8. Aufgabenblatt zu Funktionale Programmierung vom 25.11.2008. Fällig: 02.12.2008 / 09.12.2008 (jeweils 15:00 Uhr)

Themen: *Funktionen höherer Ordnung*

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 Aufgabenblatts also 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. "Teile und Herrsche" beschreibt ein wichtiges Organisationsprinzip von Algorithmen. Wenn wir zwei Typen `p` und `s` annehmen, deren Werte Probleme bzw. Lösungen dieser Probleme beschreiben, läßt sich das diesem Prinzip zugrundeliegende Organisationsschema in einem Funktional `divAndConquer` kapseln:

```
divAndConquer :: (p -> Bool) -> (p -> s) -> (p -> [p]) -> (p -> [s] -> s) -> p -> s
divAndConquer ind solve divide combine initProblem
    = dac initProblem
    where dac problem
              | ind problem = solve problem
              | otherwise   = combine problem (map dac (divide problem))
```

   Dabei stützt sich das Funktional `divAndConquer` auf folgende Argumentfunktionen:

   - `ind :: p -> Bool`: ...liefert `True`, wenn die als Argument übergebene Probleminstanz nicht mehr teilbar ist, `False` sonst.
   - `solve :: p -> s`: ...liefert die Lösung zu einer nicht mehr teilbaren Probleminstanz.
   - `divide :: p -> [p]`: ...liefert eine Liste von Instanzen von Teilproblemen, wenn die als Argument übergebene Probleminstanz teilbar ist.
   - `combine :: p -> [s] -> s`: ...konstruiert aus der Instanz des Ausgangsproblems und den Lösungen seiner Teilprobleme die Lösung des Ausgangsproblems.

   (a) Am 27.11.2007 hat Sir Tony Hoare an der TU Wien einen Kolloquiumsvortrag gehalten. Zeigen Sie aus Anlass dieses Besuchs, dass der von Tony Hoare erfundene `quickSort`-Algorithmus und die in der Vorlesung besprochene darauf beruhende Funktion `quickSort` dem "Teile und Herrsche"-Prinzip genügt und sich als Spezialisierung des Funktionals `divAndConquer` ergibt. Geben Sie dazu geeignete Implementierungen der Funktionen `indq`, `solveq`, `divideq` und `combineq` an, so dass `quickSort` in Ihrer Abgabedatei insgesamt wie folgt implementiert ist:

```
quickSort :: Ord a => [a] -> [a]
quickSort = divAndConquer indq solveq divideq combineq
```

   (b) Betrachten Sie die folgende Gleichung zur Definition der Binomialkoeffizienten:

$$\begin{pmatrix} n \\ k \end{pmatrix} = \begin{pmatrix} n-1 \\ k-1 \end{pmatrix} + \begin{pmatrix} n-1 \\ k \end{pmatrix}$$

   und zeigen Sie, dass sich die Rechenvorschrift `binom :: (Integer,Integer) -> Integer` ebenfalls als Spezialisierung des Funktionals `divAndConquer` angeben lässt, indem Sie wieder geeignete Implementierungen der Funktionen `indb`, `solveb`, `divideb` und `combineb` angeben, so dass `binom bbbn` in Ihrer Abgabedatei insgesamt wie folgt implementiert ist:

```
binom :: (Integer,Integer) -> Integer
binom = divAndConquer indb solveb divideb combineb
```

In den folgenden beiden Teilaufgaben betrachten wir folgenden polymorphen Typ auf Bäumen:

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

2. Analog zu den in Haskell vordefinierten Funktionen auf Listen `zip` und `unzip` wollen wir jetzt Funktionen auf Werten vom Typ `Tree a` mit vergleichbarer Funktionalität schreiben. Im einzelnen:

   - Schreiben Sie eine Haskell-Rechenvorschrift `zipT` mit der Signatur `zipT :: Tree a -> Tree b -> Tree (a,b)`. Angewendet auf zwei Argumentbäume liefert die Funktion `zipT` als Resultat einen neuen Baum, dessen Knoten mit Paaren benannt sind, deren Komponenten die Benennungen der Argumentbäume sind. Hat einer der Teilbäume den Wert `Nil`, während der andere noch einen Wert davon verschieden hat, so bleibt ähnlich wie bei der Funktion `zip` auf Listen dieser Teilbaum unberücksichtigt. Folgende Beispiele illustrieren die gewünschte Funktionalität von `zipT`:

