4. Aufgabenblatt zu Funktionale Programmierung vom 30.10.2007. Fällig: 06.11.2006 / 13.11.2007 (jeweils 15:00 Uhr)

Themen: *Funktionen auf algebraischen Datentypen*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe4.hs` ablegen. Wie etwa für die Lösung zum ersten Aufgabenblatt sollen Sie dieses Mal 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. Im einzelnen sollen Sie die im folgenden beschriebenen Problemstellungen bearbeiten.

1. In dieser Aufgabe betrachten wir Binärbäume, die wir in Haskell durch folgenden Typ repräsentieren:

   ```
   data Tree = Nil |
               Node Int Tree Tree
   ```

   Schreiben Sie eine Haskell-Rechenvorschrift `flatten` mit der Signatur `flatten :: Tree -> DOrd -> [Int]`, die die Knotenmarkierungen des Argumentbaums in Form einer Liste ausgibt. Dabei kontrolliert das Argument vom Typ `DOrd`, ob die Knoten des Argumentbaums in Infixordnung (also in der Reihenfolge: linker Teilbaum, Wurzel, rechter Teilbaum), Präfixordnung (Wurzel, linker Teilbaum, rechter Teilbaum) oder Postfixordnung (linker Teilbaum, rechter Teilbaum, Wurzel) besucht werden. Zusätzlich betrachten wir zu jeder dieser drei Durchlaufordnungen eine zugehörige *gespiegelte* Durchlaufordnung, in der der Besuch von Wurzel, linkem und rechtem Teilbaum in jeweils umgekehrter Reihenfolge erfolgt. Gespiegelt in Infixordnung steht also für die Durchlaufordnung: Rechter Teilbaum, Wurzel, linker Teilbaum. Gespiegelt in Präfixordnung für: Rechter Teilbaum, linker Teilbaum, Wurzel. Gespiegelt in Postfixordnung für: Wurzel, rechter Teilbaum, linker Teilbaum. Der Typ `DOrd` ist als Aufzählungstyp schließlich folgendermaßen definiert.

   ```
   data DOrd = Infix | Praefix | Postfix |
               GspInfix | GspPraefix | GspPostfix
   ```

2. In dieser Aufgabe betrachten wir noch einmal Binärbäume, dieses Mal aber modelliert durch folgenden Haskell-Typ:

   ```
   data BTree = Leaf Char |
                BNode Int BTree BTree deriving (Show,Eq)
   ```

   Schreiben Sie eine Haskell-Rechenvorschrift `topspiegel` mit der Signatur `topspiegel :: BTree -> BTree`, die angewendet auf einen Baum B vom Typ `BTree` als Resultat einen Baum C zurückliefert, dessen Wurzel wie diejenige von B benannt ist und dessen linker (rechter) Teilbaum mit dem rechten (linken) Teilbaum von B übereinstimmt. Besteht der Argumentbaum nur aus der Wurzel (d.h. nur aus einem benannten Blatt), so ist das Resultat ein ebenso benannter Wurzelbaum.

3. Wir betrachten noch einmal die Datenstruktur aus der vorigen Teilaufgabe. Wandeln Sie die Rechenvorschrift `topspiegel` zu einer Haskell-Rechenvorschrift `spiegel` mit der Signatur `spiegel :: BTree -> BTree` ab, die angewendet auf einen Baum als Resultat denjenigen Baum zurückliefert, in dem der rechte und linke Teilbaum des Argumentbaums auf allen Ebenen miteinander vertauscht sind. Besteht der Argumentbaum nur aus der Wurzel, so ist das Resultat wie schon bei der Funktion `topspiegel` ein ebenso benannter Wurzelbaum.

4. Neben Adjazenzmatrizen sind Adjazenzlisten eine weitverbreitete Form zur Darstellung von Graphen. Im Prinzip ist eine Adjazenzlistendarstellung eines Graphen eine Liste von Knoten, wobei jedem Knoten die Liste der von ihm aus über Kanten erreichbaren Nachbarknoten zugeordnet ist, wobei zugleich dadurch die Richtung der Kanten kodiert ist: Vom Knoten jeweils zu den Knoten in seiner Nachbarknotenliste. Wir erlauben dabei, dass Knoten mehrfach in einer Nachbarknotenliste auftauchen können und interpretieren dies als unterschiedliche Wege zwischen zwei Knoten.

   In Haskell können wir zur Darstellung folgenden algebraischen Typ verwenden, wobei die Knotenliste des Graphen eine geschachtelte Struktur annimmt:

```
data Graph = Empty |
             GNode Int [Int] Graph
```

Für diese Aufgabe nehmen wir nun an, dass die Knotenmenge eines Graphen implizit definiert ist, d.h. als Vorkommen als Argument eines Vorkommens des Konstruktors `GNode` oder als Vorkommen in der Adjazenzliste eines Knotens. Knoten, die ausschliesslich in einer Adjazenzliste auftauchen, besitzen also keine Nachfolger.

Schreiben Sie jetzt eine Haskell-Rechenvorschrift `wege` mit der Signatur `wege :: Graph -> Int -> Int -> Classification`, die bestimmt, ob es zwischen zwei Knoten eines Graphen keinen Weg, genau einen Weg oder mehr als einen Weg gibt. D.h., angewendet auf einen Graphen g und zwei Knoten m und n liefert die Funktion `wege` entsprechend die Werte `NoWay`, `OneWay` oder `ManyWays` ab. Ist einer der Argumentknoten nicht in g, liefert die Funktion als Resultat den Wert `InvalidInput` zurück. Der Typ `Classification` ist also durch folgenden Aufzählungstyp in Haskell gegeben:

```
data Classification = NoWay | OneWay | ManyWays | InvalidInput
```