Exercise sheet 03— solutions

1 Standard deviation

```
def standardDeviation(l: List[Double]): Option[Double] =
  mean(l)
    .flatMap(m => mean(l.map(x => math.pow(x - m, 2))))
    .map(math.sqrt)
```

First `mean(l)` returns an `Option[Double]`, which is either `None` (if the input list is empty) or `Some(m)`, with `m` being the mean value of the passed list `l`.

If the list was empty, we want our result to stay `None`, otherwise we want to calculate the squared difference between each single value and the mean we just calculated, and then calculate the mean of those differences. As the latter returns an `Option` again, we need to use `flatMap` on our first `Option` instead of `map`, otherwise we would get nested `Option`s.

The parameter `m` in the function we give to `flatMap` contains the calculated mean. The call `l.map(x => math.pow(x - m, 2))` creates a new list by subtracting `m` from every element and squaring the result. We now calculate the `mean` on this new list. The result again is an `Option[Double]` and is also called the variance of the list (in the mathematical sense, not to be confused with variance of types). We now have to calculate the square root. As `math.sqrt` always returns a Double, `map` suffices.

The whole thing becomes more clearly readable, if we use a for comprehension:

```
def standardDeviationFor(l: List[Double]): Option[Double] = for
  m <- mean(l)
  v <- mean(l.map(x => Math.pow(x - m, 2)))
yield math.sqrt(v)
```

Like above, the result of `mean(l)` is assigned to the variable `m` and the second line calculates the squared difference. The return value of the second `mean` is stored in `v`. The `yield` expression calculates the square root of `v` as the result of the for-comprehension. Here `m` and `v` are the unboxed values in `Some`, if a `None` occurs anywhere, the whole for comprehension returns `None`.

2 sequence and traverse for Option

In the lecture the function `map2` was only shown for `Either`. It is defined pretty similarly for `Option` [1]:

```
def map2[B,R](optB: Option[B])(f: (A,B) => R): Option[R] =
  for
    a <- this
    b <- optB
  yield f(a, b)
```

`map2` combines two `Option`s with types `A` and `B` using a function `f`, resulting in a new `Option` with type `R`. If at least one `Option` is `None`, the result is `None` too. For example, `Some(2).map2(Some(3))(_+_)` results in `Some(5)`.

[1] slightly different in the template, as the standard library doesn't define `map2` on `Option`

2.1 sequence mit foldRight und map2

The basic idea when implementing `sequence` via `foldRight` is to walk through the list of `Option`s, unpacking each of the `Option`s and combine the values into a new list. The final result should be wrapped in a `Some`, or be `None` if any of the `Option`s in the list was `None`.

```
def sequenceViaFold[A](l: List[Option[A]]): Option[List[A]] =
  l.foldRight[Option[List[A]]](Some(Nil))(_.map2(_)(_ :: _))
```

`foldRight`'s type parameter states, which return type we want. If we don't specify it, the compiler will infer it from the first parameter, which is `Some(Nil)` here, so it would be inferred to `Option[List[Nothing]]`. To have the correct type, we specify the desired `Option[List[A]]` explicitly.

`foldRight`'s second parameter is a function accepting a value of the element type of our list and one of our return type, i.e. a `Option[A]` and a `Option[List[A]]`. We use `map2` to combine those two `Option`s into one. `map2` itself requires a function which takes two parameters, namely the values inside the combined options. They have the types `A` and `List[A]`. We want a `List[A]` inside an `Option` as a result, so we prepend the `A` to the list using `::`.

This solution uses underscore syntax for functions, which can look very confusing here, as we have two nested anonymous functions. Let's break down the expression `_.map2(_)(_ :: _)`:

- Our foldRight expects a function with two parameters. So two of the underscores must be the ones from that function.
- The map2 function itself expects a function in its second parantheses, so the Scala compiler will treat the underscores therein as parameters of the function passed to map2. The underscores can only refer to the directly enclosing scope, so that with such nesting the code does not become ambiguous.
- Therefore we have a function `_ :: _`, which we pass to `map2`. We'll call it `f`.
- Now we pass `_.map2(_)(f)` to `foldRight`
- Without using underscores and with all types explicitly given, the expression would look like this:

  ```
  (optA: Option[A], optListA: Option[List[A]]) =>
    optA.map2(optListA)( (a: A, listA: List[A]) => a :: listA)
  ```

Here is a trace of a call of this function, converting `List(Some(2), Some(3))` into `Some(List(2, 3))`:

```
val l = List(Some(2), Some(3))
val f: (Option[Int], Option[List[Int]]) => Option[List[Int]] = _.map2(_)(_ :: _)
l.foldRight(Some(Nil))(f)
Some(2).map2(List(Some(3)).foldRight(Some(Nil))(f))(_ :: _)
Some(2).map2(Some(3).map2(Nil.foldRight(Some(Nil))(f))(_ :: _))(_ :: _)
Some(2).map2(Some(3).map2(Some(Nil))(_ :: _))(_ :: _)
Some(2).map2(Some(3 :: Nil))(_ :: _)
Some(2 :: (3 :: Nil))
Some(List(2, 3))
```

We left out specifying [Option[List[Int]]] for every foldRight call here for clarity. The trace wouln't compile as shown.

