Paradigmen der Programmierung
Praktikum 2 - Funktionale Programmierung

In dieser Aufgabe sollen Sie ein paar Aspekte der Funktionalen Programmierung anhand von Kotlin kennenlernen. Kotlin ist keine funktionale Programmiersprache. Allerdings reicht Kotlin aus, um die Kernaspekte zu verstehen und zu üben.

Empfohlene Literatur:

- Alle Screencasts zur Funktionalen Programmierung, bis auf Invarianz, Kovarianz und Kontravarianz Teil 1 und 2 (FP_15 und 16).
- Kapitel 12.10 "Die Ideen hinter Funktionaler Programmierung" bis inklusive 12.13 "Funktionen höherer Ordnung" und Kapitel 25 "Funktionen höherer Ordnung für Datensammlungen" aus dem Buch Programmieren lernen mit Kotlin.
- Kapitel 5 "Funktionale Programmierung" im Skript von Prof. Dr. Erich Ehses.
  Achtung: Das Skript lehrt Funktionale Programmierung am Beispiel der Programmiersprache Scala. Allerdings sind die Konzepte immer gleich und dort sehr gut erklärt.
- Funktionen höherer Ordnung und Lambdas sind auch in der Dokumentation von Kotlin gut erklärt.

In diesem Übungsblatt programmieren Sie mit Kotlin. Erledigen Sie alle Programmieraufgaben in IntelliJ.

Inhaltsverzeichnis

1 Von objektorientiert zu funktional

Im ersten Vorlesungsblock haben wir das `Ordering` Beispiel mit Entwurfsmustern objektorientiert implementiert. In dieser Aufgabe soll dieses Beispiel mit funktionalen Prinzipien vollständig neu implementiert werden. Es sollen beispielsweise ausschließlich Funktionen und Werte und keine Klassen und Interfaces verwendet werden.

Als Orientierung können Sie den objektorientierten Code aus der Vorlesung verwenden: `https://gist.github.com/alexdobry/57a4e88f975c8dd46926668ca16e2856`.

1.1 Ordering als Funktionstyp und primitive Orderings

Als erstes soll ein `Ordering` als Funktionstyp ausgedrückt werden. Ein `Ordering` ist eine Funktion, die zwei Werte vom Typ `A` entgegennimmt und ein `OrderResult` zurückgibt. Also eine Funktion mit dem Typ `(A, A) -> OrderResult`. In Kotlin können wir einen Funktionstypen mit dem Schlüsselwort `typealias` einen Namen geben. So können wir ein Ordering als Funktion ausdrücken:

```
typealias Ordering<A> = (A, A) -> OrderResult
```

Nun können wir ein primitives `Int` Ordering definieren, indem wir eine Funktion erzeugen, die vom Typ `Ordering<Int>` ist:

```
val intOrd: Ordering<Int> = { left, right ->
  if (left < right) OrderResult.Lower
  else if (left > right) OrderResult.Higher
  else OrderResult.Equal
}
```

Der Typ von `intOrd` ist zwar `Ordering<Int>`, allerdings ist `Ordering<Int>` nur ein Alias für den Funktionstypen `(Int, Int)-> OrderResult`.

a) Implementieren Sie weitere primitive Orderings vom Typ `Ordering<String>`, `Ordering<Double>` und `Ordering<Boolean>`.

1.2 Funktionen höherer Ordnung auf Orderings

a) Da wir nun primitive Orderings haben, sollen Sie Funktionen mit Orderings definieren:

- `reversed`: Die Funktion `reversed` nimmt ein `Ordering` entgegen und gibt ein neues `Ordering` zurück, in dem das `OrderResult` getauscht wurde (vgl. `ReversedOrdering`).
- `debug`: Die Funktion `debug` nimmt ebenfalls ein `Ordering` entgegen und gibt ein neues `Ordering` zurück. In der Implementierung werden die zwei zu vergleichenden Elemente und das Ergebnis dessen ausgegeben (vgl. `DebugOrdering`).

Die Verwendung kann beispielsweise so aussehen:

```
fun main() {
  val intDesc = reversed(intOrd)
  val string = debug(stringOrd)
  val doubleDesc = debug(reversed(doubleOrd))
}
```

b) Beantworten Sie zudem folgende Fragen:

- Warum sind `reversed` und `debug` Funktionen höherer Ordnung?
- Welches Entwurfsmuster wurde durch die Verwendung von Funktionen höherer Ordnung realisiert?
- Warum kann das Entwurfsmuster dadurch implementiert werden? Was ist die Grundlegende Struktur des Entwurfsmusters und inwiefern korreliert diese Struktur mit der von Funktionen höherer Ordnung?

