Klausur „Algorithmen und Datenstrukturen (AuD) I“
(Vorlesung WS 2009/10)

22. März 2010, 10:30 – 12:30

Aufgabe 1 [Gesamtpunktzahl: 10]

Die im Modul `List.hs` vordefinierte Funktion `maximumBy` erwartet ein Vergleichsprädikat und eine Liste und gibt ein maximales Element aus der Liste bezüglich des Vergleichsprädikats zurück.

```
maximumBy  :: (a -> a -> Ordering) -> [a] -> a
```

Zur Erinnerung: Ein Vergleichsprädikat bestimmt für seine beiden Argumente die geltende Ordnungsbeziehung und gibt sie mit einem der `Ordering`-Werte zurück.

```
data Ordering = LT | EQ | GT
  deriving (Eq, Ord, Ix, Enum, Read, Show, Bounded)
```

Da es im allgemeinen mehrere maximale Elemente geben kann, interessiert uns eine Variante `maximaBy`, die statt nur eines maximalen Elements *alle maximalen Elemente* als Liste zurückgibt :

```
maximaBy  :: (a -> a -> Ordering) -> [a] -> [a]
```

a) Analysieren Sie den folgenden Lösungsvorschlag für `maximaBy`. Welches sind die Probleme damit?

```
import List

maximaBy _ [x] = [x]

maximaBy cmp xs = max:(maximaBy cmp (delete max xs))
          where max = maximumBy cmp xs
```

b) Schreiben Sie in **Haskell** eine Definition für `maximaBy`, bei der die Bestimmung aller maximalen Elemente *nur einen Durchlauf* durch die Liste erfordert.

Aufgabe 2 [Gesamtpunktzahl: 12]

a) Bauen Sie schrittweise von Hand mit den Elementen der folgenden Liste (in der Reihenfolge von links nach rechts) einen (gewöhnlichen) **binären Suchbaum** auf:

```
[10,1,8,6,3,2]
```

Skizzieren Sie das Ergebnis.

b) Bauen Sie dann schrittweise von Hand mit den Elementen der Liste aus a) (in der Reihenfolge von links nach rechts) einen **AVL-Baum** auf. Erläutern Sie dabei insbesondere jeweils mit Skizzen, wann (d.h. bei welchen Einfügungen) und warum Sie welche Rotation verwenden und zeigen Sie deren Auswirkung auf die Gestalt des Baumes.

c) Für die Navigation in einem binären Suchbaum sei zu einem Element, sofern dieses in einem Knoten des Baumes enthalten, der Geschwisterknoten dieses Knotens (d.h. das andere Kind des Elternknotens, sofern dieser existiert) zu bestimmen.

c1) Erläutern Sie zunächst mit Worten (und ggf. Skizzen), welche verschiedenen Fälle Sie behandeln müssen.

c2) Implementieren Sie dann **in Haskell** die Funktion:

```
brother :: (Show a, Ord a) => a ->  BinTree a -> (Bool, BinTree a)
```

Der Bool-Wert gibt dabei an, ob der Knoten mit dem als erstem Argument übergebenen Wert im Baum existiert. Falls ja, ist das zweite Element des Paars der gesuchte Knoten, sonst ist es immer der leere Baum.

Zur Erinnerung:

```
data (Ord a) => BinTree a = EmptyBT
                          | NodeBT a (BinTree a) (BinTree a)
    deriving (Show,Eq)

emptyTree = EmptyBT

inTree v' EmptyBT                       = False
inTree v' (NodeBT v lf rt) | v==v' = True
                           | v'<v  = inTree v' lf
                           | v'>v  = inTree v' rt

addTree v' EmptyBT                          = NodeBT v' EmptyBT EmptyBT
addTree v' (NodeBT v lf rt) | v'==v     = NodeBT v lf rt
                            | v' < v    = NodeBT v (addTree v' lf) rt
                            | otherwise = NodeBT v lf (addTree v' rt)
```

Aufgabe 3 [Gesamtpunktzahl: 8]

