8. Aufgabenblatt zu Funktionale Programmierung vom 04.12.2006. Fällig: 12.12.2006 / 09.01.2007 (jeweils 15:00 Uhr)

Themen: *Funktionale auf Listen, Programmieren mit Monaden*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe8.hs` ablegen. Sie sollen für die Lösung dieses Aufgabenblats also wieder ein "gewöhnliches" Haskell-Skript schreiben. Versehen Sie wieder 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. Denken Sie bei der Kommentierung daran, dass Ihre Lösungen auch Grundlage für das Abgabegespräch am Semesterende sind.

Im einzelnen sollen Sie die im folgenden beschriebenen Problemstellungen bearbeiten.

1. Versuchen Sie bei der Implementierung der Rechenvorschriften `foldlr` und `sumMinBoth` aus den nächsten beiden Teilaufgaben möglichst vorteilhaft von vordefinierten Funktionen wie `map`, `foldr`, `foldl`, `flip`, `zip`, `take`, etc. Gebrauch zu machen.

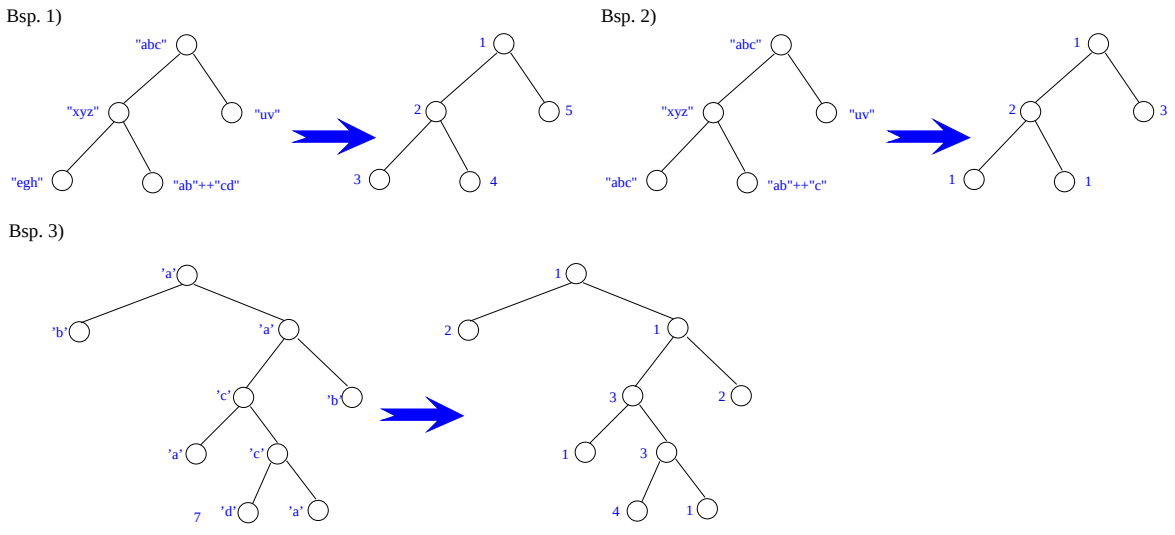
   (a) Schreiben Sie eine Haskell-Rechenvorschrift `foldlr` mit der Signatur `foldlr :: (a -> a -> b) - [a] -> [b]`. Angewendet auf eine Funktion f und eine Argumentliste $[x1, \ldots, x_n]$ liefert die Rechenvorschrift `foldlr` für gerades n die Resultatliste $[f\ x_1\ x_n, f\ x_2\ x_{n-1}, \ldots, f\ x_{\frac{n}{2}}\ x_{\frac{n}{2}+1}]$, für ungerades n die Resultatliste $[f\ x_1\ x_n, f\ x_2\ x_{n-1}, \ldots, f\ x_{\frac{n+1}{2}-1}\ x_{\frac{n+1}{2}+1}]$. Mit anderen Worten: Bei Argumentlisten ungerader Länge bleibt das mittlere Element unberücksichtigt.

