030 Errors and Partial Computations

This exercise set assumes that you have read chapters 1–4 of [Chiusano, Bjarnason 2015]. Exercises marked **[–]** are meant to be easy, and can be skipped by students that already know functional programming and feel comfortable.

Solve the exercises in the file `Exercises.scala`, inline. This is the only file you shall need to hand in. Uncomment function declarations as you proceed. In the bottom of the file you can find tests, to check your solutions. Uncomment them as you go. Make sure that you understand the test cases, and try to add some yourself to confirm understanding.

The entire exercise set is normed to take about 4-6 hours of intensive and focused programming. All functions are extremely short (most of them one line long, but require careful attention to formulate).

We start with few simple exercises on ADTs and traits (Chapter 3), before we proceed to Chapter 4. First, we see how to use traits to implement dependency injection in existing code base. Then, we have a second look at simple ADTs using trees. In Chapter 4, we experience advanced functional programming for the first time (especially using `map`, `flatMap`, for comprehensions, `sequence` and `traverse` with partial computations encapsulated in `Option` values).

As usual, do not use variables, side effects, exceptions or return statements. Additionally, avoid using pattern matching in Exercises from Chapter 4, if possible. Once you implemented monadic API for your ADT, you should not need to use pattern matching too much.

Exercise 1. (25 minutes) This exercise is not about functional programming, but about traits (a feature of Scala that is independent of functional programming). We will use it to obtain a simple form of dependency injection. We will extend an existing class `java.awt.Point` with a new set of operators (comparisons).

Define a class `OrderedPoint` by mixing `scala.math.Ordered[Point]` into `java.awt.Point`. Use the lexicographic ordering, i.e. $(x, y) < (x', y')$ iff $x < x'$ or $x = x' \wedge y < y'$, symmetrically for greater than. Test the extension in the Scala REPL.[1]

Notice, that the new class only has the default constructor (unless you implemented some more). You can only call `new OrderedPoint()`. We lost the constructors of the original `Point` class, as constructors are not inherited.

This would not be necessary if we mixed the `OrderedPoint` in at object creation time (like in the slides) instead of creating a new class. Reimplement `OrderedPoint` as a trait. To make it work you will have to restrict the trait to classes extending `java.awt.Point`. Find out what are trait's self-types (online, or in Scala books) and use them to achieve this. Then instantiate `java.awt.Point(1,2) with OrderedPoint` and see whether your implementation behaves correctly.

Note that we can use infix comparison operators this way with classes that were defined in the `java.awt` package long before Scala existed, without modifying their source or recompiling them.

Exercise 2 [–]. (14 minutes) Write a function `size` that counts the number of nodes (leaves and branches) in a tree.[2]

Exercise 3 [–]. (10 minutes) Write a function `maximum` that returns the maximum element in a `Tree[Int]`. Note: In Scala, you can use `x.max(y)` or `x max y` to compute the maximum of two

[1]Exercise 10.2 [Horstmann 2012]
[2]Exercise 3.25 [Chiusano, Bjarnason 2015]

integers `x` and `y`.[3]

Exercise 4 [–]. (12 minutes) Write a function `depth` that returns the maximum path length from the root of a tree to any leaf. The path length is defined as the number of edges on the path.[4]

Exercise 5 [–]. (10 minutes) Write a function `map`, analogous to the method of the same name on `List`, that modifies each element in a tree with a given function.[5]

Exercise 6. (35 minutes) Generalize `size`, `maximum`, `depth`, and `map`, writing a new function `fold` that abstracts over their similarities. Reimplement them in terms of this more general function.[6]

Exercise 7. (35 minutes) Implement `map`, `getOrElse`, `flatMap`, `orElse`, `filter` on `Option`. As you implement each function, try to think about what it means and in what situations you'd use it. Refer to the book's Chapter 4, and Exercise 4.1 for hints and context information, if you do not know what to do.[7]

Exercise 8. (20 minutes) Implement the `variance` function in terms of `flatMap`. If the mean of a sequence is `m`, the variance is the mean of `math.pow(x - m, 2)` for each element `x` in the sequence.[8]

This is a realistic exercise, in the sense that a variance computation, is something that you could need to implement in a machine learning or a data analytics application. Don't use pattern matching in this exercise (use the `isEmpty` method if you need to see if a sequence is empty). You will have (likely your first) chance to experience a computation in a monad.

If you feel comfortable with functional programming this is a good point to start experimenting with for comprehensions. Try to write your implementation using for comprehensions, without using `flatMap` (using `List.map` is OK to use here). Once succeeded, reflect whether this code is more readable than the code using `flatMap` directly. Notice that, it looks strangely similar to imperative code in Scala/Java/C#...

On the other hand, if you are overwhelmed, you can skip the for comprehensions for now, and revisit this (and similar exercises) in a week or two.

Exercise 9. (15 minutes) Write a generic function `map2` that combines two `Option` values using a binary function. If either `Option` value is `None`, then the return value is too. Do not use pattern matching in this exercise. [9]

The entire exercise makes much more sense if you read section 4.3.2 in the book, until the Exercise 4.3.

After you are done, have a look in the end of Section 4.3 in how this can be rewritten using for comprehensions in Scala.

Exercise 10. (35 minutes) Write a function `sequence` that combines a list of `Option`s into one

[3]Exercise 3.26 [Chiusano, Bjarnason 2015]
[4]Exercise 3.27 [Chiusano, Bjarnason 2015]
[5]Exercise 3.28 [Chiusano, Bjarnason 2015]
[6]Exercise 3.29 [Chiusano, Bjarnason 2015]
[7]Exercise 4.1 [Chiusano, Bjarnason 2015]
[8]Exercise 4.2 [Chiusano, Bjarnason 2015]
[9]Exercise 4.3 [Chiusano, Bjarnason 2015]

`Option` containing a list of all the `Some` values in the original list. If the original list contains `None` even once, the result of the function should be `None`; otherwise the result should be `Some` with a list of all the values.

Do not use pattern matching, and recall that you have `foldRight` available on lists. A solution fits a (longish) single line.

NB. This is a clear instance where it's not appropriate to define the function in the OO style. This shouldn't be a method on List (which shouldn't need to know anything about Option), and it can't be a method on Option, so it goes in the Option companion object.[10]

This function captures a very realistic situation, where you have a bunch of results from computations that may fail, and the entire list has no value to you, if at least one of these failed. It demonstrates how to handle 'exceptions' in bulk in functional style. This function sequences computations in the `Option` monad. We shall see more interesting instances of sequencing later on, and eventually a formal definition of a monad in the end of the course.

Exercise 11. (8 minutes) Implement function `traverse`. It's straightforward to do using `map` and `sequence`, but try for a more efficient implementation that only looks at the list once (then `sequence` can be implemented in terms `traverse`, but we shall skip that part of the exercise).[11]

[10]Exercise 4.4 [Chiusano, Bjarnason 2015]
[11]Exercise 4.5 [Chiusano, Bjarnason 2015]