     ```
     zipT (Node 4 (Node 3 Nil Nil) Nil) (Node "abc" (Node "xyz" Nil Nil) Nil)
               == (Node (4,"abc") (Node (3,"xyz") Nil Nil)) Nil)
     zipT (Node 4 (Node 3 Nil Nil) Nil) (Node "abc" Nil Nil)
               == (Node (4,"abc") Nil Nil) Nil)
     ```

   - Schreiben Sie eine Haskell-Rechenvorschrift `unzipT` mit der Signatur `unzipT :: Tree (a,b) -> (Tree a,Tree b)`. Angewendet auf einen Argumentbaum liefert die Funktion `unzipT` als Resultat ein Paar strukturell gleicher Bäume, deren Knoten mit der linken bzw. rechten Komponente des Paares bezeichnet ist, das den entsprechenden Knoten im Argumentbaum bezeichnet. Folgende Beispiele illustrieren die gewünschte Funktionalität von `unzipT`:

     ```
     unzipT (Node (4,"abc") (Node (3,"xyz") Nil Nil)) Nil
      == ((Node 4 (Node 3 Nil Nil)) Nil),(Node "abc" (Node "xyz" Nil Nil)) Nil))
     unzipT (Node (4,"abc") Nil Nil) Nil)
      == ((Node 4 Nil Nil) Nil),(Node "abc" Nil Nil) Nil))
     ```

3. In dieser Aufgabe wollen wir die Funktionen `zipT` und `unzipT` um ein weiteres Argument erweitern, eine Manipulationsfunktion `f`. Auf diese Weise erhalten wir die Funktionale `zipTF` und `unzipTF`.

   - Schreiben Sie eine Haskell-Rechenvorschrift `zipTF` mit der Signatur `zipTF :: (a -> b -> c) -> Tree a -> Tree b -> Tree c`. Angewendet auf eine Manipulationsfunktion `f` und zwei Argumentbäume `s` und `t` liefert die Funktion `zipTF` als Resultat einen Baum, dessen Knotenbenennungen sich als Bild der Anwendung der Funktion `f` auf die Benennungen der entsprechenden Knoten von `s` und `t` ergibt. Hat einer der Teilbäume von `s` oder `t` den Wert `Nil`, während der andere noch einen Wert davon verschieden hat, so bleibt ähnlich wie bei der Funktion `zip` auf Listen dieser nichtleere Teilbaum unberücksichtigt. Folgende Beispiele illustrieren die gewünschte Funktionalität von `zipTF`:

     ```
     zipTF (+) (Node 4 (Node 3 Nil Nil) Nil) (Node 2 (Node 7 Nil Nil) Nil)
                            == (Node 6 (Node 10 Nil Nil)) Nil)
     zipTF (++) (Node "abc" (Node "def" Nil Nil) Nil) (Node "xyz" Nil Nil)
                            == (Node "abcxyz") Nil Nil) Nil)
     ```

   - Schreiben Sie eine Haskell-Rechenvorschrift `unzipTF` mit der Signatur `unzipTF :: ((a,b) -> (c,d)) -> Tree (a,b) -> (Tree c,Tree d)`. Angewendet auf eine Manipulationsfunktion `f` und einen Argumentbaum `t` liefert die Funktion `unzipTF` als Resultat einen Baum, dessen Knotenbenennungen sich als Projektion auf die erste bzw. zweite Komponente des Bildes der Anwendung der Funktion `f` auf die Benennung des entsprechenden Knotens von `t` ergibt. Folgende Beispiele illustrieren die gewünschte Funktionalität von `unzipTF`:

     ```
     unzipTF (\x->x) (Node (4,"abc") (Node (3,"xyz") Nil Nil) Nil)
       == ((Node 4 (Node 3 Nil Nil) Nil),(Node "abc" (Node "xyz" Nil Nil)
     unzipTF (\(x,y)->(y,x)) (Node (4,"abc") (Node (3,"xyz") Nil Nil) Nil)
       == ((Node "abc" (Node "xyz" Nil Nil),(Node 4 (Node 3 Nil Nil) Nil))
     ```