Exercise sheet for lecture 03

In this exercise we deal with handling errors without exceptions, using `Option` and `Either`. We will use the implementation of those from the standard library. You can find the signatures of the methods you should implement, as well as some given implementations, in the git repository at https://gitlab2.informatik.uni-wuerzburg.de/intro-to-fp/tasksheets.

1 Standard deviation

Using `flatMap` we can create algorithms, whose calculations have several sections that could each fail. The calculation stops, as soon as the first error occurs, because `None.flatmap(f)` immediately returns `None` without calling `f`.

Implement the function `standardDeviation` using `flatMap`!

Let the `mean` of a sequence of numbers be m. Then the standard deviation is the square root of the `mean` of `math.pow(x-m, 2)` for every `x` in the sequence. Use the `mean` function from the lecture, which returns an `Option[Double]`. You can use `math.sqrt` to calculate the square root.

```
def standardDeviation(xs: List[Double]): Option[Double] = ???
```

2 sequence and traverse for Option

In this exercise you will implement the functions `sequence` and `traverse`, which were shown in the lecture, in several different ways.

The goal is to practice using folds and maps and to see how different functions can be implemented "'in terms of each other"'. Here are once again the signatures of `sequence` and `traverse` for `Option`

```
def sequence[A](a: List[Option[A]]): Option[List[A]] = ???

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

a) Implement `sequence` using `foldRight` and `map2`!

b) Implement `traverse` using explicit pattern matching and `map2` without using `sequence`!

c) Implement `traverse` using `foldRight` and `map2`!

d) Implement `sequence` using `traverse`!

The function `map2` combines two `Option` objects (or similar) into one object[1]:

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

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

3 sequence and traverse for Either

In this exercise you have to implement `sequence` and `traverse` for `Either`. The functions don't differ much from the ones you know from `Option`.

```
def sequence[E, A](es: List[Either[E, A]]): Either[E, List[A]] = ???

def traverse[E, A, B](as: List[A])(f: A => Either[E, B]): Either[E, List[B]] = ???
```

a) Implement `sequence` first, then `traverse` using `sequence` as seen with `Option` in the lecture!

b) Now implement `traverse` first and then `sequence` using `traverse` as in exercise 2!

4 Accumulating errors

The following example shows an application of `map2`, in which the function `mkPerson` checks the passed name as well as the age, before creating a valid `Person`.

```
case class Person(name: Name, age: Age)
case class Name(value: String)
case class Age(value: Int)

import Either.{Left, Right}

def mkName(name: String): Either[String, Name] =
  if name == "" then Left("Name is empty.")
  else Right(Name(name))

def mkAge(age: Int): Either[String, Age] =
  if age < 0 then Left("Age is out of range.")
  else Right(Age(age))

def mkPerson(name: String, age: Int): Either[String, Person] =
  mkName(name).map2(mkAge(age))(Person(_, _))
```

a) In this implementation, `map2` can only return one error. How could the datatype `Either` be modified, to allow `map2` to return all errors?

b) Why can `flatMap` never collect errors (and thus no implementation of `map2` based on `flatMap`)?