6. Aufgabenblatt zu Funktionale Programmierung vom 23.11.2011.
Fällig: 30.11.2011 / 07.12.2011 (jeweils 15:00 Uhr)

Themen: *Funktionen höherer Ordnung, algebraische Datentypen*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe6.hs` ablegen. 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.

- Wir betrachten folgende Typen:

```
data DOrd = Infix | Praefix | Postfix |
            GInfix | GPraefix | GPostfix
data BTree = Nil |
             BNode Int BTree BTree deriving (Eq,Ord,Show)
```

  Schreiben Sie eine Haskell-Rechenvorschrift `flatten :: BTree -> DOrd -> [Int]`, die die Knotenmarkierungen des Argumentbaums in Form einer Liste ausgibt. Das Argument vom Typ `DOrd` kontrolliert dabei, 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.

  Folgende Beispiele zeigen das gewünschte Aufruf-/Rückgabeverhalten:

```
t1 = Nil
t2 = BNode 2 (BNode 3 Nil Nil) (BNode 5 Nil Nil)
flatten t1 Infix ->> []
flatten t2 Infix ->> [3,2,5]
flatten t2 GInfix ->> [5,2,3]
flatten t2 Praefix ->> [2,3,5]
flatten t2 GPostfix ->> [2,5,3]
```

- Ein Baum von Typ `BTree` heißt *Suchbaum*, wenn für alle Knoten `k` des Baums gilt, dass die Benennungen im linken Teilbaum, so vorhanden, echt kleiner und die im rechten Teilbaum, so vorhanden, echt größer als die Benennung von `k` sind. Schreiben Sie eine Wahrheitswertfunktion `isST :: BTree -> Bool`, die überprüft, ob ein Argumentbaum ein Suchbaum ist. Folgende Beispiele zeigen das gewünschte Aufruf-/Rückgabeverhalten:

```
t1 = Nil
t2 = BNode 2 (BNode 3 Nil Nil) (BNode 5 Nil Nil)
t3 = BNode 4 (BNode 3 Nil Nil) (BNode 5 Nil Nil)
t4 = BNode 2 (BNode 3 Nil Nil) (BNode 3 Nil Nil)
isST t1 ->> True
isST t2 ->> False
isST t3 ->> True
isST t4 ->> False
```

- Wir betrachten folgende Typen:

```
type Control = String
type Func    = Integer -> Integer
```

```
type Data    = Integer

data Tree    = Leaf Func
             | Node Func Tree Tree Tree
```

Zeichenreihen vom Typ `Control` sollen nur die Zeichen 'l', 'm' und 'r' enthalten. Schreiben Sie deshalb eine Haskell-Rechenvorschrift `mkControl :: String -> Control`, die angewendet auf eine Zeichenreihe s eine Zeichenreiche c liefert, in der alle Zeichen in s verschieden von 'l', 'm' und 'r' gestrichen sind. Werte vom Typ `Control` verstehen wir in der Folge als Steuerzeichen, die den Durchlauf durch einen Baum kontrollieren. Dabei steht 'l' für "linker Teilbaum", 'm' für "mittlerer Teilbaum" und 'r' für "rechter Teilbaum".

Schreiben Sie nun eine Haskell-Rechenvorschrift `apply :: Control -> Data -> Tree -> Integer`. Die Zeichen aus dem Argument `mkControl s` für `s` vom Typ `Zeichenreihe` geben dabei an, wie der Baum durchlaufen werden soll; ist das Kopfelement von `mkControl s` ein 'l', fahre im linken Teilbaum fort, ist das Kopfelement von `mkControl s` ein 'm', fahre im mittleren Teilbaum fort; ist das Kopfelement von `mkControl s` ein 'r', fahre im rechten Teilbaum fort. Angewendet auf Argumente `mkControl s`, d und t, wird auf d die Funktion f angewendet, die den Wurzelknoten von t markiert. Anschließend wird `apply` rekursiv auf das um das Kopfelement verkürzte Kontrollelement, den Wert $f(d)$ und in Abhängigkeit des Wertes des Kopfelementes von c linken, mittleren oder rechten Teilbaum angewendet. Ist das Kontrollelement leer, wird auf das Datumsargument von `apply` noch die den Wurzelknoten markierende Funktion angewendet und die Auswertung von `apply` terminiert. Ist das Kontrollelement noch nicht leer, aber bereits ein Blatt erreicht, d.h. die Auswertung kann nicht mehr in einem Teilbaum fortgesetzt werden, terminiert die Auswertung von `apply` ebenso.

Folgende Beispiele zeigen das gewünschte Aufruf-/Rückgabeverhalten:

```
t1 = Leaf (\x->2*x+3)
t2 = Node (*2) (Leaf (*3)) (Leaf (*5)) (Leaf (*7))
t3 = Node (*2) (Leaf (*3)) (Node (+1) (Leaf (*5)) (Leaf (+5)) (Leaf (+2))) (Leaf (*7))
apply ['r','l','l','m'] 5 t1 ->> 13
apply ['r','l','l','m'] 5 t2 ->> 70
apply ['m','l'] 5 t3 ->> 55
apply [] 5 t2 ->> 10
```

- Wir betrachten folgende Typen:

```
type Func  = Integer -> Integer
data LTree = LNode Integer [LTree] deriving Show
```

Schreiben Sie eine Haskell-Rechenvorschrift `mapLT :: Func -> LTree -> LTree`, die das Analogon des auf Listen arbeitenden Funktionals `map` für Bäume vom Typ `LTree` ist.

Folgende Beispiele zeigen das gewünschte Aufruf-/Rückgabeverhalten:

```
t1 = LNode 5 []
t2 = LNode 5 [LNode 10 [], LNode 15 [LNode 20 [], LNode 25 []]]
t3 = LNode 5 [LNode 10 [LNode 15 [], LNode 20 [LNode 25 []]]]
mapLT (*2) t1 ->> LNode 10 []
mapLT (*2) t2 ->> LNode 10 [LNode 20 [], LNode 30 [LNode 40 [], LNode 50 []]]
mapLT (*2) t3 ->> LNode 10 [LNode 20 [LNode 30 [], LNode 40 [LNode 50 []]]]
```

Haskell Live

Der nächste *Haskell Live*-Termin ist am Freitag, den 25.11.2011.