3. Aufgabenblatt zu Funktionale Programmierung vom Mi, 01.11.2017.
Fällig: Do, 09.11.2017 (15:00 Uhr)

Themen: *Funktionen über neuen Typen, Typklassen, Instanzen und Funktionale*

Zur Frist der Zweitabgabe: Siehe „Hinweise zu Organisation und Ablauf der Übung“ auf der Homepage der LVA.

Aufgabe

Für dieses Aufgabenblatt sollen Sie die zur Lösung der unten angegebenen Aufgabenstellungen zu entwickelnden Haskell-Rechenvorschriften in einer Datei namens `Aufgabe3.lhs` im home-Verzeichnis Ihres Accounts auf der Maschine g0 ablegen. **Anders** als bei der Lösung zu den ersten beiden Aufgabenblättern sollen Sie dieses Mal also ein “**literate Script**” schreiben. Versehen Sie wie auf den bisherigen Aufgabenblättern alle Funktionen, die Sie zur Lösung benötigen, mit ihren Typdeklarationen und kommentieren Sie Ihre Programme aussagekräftig. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

1. Eine positive ganze Zahl ist eine *Tripelprimzahl*, wenn sie das Produkt dreier aufeinanderfolgender Primzahlen ist. $30 = 2 * 3 * 5$ ist eine Tripelprimzahl, $42 = 2 * 3 * 7$ ist keine. (1 ist keine Primzahl; Zahlen wie $6 = 1 * 2 * 3$ sind entsprechend keine Tripelprimzahlen).

   Schreiben Sie eine Haskell-Rechenvorschrift `schuerfen` mit der Signatur `schuerfen :: Zahlenliste -> [Tripelprimzahl]`, die aus einer Liste ganzer Zahlen genau diejenigen herausschürft, die eine Tripelprimzahl sind. Die relative Reihenfolge der Zahlen in Argument- und Ergebnisliste soll dabei übereinstimmen (soweit sie in beiden vorkommen).

   Dabei sollen folgende Typen verwendet werden:

   ```
   type Zahlenliste    = [Integer]
   type Tripelprimzahl = Integer
   schuerfen :: Zahlenliste -> [Tripelprimzahl]
   ```

2. Machen Sie die als neue Typen deklarierten Typen `Kurs` und `Pegelstand` zu Instanzen der Typklasse `Num`, so dass sich die Operationen aus `Num` auf Werten dieser beiden Typen analog zu den entsprechenden Operationen auf Gleitkommawerten vom Typ `Float` verhalten:

   ```
   newtype Kurs       = K Float deriving (Eq,Ord,Show)
   newtype Pegelstand = Pgl Float deriving (Eq,Ord,Show)
   ```

   Als Minimum müssen Sie dafür bei den Instanzbildungen für `Kurs` und `Pegelstand` eine der beiden Funktionen `negate` oder $(-)$ implementieren sowie alle anderen in der Typklasse `Num` vorgesehenen Rechenvorschriften.

3. Schreiben Sie in Analogie zu den Funktionalen `curry`, `uncurry` und `flip` aus der Vorlesung entsprechende Funktionale

   - `curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d`
   - `uncurry3 :: (a -> b -> c -> d) -> (a,b,c) -> d`

   und

   - `curry_flip :: ((a,b) -> c) -> (b -> a -> c)`
   - `uncurry_flip :: (a -> b -> c) -> ((b,a) -> c)`

   Überlegen Sie sich geeignete Funktionen, um die korrekte Arbeitsweise der einzelnen Funktionale zu überprüfen und testen. Implementieren Sie sie und testen Sie Ihre Implementierungen der Funktionale damit.

4. Schreiben Sie analog zur Haskell-Rechenvorschrift `verflechten` von Aufgabenblatt 1 eine Haskell-Rechenvorschrift `verflechten3 :: [Int] -> [Int] -> [Int] -> [Int]`, die in der gleichen Weise wie `verflechten` drei Argumentlisten zu einer Ergebnisliste verflechtet. Dabei soll das erste Element der Ergebnisliste aus der ersten Argumentliste sein, wenn diese nicht leer ist. Sonst aus der zweiten, wenn diese nicht leer ist. Sonst aus der dritten, wenn diese nicht leer ist. Ist eine Argumentliste vollständig verflochten, werden die beiden verbleibenden Argumentlisten ineinander verflochten. Ist auch die nächste Argumentliste vollständig verflochten, so bilden die verbliebenen Elemente der letzten Argumentliste den Rest der Ergebnisliste.

