Übungsblatt zu Vorlesung 03

In dieser Übung beschäftigen wir uns mit Fehlerbehandlung ohne Exceptions mit `Option` und `Either`. Für die Aufgaben verwenden wir die Implementierungen aus der Scala-Standardbibliothek. Die Signaturen der in den Aufgaben geforderten Methoden sowie die vorgegebenen Implementierungen finden Sie auch online unter https://gitlab2.informatik.uni-wuerzburg.de/intro-to-fp/tasksheets als Git-Repository.

1 Standardabweichung

Mit `flatMap` können Algorithmen erstellt werden, deren Berechnung mehrere Abschnitte durchläuft von denen jeder fehlschlagen könnte. Die Berechnung bricht ab, sobald der erste Fehler auftritt, denn `None.flatmap(f)` gibt sofort `None` zurück, ohne `f` aufzurufen.

Implementieren Sie die Funktion `standardDeviation` mittels `flatMap`!

Wenn der `mean` einer Folge von Zahlen m ist, ist die Standardabweichung die Wurzel vom `mean` von `math.pow(x-m, 2)` für jedes Element `x` in der Folge. Benutzen Sie die `mean` Funktion aus der Vorlesung, die eine `Option[Double]` zurückgibt. Zur Berechnung der Wurzel können sie `math.sqrt` nutzen.

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

2 sequence und traverse für Option

In dieser Aufgabe werden Sie die in der Vorlesung vorgestellten Funktionen `sequence` und `traverse` auf verschiedene Arten implementieren.

Das Ziel ist es, etwas Übung bei der Verwendung von folds und maps zu bekommen und zu sehen wie verschiedene Funktionen „*in terms of each other*“ implementiert werden können. Gegeben sind hier noch einmal die Signaturen der Funktionen `sequence` und `traverse` für `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) Implementieren Sie `sequence` mit `foldRight` und `map2`!

b) Implementieren Sie `traverse` mit explizitem Pattern Matching und `map2` ohne `sequence` zu verwenden!

c) Implementieren Sie `traverse` mit `foldRight` und `map2`!

d) Implementieren Sie `sequence` via `traverse`!

Die Funktion `map2` verbindet zwei `Option`-Objekte (oder vergleichbares) zu einem Objekt[1].

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

[1] Im Template etwas anders, weil `map2` in der Standardlibrary nicht auf `Option` definiert ist

3 sequence und traverse für Either

In dieser Aufgabe sind `sequence` und `traverse` für `Either` zu implementieren. Die Funktionen unterscheiden sich nicht stark zu denen, die Sie bereits von `Option` kennen.

```
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) Implementieren Sie zunächst `sequence` und anschließend `traverse` via `sequence` wie in der Vorlesung bei `Option` gesehen!

b) Implementieren Sie nun wie in Aufgabe 2 zunächst `traverse` und anschließend `sequence` via `traverse`!

4 Akkumulieren von Fehlern

Das folgende Beispiel zeigt eine Anwendung von `map2`, in der die Funktion `mkPerson` sowohl den übergebenen Namen als auch das Alter überprüft bevor eine valide `Person` erstellt wird.

```
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 dieser Implementierung kann `map2` nur einen Fehler zurückliefern. Wie könnte man den Datentyp `Either` verändern, damit `map2` alle Fehler zurückliefern kann?

b) Warum kann `flatMap` prinzipiell keine Fehler sammeln (und folglich auch keine auf `flatMap` basierende Implementation von `map2`)?