PFP-Team
pfp@i2.cs.fau.de

Übungsblatt 11

Abgabe bis 17.07.2019, 18:00 Uhr

Hinweise:

- Verwenden Sie zur Lösung der Aufgaben nur die aus der Vorlesung bekannten, sowie die in den Übungen bekannt gegebenen Methoden und Funktionen der *Scala*-Standardbibliothek.
- Achten Sie bei der Bearbeitung der *Scala*-Aufgaben darauf, dass alle von Ihnen erstellten Methoden und Funktionen keine Seiteneffekte haben dürfen. Diese Regelung gilt sowohl für die Übungsaufgaben (Blätter 10–12) als auch für die Klausur!
- Nutzen Sie nur unveränderliche (immutable) Collection-Klassen. Die Verwendung von Klassen aus dem Paket `scala.collection.mutable` ist untersagt.
- Verwenden Sie nur das Scala-Schlüsselwort `val` zur Deklaration von Attributen und Konstanten. Die Verwendung von `var` ist untersagt.
- Geben Sie Ihre Lösungen für die Blätter 10–12 über die EST-Veranstaltung *PFP (F)* ab.

Aufgabe 11.1: Seiteneffekte

Welche der folgenden *Java*-Methoden hat Seiteneffekte? Welche ist seiteneffektfrei?

```
public class SeiteneffekteJava {

    private int a = 3;
    public static int b = 5;

    public void meth1(int x) {
        a = x;
    }
    public int meth2() {
        return a++;
    }
    public int meth3(int y) {
        return a + b + y;
    }
    public int meth4(int k) {
        b = b + 19 / b;
        return 15;
    }
    public int meth5(int k) {
        int a = ++k;
        return a;
    }
    public void meth6() {
        System.out.println("Hallo");
    }
}
```

Aufgabe 11.2: Programmierparadigmen

a) Welche zwei grundlegenden Programmierparadigmen gibt es und worin unterscheiden sie sich?

b) Nennen Sie die Ihnen bekannten Ausprägungen innerhalb dieser Paradigmen und ordnen Sie diesen die Ihnen bekannten Programmiersprachen zu.

c) Welchem Paradigma und welcher Ausprägung lässt sich *Scala* zuordnen?

Aufgabe 11.3: Currying und Funktionen höherer Ordnung

a) Was ist eine Funktion höherer Ordnung?

b) Was ist eine anonyme Funktion?

c) Was ist der Unterschied zwischen einer 1-stelligen (bzw. 1-wertigen) und einer n-stelligen Funktion?

d) Was versteht man unter dem Begriff „Currying“?

e) Inwiefern unterscheiden sich die Signaturen der Funktionen `f` und `g`, die beide zwei Zahlen miteinander addieren?

```
def f: (Int, Int) => Int = (x, y) => x + y
def g: Int => Int => Int = x => y => x + y
```

Implementieren Sie in der Datei „Currying.scala“ im `object Currying` folgende Funktionen:

f) `curry: ((Int, Int) => Int) => (Int => Int => Int)`: Wandelt eine 1-stellige Funktion `f(a, b)` in eine 2-stellige Funktion um: `g(a)(b)`.

g) `uncurry: (Int => Int => Int) => ((Int, Int) => Int)`: Wandelt eine 2-stellige Funktion `g(a)(b)` in eine 1-stellige Funktion um: `f(a, b)`.

h) Gegeben sind folgende zwei Richtungsfunktionen, die bestimmen, ob zwei Werte in auf- oder absteigender Reihenfolge übergeben wurden:

```
def isAscending: (Int, Int) => Boolean = (a, b) => a <= b
def isDescending: (Int, Int) => Boolean = (a, b) => a > b
```

Diese Funktionen können nun beispielsweise dafür benutzt werden, eine Liste mithilfe der Methode `sortWith` der Klasse `List` zu sortieren: `List(3, 6, 2).sortWith(isAscending)`.

- Deklarieren Sie die Signatur einer 2-stelligen Funktion `curriedSort`. Die 1. Parameterliste soll die Richtungsfunktion enthalten, die 2. Parameterliste eine zu sortierende Liste. Als Ergebnis liefert die Funktion die sortierte Liste.
- Implementieren Sie die Funktion `curriedSort`.

- Definieren Sie zwei Funktionen `sortAscending` und `sortDescending`, die eine Liste vom Typ `List[Int]` entgegennehmen und diese sortieren. Die Implementierung soll auf die Funktionen `curriedSort`, `isAscending` und `isDescending` unter Ausnutzung von Currying zurückgreifen.

Aufgabe 11.4: Listengeneratoren

Implementieren Sie die nachfolgend genannten Funktionen in der Datei „ListGenerators.scala“ im `object ListGenerators`. Die Implementierung soll dabei unter Verwendung von Listengeneratoren erfolgen.

a) `map: (Int => Int) => List[Int] => List[Int]`: Die Funktion `map(f)(xs)` wendet die Funktion `f` auf jedes Element von `xs` an. Die Ergebnisliste enthält die jeweiligen Ergebnisse von `f`.