1.3 Orderings mappen

Bislang haben wir Orderings auf primitive Typen wie `Int`, `String`, `Double` und `Boolean` definiert. Nun wollen wir neue Orderings für *eigene Typen*, wie z.B. Personen, definieren. Allerdings soll dies auf **Grundlage bestehender Orderings** passieren.

Das Ziel ist es, eine Funktion zu finden, die ein `Ordering` vom Typ `A` auf ein `Ordering` vom Typ `B` überführt. Dieses Muster kennen wir als `map` Funktion, beispielsweise um von einer Liste von `A` zu einer Liste von `B` zu mappen, indem man für jedes Element die Funktion `A -> B` anwendet:

```
fun <A, B> map(list: List<A>, transform: (A) -> B): List<B> {
  val bs = mutableListOf<B>()
  for (a in list) bs += transform(a)
  return bs
}
```

In dieser Aufgabe wollen wir eine ebenfalls eine `map` Funktion für Orderings schreiben, die die gleiche Signatur wie die `map` Funktion für Listen hat, aber mit `Ordering` arbeitet. **Allerdings muss die Struktur der `transform` Funktion umgedreht werden**, nämlich von `B` nach `A`. Da wir die `transform` Funktion umdrehen müssen, nennen wir die gesamte Funktion `contraMap`[1].

a) Definieren Sie die Klasse `Person`, die aus einem Namen und einem Alter besteht.

b) Implementieren Sie die `contraMap` Funktion, die auf Grundlage eines bestehenden `Ordering<A>` mithilfe der `transform` Funktion `(B) -> A` ein neues `Ordering<B>` erzeugt.

c) Verwenden Sie `contraMap`, um folgende `Person Orderings` zu erzeugen:

- Erzeugen Sie einen Wert vom Typ `Ordering<Person>`, welches das `String Ordering` aus Aufgabe 1.1 nutzt, um Personen nach Namen zu vergleichen.
- Erzeugen Sie danach ein weiteres `Person Ordering` für das Alter.

Hier ein Beispiel für die Verwendung von `contraMap`, wo ein *alternatives* `String Ordering` definiert wird, welches Strings nach *ihrer Länge* vergleicht. Da die Länge vom Typ `Int` ist, wird das in Aufgabe 1.1 definierte `Int Ordering` als Grundlage verwendet. Die `transform` Funktion beschreibt die entsprechende Überführung von `String` nach `Int`:

```
fun main() {
  val stringLengthOrd: Ordering<String> = contraMap(
    ord = intOrd,
    transform = { string -> string.length }
  )
}
```

[1] Tatsächlich ist es unmöglich, eine klassische `map` Funktion für Orderings zu definieren, weil der zu mappende Typ in einer **contravarianten Position** (A ist der Parameter) steht. Daher ist nur ein `contraMap` möglich. Das contra bedeutet, dass die transform Funktion geflipt werden muss. Siehe `https://typelevel.org/cats/typeclasses/contravariant.html`

1.4 Orderings zippen

Als nächstes implementieren wir eine Funktion, um zwei Orderings **unterschiedlichen Typs** miteinander zu kombinieren. Auch hier schauen wir uns zuerst das Äquivalent zu Listen an. Wenn wir eine `List<A>` mit einer `List<B>` kombinieren, erhalten wir eine neue Liste, die jeweils ein Paar der nten Elemente beider Listen enthält. Die Operation heißt `zip` und sieht so aus:

```
fun main() {
  val ints: List<Int> = listOf(25, 33, 28)
  val strings: List<String> = listOf("Nathalie", "Alex", "Zah")
  val intsWithStrings: List<Pair<Int, String>> = ints.zip(strings)
}
```

`intsWithStrings` ist die Kombination beider Eingangslisten und hat folgenden Inhalt:

```
[
  Pair(25, "Nathalie"),
  Pair(33, "Alex"),
  Pair(28, "Zah")
]
```

Im Kontext von Orderings bedeutet eine Kombination von zwei Orderings, dass **zuerst nach dem ersten Ordering** verglichen wird. Erst wenn das erste Ordering **gleich** ist, wird **nach dem zweiten Ordering** verglichen (vgl. `CombineOrdering`).

a) Implementieren Sie eine Funktion namens `zip`, die zwei Orderings **unterschiedlichen Typs** entgegennimmt und ein `Ordering<Pair<A, B>>` zurückgibt. Die Implementierung soll der **zuvor genannten Semantik** folgen. Damit können wir ein `Ordering` erzeugen, welches z.B. zuerst nach `String` und danach nach `Int` vergleicht:

```
val ord: Ordering<Pair<String, Int>> = zip(stringOrd, intOrd)
```

