Option: Partial Computations & Handling Errors without Exceptions

This exercise set assumes that you have read chapters 1–4 of [Pilquist, Bjarnasson, Chiusano 2023]. This counts as one of the mandatory hand-ins. See detailed rules on the course website. Hand in only the file `Exercises.scala`. **No zip files are accepted.**

This week we start to use the implementation of standard library `List`.[1] The main difference is that `Cons` is an infix operator "`::`". Of course, the standard library also supports many more functions.

We start with few warm-up exercises on ADTs and traits (Chapter 3).

Exercise 1 [M]. 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, so we will inject new functionality to the old types, which is an important thing to be able to do in long-living systems, to maintain a clean implementation, while they are being evolved.

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.

$$(x, y) < (x', y') \text{ if and only if } x < x' \lor (x = x' \land y < y') \ . \tag{1}$$

The method should return +1 if the left argument is larger, -1 if the right argument is larger, and zero if both sides are equal.

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 trait. Test the extension in the Scala REPL by comparing some point instances using the less than ($<$) operator.[2]

Reflect on what has just happened: We can use infix comparison operators with classes that were defined in the `java.awt` package. These classes existed long before Scala existed, but we did not need to modify their source or to recompile them. So we adapted an old class to a new library without having access to its source code.

With this solution though, we still need to control how objects are created, as we need to mix a trait in, at the object creation type. Later in the course, we will see how to extend classes even though we do not control when and how the objects are created. (Await the concepts of *extensions* and *type classes*.)

Exercise 2 [E]. Write a recursive function `size` that counts nodes (leaves and branches) in a tree.

Exercise 3 [E]. Write a recursive 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 [E]. Write a function `map`, analogous to the function of the same name on `List`, that modifies each element in a tree with a given function transforming the elements.

[1] https://www.scala-lang.org/api/current/scala/collection/immutable/List.html
[2] A variation of Exercise 10.2 [Horstmann 2012]
[3] Exercise 3.25 [Pilquist, Chiusano, Bjarnason, 2023]

Exercise 5 [H]. Generalize `size`, `maximum`, and `map`, writing a new function `fold` that abstracts over their similarities. Reimplement them in terms of this more general function.[4]

Hint: The type inference fails more often than usual when implementing higher order generic functions. It is convenient to make the type parameters explicit when calling fold, for instance, write `fold[Int,Int]`, when folding over a tree of integers and producing an integer.

We are now leaving the trees, and moving to the type `Option`.

Exercise 6 [M]. Implement `map`, `getOrElse`, `flatMap`, `filter` on `Option` (Chapter 4). This time, just to train the difference, we implement them as methods (member functions), not as static functions. 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.

In this exercise, you shall use pattern matching. We implement these standard combinators using pattern matching, in order to be able to reduce its use later on. There is nothing wrong with using pattern matching per se, but for an inexperienced programmer it might be a signal that a standard more abstract combinator should have been used, but has not been identified.[5]

Note: Now that you had implemented a monadic API for Option, you should not need to use pattern matching much below.

Exercise 7 [E]. A grading sheet is a list of pairs, where the first component in each pair is a String storing a name, and the second component in each pair is an integer representing the exam result. For example: `List[("Alice", 12), ("Bob", 12), ("Charles", 12)]`. Write a function `headGrade` that, given such a list, returns the grade of the first person on the list, or fails with `None`:

```
def headGrade(results: List[(String,Int)]): Option[Int]
```

Use the helper function `headOption` provided in the exercise Scala file. First, use `map` then rewrite your solution using a `for-yield` comprehension. Do not use pattern matching. The easiest way to get the second element from a pair value in Scala, is to call the `_2` method, as in: `pairValue._2` .

Exercise 8 [H]. A variance computation is something that you could need to implement in a data analytics application. If the mean of a sequence x_i is μ, then the variance is the mean of the sequence $(x_i - \mu)^2$ (the sequence of squared deviations from the mean).[6]

Implement the `variance` function for a list in terms of (using) `flatMap`. Do not use pattern matching or `map` on `Option`. You should also be able to avoid `isEmpty`. This is likely your first experience of a computation in a monad.

Once it works with `flatMap`, rewrite it with `for-yield` comprehensions without using `flatMap`. Reflect on the differences between the two implementations. Which one you find easier to understand?

Exercise 9 [M]. Section 4.3.2 in the text book recasts the `map` function as a general lifter that can change any function of type `A =>B` into a function of type `Option[A] =>Option[B]`. This only works for unary functions. This exercise tries to achieve a similar effect for binary functions. Write a generic function `map2` that combines two `Option` values using a binary function:

```
def map2[A, B, C](ao: Option[A], bo: Option[B])(f: (A,B) =>C): Option[C]
```

[4] Exercise 3.28 [Pilquist, Chiusano, Bjarnason, 2023]
[5] Exercise 4.1 [Pilquist, Chiusano, Bjarnason, 2023]
[6] Exercise 4.2 [Pilquist, Chiusano, Bjarnason, 2023]

If either `Option` value is `None`, then the return value is `None`, too. Do not use pattern matching.Use `map`/`flatMap` or `for-yield` comprehensions.[7]

Besides being a lifter of binary functions to the `Option` universe, the `map2` function can also be seen as a sequencer: it combines (sequences) the result of a fallible execution producing `ao` with the result of a fallible execution producing `ab`. The combination is done using `f` if both results are successful.

Exercise 10 [H]. We want to generalize `map2` from two to arbitrary many values. Imagine that we have a list of employees and we had retrieved salary rate for each of them. If the database connection failed for an employee, we received a `None` otherwise a `Some` object carrying the value. Thus we get a list of options. We would like to abort the entire computation and return `None` if at least one value on the list is `None` — if the database failed at least once we should report to the caller that there was an error and stop. To do this, we need sequence not a pair of results but an entire list of results.

Write a function `sequence` that combines a list of `Options` 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.

NB. This is an example, where it seems inappropriate to define the function as a method in the 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` or as a top-level function in the option package.[8]

Exercise 11 [H]. Implement a 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 (the traversal) fails. The `traverse` function does not increase the power of what we can do, but it can be implemented more efficiently than using `map` and `sequence` in combination. This is because `map`, being a polymorphic structure-preserving function oblivious to the types of transformed values, will not see by itself that it should stop on the first `None`. Think which lower-level combinator you can use, if map is too simple here.[9]

---

[7] Exercise 4.3 [Pilquist, Chiusano, Bjarnason, 2023]
[8] Exercise 4.4 [Pilquist, Chiusano, Bjarnason, 2023]
[9] Exercise 4.5 [Pilquist, Chiusano, Bjarnason, 2023]