030 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.

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 much.

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

Exercise 1. This exercise is 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 trait `OrderedPoint` extending the generic `scala.math.Ordered[Point]`. Implement the missing method `compare` from `Ordered[Point]`, using the lexicographic ordering, i.e. $(x, y) < (x', y')$ iff $x < x'$ or $x = x' \wedge y < y'$.

You will need to restrict the trait to only be allowed to be mixed into subclassess of `java.awt.Point` to access the x and y components of the objects. This is done by inserting the following constraint in the beginning of the trait block: “`this :java.awt.Point =>`” (Google for trait’s self-types if you want to know more about these constraints).

Now mix the new `java.awt.Point` into `java.awt.Point` and create some ad hoc point objects using this mixin. Test the extension in the Scala REPL by comparing some point instances using the less than ($<$) operator.[1]

This way, 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 [–]. Write a function `size` that counts nodes (leaves and branches) in a tree.[2]

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

Exercise 4 [–]. Write a function `map`, analogous to the method of the same name on `List`, that

[1] A variation of Exercise 10.2 [Horstmann 2012]
[2] Exercise 3.25 [Chiusano, Bjarnason 2014]
[3] Exercise 3.26 [Chiusano, Bjarnason 2014]

modifies each element in a tree with a given function.[4]

Exercise 5. Generalize `size`, `maximum`, and `map`, writing a new function `fold` that abstracts over their similarities. Reimplement them in terms of this more general function.[5]

Exercise 6. Implement `map`, `getOrElse`, `flatMap`, `filter` on `Option`. As you implement each function, think 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.[6]

Exercise 7. 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.[7]

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 (use the `isEmpty` method if you need to see if a sequence is empty). This is likely your experience of 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 8 [+]. 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 `None` too. Do not use pattern matching.[8] The 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 9 [+]. Write a function `sequence` that combines a list of `Option`s into one `Option` containing a list of all the `Some` values in the original list:

```
def sequence[A] (aos: List[Option[A]]): Option[List[A]]
```

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.

NB1. 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 has failed. So `sequence` handles '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.

NB2. This is an example, where it seems inappropriate to define the function as a method in the

[4] Exercise 3.28 [Chiusano, Bjarnason 2014]
[5] Exercise 3.29 [Chiusano, Bjarnason 2014]
[6] Exercise 4.1 [Chiusano, Bjarnason 2014]
[7] Exercise 4.2 [Chiusano, Bjarnason 2014]
[8] Exercise 4.3 [Chiusano, Bjarnason 2014]

object-oriented style. The function `sequence` likely should not be a method on a `List`, as this would tangle `List` to `Option`, which appears not very natural. Sequencing partial computations belongs better with `Option`, which is concerned with partial computations. However, `sequence` cannot be a method on `Option` as its argument is `List`! Thus we put it in companion object of `Option`.[9]

Exercise 10. Implement function `traverse`:

```
def traverse[A,B] (a: List[A]) (f: A =>Option[B]): Option[List[B]]
```

The function behaves like `map` executed sequentially on the list `a`, where the mapped function `f` can fail. If at least one application fails, then the entire computation of the mapping (traversal) fails.

It is easy to solve using `map` and `sequence`, but try for a more efficient implementation that only looks at the list once (Incidentally, `sequence` can be implemented in terms `traverse`, but we shall skip that part of the exercise from the textbook).[10]

[9] Exercise 4.4 [Chiusano, Bjarnason 2014]
[10] Exercise 4.5 [Chiusano, Bjarnason 2014]