3. Aufgabenblatt zu Funktionale Programmierung vom 05.11.2014. Fällig: 12.11.2014 / 19.11.2014 (jeweils 15:00 Uhr)

Themen: *Rekursive Funktionen über Zahlen, Wahrheitswerten und Listen von Zahlen*

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. Im einzelnen sollen Sie die im folgenden beschriebenen Problemstellungen bearbeiten.

1. Eine *Bitfolge* ist eine Folge von Nullen und Einsen. Wir nennen eine Bitfolge *mehrfachnullfrei*, wenn sie keine unmittelbar aufeinanderfolgenden Vorkommen von Nullen enthält.

   (a) Schreiben Sie eine Haskell-Wahrheitswertrechenvorschrift mit der Signatur

   ```
   zeroTest :: [Integer] -> Bool
   ```

   Angewendet auf eine Liste ganzer Zahlen l liefert `zeroTest` den Wahrheitswert `True`, falls l mehrfachnullfreie Bitfolge ist; anderenfalls liefert `zeroTest` den Wahrheitswert `False`.

   (b) Schreiben Sie eine Haskell-Rechenvorschrift mit der Signatur

   ```
   numberOf :: Integer -> Integer
   ```

   Angewendet auf ein Argument n liefert die Funktion `numberOf` die Anzahl mehrfachnullfreier Bitfolgen der Länge Betrag $|n|$ von n.

2. Wir repräsentieren die Werte der sieben Euromünzen 1 Cent, 2 Cent, 5 Cent, 10 Cent, 20 Cent, 50 Cent, 1 Euro und 2 Euro durch die (mit Einheit Eurocent gedachten) Zahlwerte 1, 2, 5, 10, 20, 50, 100 und 200.

   (a) Schreiben Sie eine Haskell-Rechenvorschrift

   ```
   minNumOfCoins :: Integer -> Integer
   ```

   Angewendet auf ein Argument n liefert die Funktion `minNumOfCoins` die kleinste Anzahl von Euromünzen, mit der der durch den Betrag $|n|$ von n in Cent gegebene Eurobetrag in Euromünzen bezahlt werden kann.

   (b) Schreiben Sie eine Haskell-Rechenvorschrift

   ```
   numOfSplits :: Integer -> Integer
   ```

   Angewendet auf ein Argument n liefert die Funktion `numOfSplits` die Anzahl aller Möglichkeiten, wie der durch den Betrag $|n|$ von n in Cent gegebene Eurobetrag in Euromünzen bezahlt werden kann, falls $|n|$ kleiner oder gleich 500 ist. Anderenfalls liefert `numOfSplits` den Wert -1.

(c) Schreiben Sie eine Haskell-Rechenvorschrift mit der Signatur

```
change :: Integer -> Integer -> [Integer]
```

Die Funktion `change` liefert eine (von möglicherweise vielen) Möglichkeiten, wie ein Geldbetrag mit einer vorgegebenen Anzahl von Euromünzen bezahlt werden kann, so dies möglich ist. Anderenfalls oder wenn der Ausgangsgeldbetrag zu groß ist, liefert die Funktion die leere Liste als Resultat. Im Detail: Angewendet auf zwei Argumente n und m mit $|n| \leq 500$ liefert die Funktion `change` als Resultat eine absteigend sortierte Liste der Länge Betrag $|m|$ von Euromünzbeträgen, deren Summe den Wert $|n|$ ergibt. Gibt es mehrere solcher Möglichkeiten, ist es egal, welche dieser Wechselmöglichkeiten `change` liefert. Gibt es keine Möglichkeit den Betrag $|n|$ mit genau $|m|$ Euromünzen zu bezahlen, oder ist $|n| > 500$, so liefert `change` die leere Liste als Resultat. Beispiele: `change 19 4 ->> [10,5,2,2]`, `change 5 2 ->> []`, `change 1000 1000 ->> []`, `change 22 3 ->>` z.B. `[20,1,1]` oder `[10,10,2]`).

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.