3. Aufgabenblatt zu Funktionale Programmierung vom 01.11.2012. Fällig: 14.11.2012 / 21.11.2012 (jeweils 15:00 Uhr)

Themen: *Funktionen auf Zahlen und Zeichenreihen*

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 Gruppenaccounts 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 brauchen, 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.

Ähnlich wie im Binärsystem zur Basis 2, dem sog. 2-adischen Zahlsystem, lassen sich die ganzen Zahlen auch in einem Zahlsystem zur Basis -2 darstellen, dem sog. (-2)-adischen Zahlsystem.

Den Schlüssel dazu liefert uns der folgende Satz: Jede ganze Zahl $z \in \mathbb{Z}\backslash\{0\}$ läßt sich auf genau eine Weise in der Form

$$z = \sum_{i=0}^{n} d_i * (-2)^i$$

darstellen, wobei $n \in I\!N \cup \{0\}$, $d_i \in \{0, 1\}$, $d_n \neq 0$. Mit der Zusatzvereinbarung die Zahl 0 auch im (-2)-adischen System durch 0 darzustellen, besitzt jede ganze Zahl eine eindeutige Darstellung ohne führende Nullen im (-2)-adischen System (auch hier mit Ausnahme der 0 selbst). So entspricht z.B. die Dezimalzahl 2 der (-2)-adischen Zahl 110, 3 der (-2)-adischen Zahl 111, die Dezimalzahl -2 der (-2)-adischen Zahl 10, die Dezimalzahl -3 der (-2)-adischen Zahl 1101, die Dezimalzahl -5 der (-2)-adischen Zahl 1111.

In der Folge verwenden wir Zeichenreihen zur Darstellung (-2)-adischer Zahlen in Haskell.

```
type NegaBinary = String
```

- Schreiben Sie eine Haskell-Rechenvorschrift `extract :: String -> NegaBinary`, die aus einer Zeichenreihe die in ihr enthaltene (-2)-adische Zahl extrahiert. Zu diesem Zweck entfernt die Funktion `extract` alle im Argument von `'0'` und `'1'` verschiedenen Zeichen sowie alle eventuell verbliebenen führenden Nullen. Diese Zeichenreihe wird als Resultat zurückgegeben, falls sie nicht leer ist; ansonsten wird die (-2)-adische Zahl `"0"` als Resultat zurückgegeben.

  Folgende Beispiele zeigen das gewünschte Ein-/Ausgabeverhalten:

  ```
  extract "we10iopK11P iU010" ->> "1011010"
  extract "0w0e010iopK101P iU01B1" ->> "10101011"
  extract "aerfweife" ->> "0"
  extract "a0erf00we0ife" ->> "0"
  extract "a0erf00w1e0ife" ->> "10"
  ```

- Schreiben Sie eine Haskell-Rechenvorschrift `nbIncr :: NegaBinary -> NegaBinary`, die eine als Argument übergebene (-2)-adische Zahl inkrementiert, d.h. um 1 erhöht.

  Folgende Beispiele zeigen das gewünschte Ein-/Ausgabeverhalten:

  ```
  nbIncr "110" ->> "111"
  nbIncr "10" ->> "11"
  nbIncr "1101" ->> "10"
  nbIncr "11111" ->> "11100"
  nbIncr "11" ->> "0"
  ```

- Schreiben Sie eine Haskell-Rechenvorschrift `nbDecr :: NegaBinary -> NegaBinary`, die eine als Argument übergebene (-2)-adische Zahl dekrementiert, d.h. um 1 erniedrigt.

  Folgende Beispiele zeigen das gewünschte Ein-/Ausgabeverhalten:

```
nbDecr "1" ->> "0"
nbDecr "1101" ->> "1100"
nbDecr "1110" ->> "1001"
nbDecr "11111" ->> "11110"
nbDecr "110" ->> "1"
```

- Schreiben Sie eine Haskell-Rechenvorschrift `nbAbs :: NegaBinary -> NegaBinary`, die den Absolutbetrag einer als Argument übergebenen (-2)-adischen Zahl berechnet.

  Folgende Beispiele zeigen das gewünschte Ein-/Ausgabeverhalten:

  ```
  nbAbs "0" ->> "0"
  nbAbs "11" ->> "1"
  nbAbs "1111" ->> "101"
  nbAbs "111" ->> "111"
  nbAbs "1110" ->> "11010"
  ```

  Hinweis: Sie können den Absolutbetrag etwa über eine Konvertierung ins Dezimalsystem berechnen mit anschließender Rückkonvertierung ins (-2)-adische System. Ein anderer Weg, ohne das (-2)-adische System zu verlassen, ist es, negative Argumente so oft zu inkrementieren, bis der Wert 0 erreicht ist. Anschließend ist von 0 ausgehend noch einmal so oft zu inkrementieren, um so den Absolutbetrag des ursprünglichen Arguments zu erreichen. Überlegen Sie sich dazu, wie Sie feststellen können, ob ein Argument einen negativen Wert repräsentiert, ohne das (-2)-adische System zu verlassen, also insbesondere ohne das Argument etwa ins Dezimalsystem zu konvertieren.

Die Addition und Multiplikation zweier ganzer Zahlen m und n lässt sich auf die n-fache Inkrementierung von m und die n-fache Addition von n zu m zurückführen. Entwickeln Sie aufbauend auf dieser Idee zwei rekursive Rechenvorschriften zur Addition und Multiplikation zweier (-2)-adischer Zahlen.

- Schreiben Sie eine Haskell-Rechenvorschrift `nbPlus :: NegaBinary -> NegaBinary -> NegaBinary`, die ihre beiden Argumente addiert.

  Folgende Beispiele zeigen das gewünschte Ein-/Ausgabeverhalten:

  ```
  nbPlus "0" "1110" -> "1110"
  nbPlus "10101" "0" ->> "10101"
  nbPlus "110" "111" ->> "101"
  nbPlus "10" "1100" ->> "1110"
  nbPlus "1100" "101" ->> "1"
  ```

- Schreiben Sie eine Haskell-Rechenvorschrift `nbTimes :: NegaBinary -> NegaBinary -> NegaBinary`, die ihre beiden Argumente multipliziert.

  Folgende Beispiele zeigen das gewünschte Ein-/Ausgabeverhalten:

  ```
  nbTimes "0" "1110" -> "0"
  nbTimes "10101" "0" ->> "0"
  nbTimes "10" "1100" ->> "11000"
  nbTimes "110" "111" ->> "11010"
  nbTimes "110" "1101" ->> "1110"
  ```

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

Hinweis: Blatt 4 erscheint am 07.11.2012 mit Abgabeterminen am 14.11.2012 und 21.11.2012. Planen Sie bitte entsprechend.

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.