   Anwendungsbeispiele:

```
verflechten3 [1,2,3] [4,5,6] [7,8,9] ->> [1,4,7,2,5,8,3,6,9]
verflechten3 [1,2,3] [4] [5,6,7]     ->> [1,4,5,2,6,3,7]
verflechten3 [1,2] [3,4,5,6] []      ->> [1,3,2,4,5,6]
verflechten3 [] [1,2,3,4] [5,6]      ->> [1,5,2,6,3,4]
```

Wichtig: Wenn Sie einzelne Rechenvorschriften aus früheren Lösungen für dieses oder spätere Aufgabenblätter wieder verwenden möchten, so kopieren Sie diese in die neue Abgabedatei ein. Ein `import` schlägt für die Auswertung durch das Abgabeskript fehl, weil Ihre alte Lösung, aus der importiert wird, nicht mit abgesammelt wird. Deshalb: Kopieren statt importieren zur Wiederwendung!

Denken Sie bitte daran, dass Sie für die Lösung dieses Aufgabenblatts ein "literate" Haskell-Skript schreiben sollen!

Haskell Live

An einem der kommenden Termine werden wir uns in *Haskell Live* mit den Beispielen der ersten beiden Aufgabenblätter beschäftigen, sowie mit der Aufgabe *City-Maut*.

City-Maut

Viele Städte überlegen die Einführung einer City-Maut, um die Verkehrsströme kontrollieren und besser steuern zu können. Für die Einführung einer City-Maut sind verschiedene Modelle denkbar. In unserem Modell liegt ein besonderer Schwerpunkt auf innerstädtischen Nadelöhren. Unter einem *Nadelöhr* verstehen wir eine Verkehrsstelle, die auf dem Weg von einem Stadtteil A zu einem Stadtteil B in der Stadt passiert werden muss, für den es also keine Umfahrung gibt. Um den Verkehr an diesen Nadelöhren zu beeinflussen, sollen an genau diesen Stellen Mautstationen eingerichtet und Mobilitätsgebühren eingehoben werden.

In einer Stadt mit den Stadtteilen A, B, C, D, E und F und den sieben in beiden Richtungen befahrbaren Routen B–C, A–B, C–A, D–C, D–E, E–F und F–C führt jede Fahrt von Stadtteil A in Stadtteil E durch Stadtteil C. C ist also ein Nadelöhr und muss demnach mit einer Mautstation versehen werden.

Schreiben Sie ein Programm in Haskell oder in einer anderen Programmiersprache Ihrer Wahl, das für eine gegebene Stadt und darin vorgegebene Routen Anzahl und Namen aller Nadelöhre bestimmt.

Der Einfachheit halber gehen wir davon aus, dass anstelle von Stadtteilnamen von 1 beginnende fortlaufende Bezirksnummern verwendet werden. Der Stadt- und Routenplan wird dabei in Form eines Tupels zur Verfügung gestellt, das die Anzahl der Bezirke angibt und die möglichen jeweils in beiden Richtungen befahrbaren direkten Routen von Bezirk zu Bezirk innerhalb der Stadt. In Haskell könnte dies durch einen Wert des Datentyps `CityMap` realisiert werden:

```
type Bezirk     = Integer
type AnzBezirke = Integer
type Route      = (Bezirk,Bezirk)
newtype CityMap = CM (AnzBezirke,[Route])
```

Gültige Stadt- und Routenpläne müssen offenbar bestimmten Wohlgeformtheitsanforderungen genügen. Überlegen Sie sich, welche das sind und wie sie überprüft werden können, so dass Ihr Nadelöhrsuchprogramm nur auf wohlgeformte Stadt- und Routenpläne angewendet wird.