   (b) Schreiben Sie eine Haskell-Rechenvorschrift `sumMinBoth` mit der Signatur `sumMinBoth :: [Int] -> [Int] -> Int`. Angewendet auf zwei Argumentlisten liefert die Funktion `sumMinBoth` die Summe all der Elemente, die sowohl in der ersten wie in der zweiten Liste vorkommen. D.h., Elemente, die nur in einer der Listen vorkommen, bleiben bei der Summenbildung unberücksichtigt; Elemente, die in beiden Listen vorkommen, werden so oft aufaddiert, wie sie in beiden Listen vorkommen. Angewendet auf die Listen `[2,3,4,2,2,3,1]` und `[3,2,1,5,1,2]` soll sich das Resultat `8` (als Summe von `1+2+2+3`) ergeben; `4` und `5` bleiben unberücksichtigt, weil sie jeweils nur in einer der beiden Argumentlisten vorkommen; ebenso die "überzähligen" Vorkommen von `1`, `2` und `3`.

2. Gegeben sei:

   ```
   data Tree a = Leaf a | Node a (Tree a) (Tree a) deriving (Eq,Show)
   ```

   Gesucht ist eine Haskell-Rechenvorschrift `renameTree` mit der Signatur `renameTree :: (Eq a, Show a) => Tree a -> Tree Int`, die angewendet auf einen Binärbaum einen Binärbaum über ganzen Zahlen zurückliefert. Gleiche Benennungen im Argumentbaum sollen dabei auf gleiche Zahlen im Resultatbaum abgebildet werden. Die Benennungen im Resultatbaum werden den Benennungen im Argumentbaum dabei beginnend mit 1 in aufsteigender Folge entsprechend eines Durchlaufs des Argumentbaums in Präfixordnung vergeben. Die folgende Abbildung illustriert das gewünschte Abbildungsverhalten:



Hinweis: In Präfixordung wird zunächst die Wurzel, dann der linke Teilbaum, schließlich der rechte Teilbaum eines Baums besucht. Um die schon den im Argumentbaum aufgefundenen Benennungen zugeordneten Zahlen zu verwalten und wiederaufzufinden, bietet sich an, diese Benennungen in einer Tafel (repräsentiert durch ein Liste) abzulegen. Die Indizes dieser Liste können dann direkt oder indirekt die einer Benennung des Argumentbaums zugehörige Zahl angeben. Denken Sie daran, dass der erste Index einer Liste die 0 ist:

```
type Table a = [a]
```

3. In dieser Aufgabe betrachten wir die gleiche Aufgabenstellung wie in der vorherigen Teilaufgabe, wollen diese nun aber mittels monadischer Programmierung lösen. Dazu betrachten wir die sog. *Zustandsmonade*, d.h. den Datentyp `State` mit

   ```
   data State s a = St (s -> (a,s))
   ```

   und der Instanzbildung

   ```
   instance Monad (State s) where
     return x     = (St (\s -> (x,s))
     (St m) >>= f =  St (\s -> let (x,s1) = m s
                                   St g = f x
                               in g s1)
   ```

   Mithilfe dieses Datentyps lassen sich *Zustandstransformationen* ausdrücken, d.h. Funktionen `f` vom Typ `f :: s -> (a,s)`, die einen initialen Zustand `s1` vom Typ `s` auf ein Paar bestehend aus einem (möglicherweise veränderten) Zustand `s2` vom Typ `s` und einem Wert `a1` vom Typ `a` abbilden.

   Gesucht ist nun eine Haskell-Rechenvorschrift `monRenameTree` mit der Signatur `monRenameTree :: (Eq a, Show a) => Tree a -> State (Table a) (Tree Int)`, die einen Binärbaum wie in der vorigen Teilaufgabe beschrieben in einen Resultatbaum überführt, wobei `Table a` wie in der vorigen Teilaufgabe vorgeschlagen ist, d.h.:

   ```
   type Table a = [a]
   ```

   und zur Buchführung über die Zuordnung von Argumentbaum- zu Resultatbaumbenennungen verwendet wird. Sehen Sie in Ihrer Lösung zusätzlich eine Funktion `mRT` vor mit

   ```
   mRT :: (Eq a, Show a) => Tree a -> Tree Int
   mRT = extract . monRenameTree
   ```

   wobei `extract` eine Extraktionsfunktion mit folgender Signatur ist:

   ```
   extract :: (Eq a, Show a) => (State s a) -> a}
   ```

Hinweis:

- Verwenden Sie *keine* Module. Wenn Sie Funktionen wiederverwenden möchten, kopieren Sie diese in die Abgabedatei `Aufgabe8.hs`. Andere als diese Datei werden vom Abgabeskript ignoriert.
- Bitte testen Sie Ihre Lösungen mit der aktuellen Version von Hugs auf der b1: `/usr/local/bin/hugs`