Ein `Pair` von `String` und `Int` bringt uns in unserer Anwendung nicht viel. Glücklicherweise haben wir mit `contraMap` die Möglichkeit, um von `Pair<String, Int>` auf `Person` zu mappen.

b) Erweitern Sie den oben gezeigen Code, sodass `ord` ein `Ordering<Person>` ist, welches zuerst nach Namen und danach nach dem Alter vergleicht. So können wir Personen nach mehreren Kriterien vergleichen:

```
fun main() {
  val people = mutableListOf(
    Person("Nathalie", 25),
    Person("Alex", 33),
    Person("Zah", 28),
    Person("Alex", 18),
    Person("Jens", 33),
  )
  val personOrd: Ordering<Person> = TODO("contraMap(zip(...))")
  Sorting().sort(people, personOrd)
```

```
  people.forEach(::println)
}
```

Die Konsolenausgabe sieht folgendermaßen aus:

```
Person(name=Alex, age=18)
Person(name=Alex, age=33)
Person(name=Jens, age=33)
Person(name=Nathalie, age=25)
Person(name=Zah, age=28)
```

1.5 Ergonomie verbessern

a) Schreiben Sie die Funktionen `reversed`, `debug`, `contraMap` und `zip` so um, dass sie als *extension functions* auf dem jeweils ersten Parameter definiert sind. Erweitern Sie zudem die Klasse `Person` um eine weitere Eigenschaft, wie z.B. `height: Double`, um ein weiteres `Ordering` zu verwenden. Der Code sollte jetzt in etwa so aussehen:

```
data class Person(val name: String, val age: Int, val height: Double)

fun main() {
  val people = mutableListOf(
    Person("Nathalie", 25, 172.5),
    Person("Alex", 33, 186.0),
    Person("Zah", 28, 158.3),
    Person("Alex", 18, 183.0),
    Person("Jens", 33, 168.5),
  )
  val personOrd: Ordering<Person> =
    stringOrd
      .zip(intOrd.reversed())
      .zip(doubleOrd)
      .contraMap { person ->
        person.name to person.age to person.height // kürzere
           Schreibweise für Pair(Pair(person.name, person.age),
           person.height)
      }
  Sorting().sort(people, personOrd)
  people.forEach(::println)
}
```

BONUS b) Schreiben Sie die `sort` Methode in `Sorting` so um, dass Sie mit *unveränderlichen Listen* arbeitet. Definieren Sie zudem eine *extension function* auf `List` namens `orderBy`, die ein *Ordering* entgegennimmt und eine entsprechend sortierte Liste zurückgibt. Die Verwendung kann dann so aussehen:

```
fun main() {
  val people: List<Person> = TODO("code von oben")
  val personOrd: Ordering<Person> = TODO("code von oben")

  people
    .orderBy(personOrd)
    .forEach(::println)
}
```

2 Algebraische Datentypen

a) Welcher Typ in Kotlin ist äquivalent zur `1` in der Algebra?

b) Zeigen Sie, ob `Either<Option<A>, B>` äquivalent bzw. isomorph zu `Option<Either<A, B>>` ist. Überführen Sie dazu die Typen in Algebra. Hinweis: `Option` ist der nullfähige Typ in Kotlin.

c) Überführen Sie das Potenzgesetz $a^0 = 1$ in Typen. Implementieren Sie auch die jeweilige `to`- und `from`-Funktion. Die Funktionen sind:

```
fun <A> to(f: (Nothing) -> A): Unit = TODO()
fun <A> from(unit: Unit): (Nothing) -> A = TODO()
```

d) Warum kann die `from`-Funktion implementiert werden, obwohl nur ein `Nothing` zur Verfügung steht, aber ein Wert vom Typ `A` zurückgegeben werden muss? Hinweis: Die Antwort liegt im Subtyping-System von Kotlin.

e) Gegeben sei der Typ `Either<A, B>` und die Funktion `makeEither`, die aus einem `A` und `B` ein `Either<A, B>` erzeugt:

```
sealed interface Either<out A, out B>
data class Left<A>(val a: A): Either<A, Nothing>
data class Right<B>(val b: B): Either<Nothing, B>

fun <A, B> makeEither(a: A, b: B): Either<A, B> = TODO()
```

Implementieren Sie die `makeEither` Funktion, sodass der Code kompiliert. Rufen Sie die Funktion mit ihrem Namen als erstes Argument und mit ihrem Alter als zweites Argument auf.

f) Warum ist die Implementierung von `makeEither` nicht 100 %ig valide? Begründe Sie ihre Antwort entweder mit der Verwendung von Algebra oder durch logische Argumente.