Übungsblatt zu Vorlesung 02

Im Git-Repository (https://gitlab2.informatik.uni-wuerzburg.de/intro-to-fp/tasksheets) finden Sie Vorlagen für die Übungsblätter. Diese enthalten u.a. bereits in der Vorlesung behandelte Teile des Codes (z.B. den List-Typ), sowie die in den Übungsblättern vorgegebenen Signaturen. Enthalten ist auch eine `build.sbt` mit der entsprechenden Compilerkonfiguration. Mittels dieser kann das Repository auch in einer IDE oder einem LSP-Editor als Projekt importiert werden.

In der Vorlesung wurde die funktionale Datenstruktur `List` vorgestellt. Die Implementation finden Sie in der Datei `fp02/List.scala` des Projekts.

1 dropWhile

In der Vorlesung wurde eine Funktion `drop` behandelt, welche die ersten n Elemente der Liste entfernt.

Implementieren Sie die Funktion `dropWhile`, die so lange Elemente vom Beginn der Liste entfernt wie die übergebene boolsche Funktion für diese `true` zurückgibt. Benutzen Sie Pattern Matching und Rekursion.

```
def dropWhile(f: A => Boolean): List[A] = ???
```

2 Folds — Schritt für Schritt

In der Vorlesung wurde die Ausführung eines `foldRight` Aufrufs Schritt für Schritt nachverfolgt.

a) Wie sieht dieser *trace* aus wenn stattdessen `foldLeft` benutzt wird?

```
Cons(1, Cons(2, Cons(3, Nil))).foldLeft(0)((x, y) => x + y)
```

b) Der folgende `foldRight` Aufruf inkrementiert jedes Element der Liste `l` um 1 und erstellt daraus eine neue Liste. Verfolgen Sie die Ausführung dieses Aufrufs Schritt für Schritt.

```
val l = Cons(1, Cons(2, Nil))
l.foldRight(Nil: List[Int])((h, t) => Cons(h + 1, t))
```

3 reverse

Implementieren Sie die Funktion `reverse`, die die Liste in umgekehrter Reihenfolge zurückgibt. Benutzen Sie dazu `foldLeft`. Überlegen Sie, was als initialer *Accumulator* an `foldLeft` übergeben werden muss, um eine Liste aufzubauen.

```
def reverse: List[A] = ???
```

4 append

Implementieren Sie die Funktion `append`, welche diese Liste mit einer übergebenen konkateniert, d.h. die übergebene an diese anhängt, indem Sie `foldRight` verwenden.

```
def append[AA >: A](r: List[AA]): List[AA] = ???
```

Eine Implementation mittels Pattern Matching und manueller Rekursion ist im Template zu finden.

5 map, filter und flatMap

map

Implementieren Sie die Funktion `map` mittels `foldRight`. Diese modifiziert jedes Element der Liste und behält dabei die Struktur der Liste bei. In der Vorlesung wurden zwei Anwendungsbeispiele dieser Funktion gezeigt.

```
def map[B](f: A => B): List[B] = ???

//Example 1: double all values
List(1,2,3).map(_ * 2) == List(2,4,6)

//Example 2: result type can be different
List(1,2,3).map(_.toString) == List("1", "2", "3")
```

Hinweis: Einen Spezialfall der Implementierung von `map` haben Sie auf diesem Übungsblatt bereits gesehen.

filter

Implementieren Sie die in der Vorlesung vorgestellte Funktion `filter`. Diese gibt eine neue Liste zurück, die nur die Elemente enthält, für die eine gegebene boolsche Funktion wahr ist.

```
def filter(p: A => Boolean): List[A] = ???

//Example: only keep odd numbers
List(1,2,3,4,5,6).filter(_ % 2 == 1) == List(1,3,5)
```

flatMap

Die Funktion `flatMap` ist ähnlich zu `map`, nur dass die übergebene Funktion eine Liste von Elementen zurückgibt. Diese zurückgegebenen Listen werden anschließend zu einer Liste konkateniert.

```
def flatMap[B](f: A => List[B]): List[B] = ???

//Example: for each number, add it multiplied by 10 to the list
List(1,2,3).flatMap(i => List(i, i * 10)) == List(1, 10, 2, 20, 3, 30)
```

Benutzen Sie explizit `foldRight` und `append` um `flatMap` zu implementieren.

6 zip

Die Funktionen der `zip`-Familie verbinden auf verschiedene Arten mehrere Listen miteinander.

a) Implementieren Sie eine Funktion `zipAdd`, die zwei Listen mit Integern übergeben bekommt und eine neue Liste konstruiert indem sie „gegenüberliegende“ Elemente addiert. Zum Beispiel entsteht aus `zipAdd(List(1,2,3), List(4,5,6))` die neue Liste `List(5,7,9)`

   Implementieren Sie diese Funktion auf dem Companion Object!

b) Verallgemeinern Sie die Funktion `zipAdd` zu `zipWith`, sodass sie nicht mehr nur auf Integer und Addition beschränkt ist, sondern die Operation angegeben werden kann.

   Implementieren Sie diese Funktion direkt im `List`-Enum. (Achtung: nicht ganz einfach!)

Überlegen Sie sich jeweils selbstständig, wie die passenden Methodensignaturen aussehen müssen.

Sind die beiden Listen nicht gleich lang, soll nach dem Ende der kürzeren abgebrochen werden, sodass die resultierende Liste immer die Länge der kürzeren hat.

Hinweis: Benutzen Sie Pattern Matching und Rekursion. Für Pattern Matching mit mehreren Elementen kann man diese z.B. in einem Tupel `(a,b)` kombinieren und dann auf diesem Tupel matchen. Beispiel mit Integern statt Listen:

```scala
val a = 2
val b = 4
(a,b) match
  case (x,y) => /* x == 2, y == 4 */
```

7 Parametricity

Implementieren Sie die folgenden Funktionen in der Datei `Parametricity.scala`

Implementieren Sie die Funktionen `curry` und `uncurry`, die Funktionen mit zwei Parametern in verschachtelte Funktionen mit je einem Parameter umwandeln (und umgekehrt).

```scala
def curry[A,B,C](f: (A,B) => C): A => (B => C) = ???

def uncurry[A,B,C](f: A => (B => C)): (A, B) => C = ???
```

Hinweis: Durch die generische Definition und die Wahl der Typparameter ist die Implementierung quasi vorgegeben.