2.2 traverse using pattern matching and map2

Traverse is given a simple List[A], where the contained elements are made into Options via the function f. But as a result we want a single Option with the processed list in it.

```
def traverseViaMap2[A,B](l: List[A])(f: A => Option[B]): Option[List[B]] = l match
  case Nil => Some(Nil)
  case a :: as => f(a).map2(traverseViaMap2(as)(f))(_ :: _)
```

As usual we recurse through the elements of our list. If we arrive at the end, we return Some(Nil) as the end of the list (if we returned None, the result of traverse would always be None).

For each element we first call f. Then we use map2 on the Option resulting from that call. The second Option passed to map2 the result of the recursive call to traverseViaMap2, which goes through the rest of the list. As traverseViaMap2 returns a Option[List[A]], we can pass the prepend function :: to map2, as we did with sequence.

2.3 traverse using foldRight and map2

```
def traverseViaFold[A,B](l: List[A])(f: A => Option[B]): Option[List[B]] =
  l.foldRight[Option[List[B]]](Some(Nil))((h, t) => f(h).map2(t)(_ :: _))
```

The difference to sequenceViaFoldRight from exercise 2a) is just the added call to f inside the function passed to foldRight, to convert the elements to Options.

2.4 sequence via traverse

Now that we know from the previous exercise, that traverse and sequence only differ in the additional function that is called, we can implement sequence via traverse by simply letting this function do nothing, it just returns its input unchanged (i.e. the identity function):

```
def sequenceViaTraverse[A](l: List[Option[A]]): Option[List[A]] =
  traverseViaFold(l)(x => x)
```

As sequence is already taking a list of Options, the identity function fulfills the required signature A => Option[B] (here A == Option[B]).

3 sequence and traverse for Either

3.1 sequence, then traverse

In the lecture we already saw the following implementation of `sequence`:

```
def sequence_lecture[E,A](l: List[Either[E,A]]): Either[E,List[A]] =
  l match
    case Nil => Right(Nil)
    case h::t =>
      for
        hh <- h
        tt <- sequence_lecture(t)
      yield hh :: tt
```

We can implement `sequence` for `Either` analogous to the `foldRight`-based implementation for `Option`:

```
def sequence[E,A](l: List[Either[E,A]]): Either[E,List[A]] =
  l.foldRight[Either[E,List[A]]](Right(Nil))(_.map2(_)(_ :: _))
```

Based on this we now want to implement `traverse`. We've seen with `Option`, that the difference is the additional passed function in `traverse`. As that function is called on each element before anything else is done to it, we can do that for the whole list in advance. So we use `map` to apply `f` to the whole list, resulting in a `List[Either[E,B]]`, which we can pass to `sequence`.

```
def traverseViaSequence[E,A,B](l: List[A])(f: A => Either[E,B]): Either[E,List[B]] =
  sequence(l.map(f))
```

3.2 traverse, then sequence

On `Either` we can implement `traverse` with `foldRight` too. The principle is the same as in the implementation for `Option`.

```
def traverse[E,A,B](l:List[A])(f: A => Either[E,B]): Either[E,List[B]] =
  l.foldRight[Either[E,List[B]]](Right(Nil))((h, t) => f(h).map2(t)(_ :: _))
```

In case of `sequence` via `traverse`, the implementation is the same as for `Option`, only the signature differs:

```
def sequenceViaTraverse[E,A](l: List[Either[E,A]]): Either[E,List[A]] =
  traverse(l)(x => x)
```

4 Accumulating errors

4.1 How could the datatype `Either` be modified, to allow `map2` to return all errors?

Modifying `Either` to allow `map2` to return all errors instead of the first only, could be done by using `Either[List[E], ]` to combine errors in a list. But this would also need different implementations for functions like `map2` and `sequence`.

Another approach would be a new datatype, which allows to keep a list of errors in the constructor that represents errors:

```
enum Validated[+E,+A]:
  case Errors(get: List[E])
  case Success(get: A)
```

For this datatype we coud then implement `map`, `map2`, `sequence` etc. (but not `flatMap`, see next exercise part) in a way that they accumulate errors.

We will look at **`Validated`** in more detail in a later chapter of the lecture.

4.2 Why can `flatMap` never collect errors?

For this question look at the signature of `flatMap` (here using the variant with fixed error type, but valid otherwise too):

```
enum Either[E, +A]:
  case Left(error: E)
  case Right(value: A)

  def flatMap[B](f: (A => Either[E, B])): Either[E,B]
```

To accumulate an error returned by the function `f`, we'd need at least one previous error to accumulate with. But if we had an error before, that means the `Either` we called `flatMap` on, is a `Left` containing an `E`. To even be able to call `f` so it can generate an error, we would need an `A`, as `f` has the type `A => Either[E, B]`. Such an `A` isn't available, if we already had an error, `Left` only contains an `E`.

This is a nice example of parametricity. We can deduct that `flatMap` stops at the first error just from the types. If on the other hand `f` had a type like e.g. `Int => Either[E, B]`, we couldn't rule it out anymore. As the function would expect a concrete type, `flatMap` could call it with some fixed value of that type e.g. 0, if only a `Left` was available. But because of `f: A => Either[E, B]` not containing concrete types beside `Either`, we already know a lot about the implementation.