Beispiel: `map(x => x * 2)(List(1, 2, 3, 4, 5))` ergibt `List(2, 4, 6, 8, 10)`.

b) `flatMap: (Int => List[Int]) => List[Int] => List[Int]`: Die Funktion `flatMap(f)(xs)` wendet die Funktion `f` auf jedes Element von `xs` an. Die Ergebnislisten von `f` werden jeweils aneinander gehängt („konkateniert“).

Beispiel: `flatMap(x => for(a <- List.range(1, 4)) yield a * x)(List(1, 2, 3, 4, 5))` ergibt `List(1, 2, 3, 2, 4, 6, 3, 6, 9, 4, 8, 12, 5, 10, 15)`.

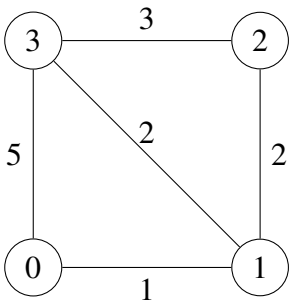
c) `pairSums: (Int, Int) => List[(Int, Int)]`: Die Funktion `pairSums(n, s)` ermittelt die Liste aller Zahlenpaare mit Elementen von 0 bis `n` (inklusive), deren Summe `s` entspricht.

Beispiel: `pairSums(3, 2) == List((0,2), (1,1), (2,0))`

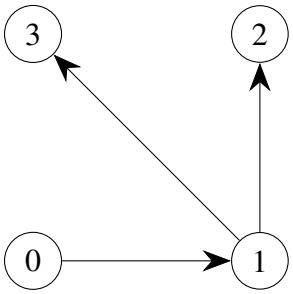
Bonusaufgabe 11.5: Minimaler Spannbaum **4 Punkte**

Implementieren Sie in der Datei „Prim.scala“ im `object Prim` eine Variante von Prims Algorithmus zur Berechnung minimaler Spannbäume in ungerichteten, gewichteten Graphen.

Beispiel:



Graph



Minimaler Spannbaum (ausgehend von 0)

```
val g = List(List((1,1), (3,5)), List((0,1), (2,2), (3,2)),
             List((1,2), (3,3)), List((0,5), (1,2), (2,3)))
```

a) `getPossibleEdges: List[List[(Int, Int)]] => List[Int] => List[(Int, Int, Int)]`: Die Funktion `getPossibleEdges(al)(rs)` erhält mit `al` einen Graphen dargestellt als Adjazenzliste (ein Eintrag besteht aus (Nachfolger, Kosten)) und mit `rs` eine Liste schon verarbeiteter Knoten. Zurückgegeben werden soll eine Liste aller Kanten (als Tupel (Start, Ziel, Kosten)), deren Start bereits verarbeitet und deren Ziel noch nicht verarbeitet wurde. Verwenden Sie zur Lösung Listengeneratoren.

Beispiel: `getPossibleEdges(g)(List(0)) == List((0, 1, 1), (0, 3, 5))`

b) `mst: (List[Int] => List[(Int, Int, Int)]) => Int => List[(Int, Int)]`: Die Funktion `mst(f)(st)` berechnet den minimalen Spannbaum durch Anwendung von Prims Algorithmus ausgehend vom Knoten `st`. Das Argument `f` ist eine Funktion, die eine Liste von bereits verarbeiteten Knoten erhält und eine Liste von möglichen Kanten (mit Kosten) zu noch unverarbeiteten Knoten zurückgibt. Aus diesen möglichen Kanten wird diejenige mit den niedrigsten Kosten ausgewählt und zum Ergebnis hinzugefügt. Sie können dazu die Funktion `minBy` der Klasse `List` verwenden. Der Zielknoten dieser Kante gilt dann als verarbeitet. Das Vorgehen wird solange wiederholt, bis `f` keine Kante mehr zurückgibt.

Beispiel: `mst(getPossibleEdges(g))(0) == List((0, 1), (1, 2), (1, 3))`

Bonusaufgabe 11.6: Listenfunktionen — 2 Punkte

Implementieren Sie die folgenden Listenfunktionen im `object ListFunctions` in der Datei „ListFunctions.scala“:

a) `find[E]: (List[E], E) => List[Int]`: Die Funktion `find(ls, e)` soll die Liste aller Indizes (beginnend bei 0) zurückgeben, an denen das Element `c` in der Liste `ls` vorkommt.

b) `powerSet[E]: List[E] => List[List[E]]`: Die Funktion `powerSet(ls)` soll die Potenzmenge von `ls` als Liste zurückgeben. Die Reihenfolge der Elemente in den Ergebnislisten ist unerheblich.

Hinweis: Versuchen Sie als Vorbereitung auf die Klausur, in obigen Funktionen möglichst viele Methoden der Klasse `List` (z. B. `map`, `flatMap`, `filter`, ...) zu verwenden.

6 Punkte