to the point when they compile. 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 on **learnIT** for the trial exam. It has the necessary library dependencies configured, but you may need to include some `fpinscala` files in `src/` if you use other functions in your solutions.

About this document. This document is prepared as a literate program: it is a Scala file (the `monospaced` fragments are Scala) interleaved with natural language comments (questions) in roman font. All Scala snippets together, after removing the questions form a legal Scala program.

Question 1. Referential Transparency

Task 1. The following function computes a 16 bit checksum of a character string.

⟨*scala*⟩≡

```
package adpro.exam2016
object Q1 {
   def checksumImp (in: String) :Int = {
    var result = 0
    for (c <- in.toList)
      result = (result + c.toInt) % 0xffff
    return result
  }
```

Translate this function to a pure one:

⟨*scala*⟩+≡

```
   def checksumFun (in :String) :Int = ??? // Task 1.
```

Task 2. Explain whether your solution is tail-recursive and show where concretely in the code we can see whether it is tail recursive or not.

⟨*scala*⟩+≡

```
   // Write your answer for Task 2 here.
}
```

Question 2. Functors

Task 3. Implement a function `onList` that promotes any `A => A` function to operate on lists of `A`s point-wise (so `f` is called on each element of the list separately). For example, if call `f(c)` capitalizes a character `c`, then `onList (f) (l)` capitalizes an entire list of characters `l`.

⟨*scala*⟩+≡

```
object Q2 {
  import fpinscala.monads.Functor
  import scala.language.higherKinds

  def onList[A] (f: A => A) :List[A] => List[A] = ??? // Task 3.
```

Task 4. Now assume any collection constructor `C[_]` for which we have a type class instance `Functor[C]` as seen in the textbook. Generalize `onList` to a function `onCollection` that provides the same functionality for any collection:

⟨*scala*⟩+≡

```
  def onCollection[C[_],A] (f: A => A)
    (implicit functorC: Functor[C]) :C[A] => C[A] = ??? // Task 4.

}
```

Question 3. Folds and Monoids

Task 5. Implement a function `foldBack` that folds an operator of a monoid `M`, by traversing through the list twice. `M` is an instance of `Monoid[A]` for some type `A`, as defined in the textbook.

For example for

`l=List(x1,x2,x3),`

the call `foldBack (l) (M)` should compute:

`(z + (x1 + (x2 + (x3 + (x3 +  (x2 + (x1 + z))))))) ,`

where `z` is the zero of the monoid `M`, and plus is the operator in `M`.

⟨*scala*⟩+≡

```
object Q3 {

  import fpinscala.monoids.Monoid
  import scala.language.higherKinds

  def foldBack[A] (l :List[A]) (implicit M :Monoid[A]) :A = ??? // Task 5.

}
```

Question 4. Monads

Task 6. Let the following type represent a computation that may fail. The result of the computation is either a value of type `A` or an error message stored in a string of characters.

⟨*scala*⟩+≡

```
object Q4 {

  type Computation[A] = A => Either[A,String]
```

Implement a function `run` that given a list of computations and an initial value of type `A`, executes the computations from the head to tail sequentially. After each successful computation, the next computation should get the result of the previous one as the input. If any computation fails, then `run` tries the next one on the list (with the same input value as used in the previous one). The function shall collect all error messages during the traversal. If all computations failed, the initial value is returned.

⟨*scala*⟩+≡

```
  def run[A] (init: A) (progs: List[Computation[A]])
    : (A,List[String]) = ??? // Task 6.

}
```

Question 5. Lazy Trees

Task 7. Consider a type of lazy binary trees:

⟨*scala*⟩+≡

```
object Q5 {

  sealed trait Tree[A]
  case class Branch[A] (l: () => Tree[A], a: A, r:() => Tree[A]) extends Tree[A]
  case class Leaf[A] (a: A) extends Tree[A]
```

Implement a function that computes the multiplication of values in a `Tree[Int]`. The function should stop exploring the possibly infinite tree as soon as the result of the multiplication is zero. Thus on finite trees the multiplication terminates, while on infinite trees it terminates if it can be established that it equals zero in a finite number of steps.

⟨*scala*⟩+≡

```
  def multiply (t: Tree[Int]) :Int = ??? // Task 7.
```

Task 8. Describe in English (or Danish) how would you test this function. In particular, explain how can you ensure that the function is not overly eager, exploring too large part of the tree.

⟨*scala*⟩+≡

```
  // Task 8. (answer below in a comment)

}
```

Question 6. Strange Numbers

Task 9. Consider the following types for representing Peano's definition of natural numbers:[1]

⟨*scala*⟩+≡

```
object Q6 {

  sealed trait Nat[+A]
  case object Zero extends Nat[Unit]
  case class Succ[A] (pred: A) extends Nat[A]
```

The first three natural numbers are represented as follows:

⟨*scala*⟩+≡

```
  val zero /* : ... */ = Zero           // Task 9.
  val one  /* : ... */ = Succ (zero)    // Task 9.
  val two  /* : ... */ = Succ (one)     // Task 9.
```

Write down type annotations for the above values (`zero`, `one`, `two`) in terms of the `Nat` type constructor and the `Unit` type (so write what are the Scala types of `zero`, `one`, and `two` using the `Nat` type constructor).

Task 10. Write a function `plus2` that takes a representation of a natural number, and returns a representation of the number bigger by two. Make the type annotations in the function signature explicit. Make sure that the function works for all input numbers, zero, one, and so on.

⟨*scala*⟩+≡

```
  // def plus2  (x : ... ) : ... = ???       // Task 10.

}
```

[1] You don't need to know Peano's definition of natural numbers to complete the exercise. This is not a practical representation for programming. It is used only to construct the question.