Gegeben seien die folgenden Definitionen der **Haskell-Funktionen** `takeWhile`, `dropWhile` und `span`:

```
takeWhile              :: (a -> Bool) -> [a] -> [a]
takeWhile p []          = []                                  -- tw1
takeWhile p (x:xs)                                            -- tw2
 | p x         = x : takeWhile p xs                           -- tw2.1
 | otherwise = []                                             -- tw2.2

dropWhile              :: (a -> Bool) -> [a] -> [a]
dropWhile p []          = []                                  -- dw1
dropWhile p xs@(x:xs')                                        -- dw2
 | p x         = dropWhile p xs'                              -- dw2.1
 | otherwise = xs                                             -- dw2.2

span                   :: (a -> Bool) -> [a] -> ([a],[a])
span p []               = ([],[])                             -- span1
span p xs@(x:xs')                                             -- span2
 | p x         = (x:ys, zs)
 | otherwise = ([],xs)
                        where (ys,zs) = span p xs'
```

Beweisen Sie durch strukturelle Induktion über die Listenlänge, dass **für alle endlichen Listen** `l` und **für beliebige Prädikate** `p` (mit passender Signatur) gilt:

```
span p l = (takeWhile p l, dropWhile p l)
```

Wichtiger Hinweis: Geben Sie bitte bei allen Umformungen in Beweisschritten jeweils die verwendeten Gleichungen oder sonstigen Begründungen an.

Aufgabe 4 [Gesamtpunktzahl: 10]

Zur Erinnerung: Wir haben den ADT Heap mit Hilfe sog. 'leftist trees' (dt. 'linkslastiger Bäume') implementiert. Ein Baum heisst (rangbasiert) 'linkslastig', wenn in jedem Knoten der Rang des linken Kinds grösser oder gleich dem Rang des rechten Kinds ist. Der **Rang** eines Knotens ist die kleinste Anzahl an Knoten, die vom aktuellen Knoten aus – diesen eingeschlossen – durchlaufen werden müssen, um zu einem leeren Knoten zu kommen.

Statt auf der Basis des Rangs (rangbasiert), lässt sich Linkslastigkeit in analoger Weise mit dem **'Gewicht'**, d.h. der Anzahl der Elemente, der Teilbäume definieren (gewichtsbasiert):

Ein Baum heisst *gewichtsbasiert 'linkslastig'*, wenn in jedem Knoten das Gewicht des linken Kinds grösser oder gleich dem Gewicht des rechten Kinds ist.

a) Zeigen Sie mit einem (einfachen) Beispiel, dass es rangbasierte linkslastige Bäume gibt, die aber nicht gewichtsbasiert linkslastig sind.

b) Zeigen Sie mit einem weiteren (einfachen) Beispiel, dass es andererseits auch gewichtsbasierte linkslastige Bäume gibt, die aber nicht rangbasiert linkslastig sind.

c) Überprüfen Sie, ob sich ihre Bäume aus a) bzw. b) jeweils durch eine einfache Vertauschung von linken und rechten Teilbäumen in die gewünschte 'richtige' Form bringen lassen (in a) also gewichtsbasiert, in b) rangbasiert linkslastig).

d) Wandeln Sie die folgende rangbasierte Implementation des ADT Heap so ab, dass sie dann gewichtsbasiert ist. Verändern oder ergänzen Sie die Implementation nur dort, wo Änderungen gemacht werden müssen.

```
data (Ord a) => Heap a = EmptyHP
                       | HP a Int (Heap a) (Heap a)
    deriving Show

emptyHeap = EmptyHP

heapEmpty EmptyHP = True
heapEmpty _       = False

findHeap EmptyHP      = error "findHeap:empty heap"
findHeap (HP x _ a b) = x

insHeap x h = merge (HP x 1 EmptyHP EmptyHP) h

delHeap EmptyHP      = error "delHeap:empty heap"
delHeap (HP x _ a b) = merge a b

rank :: (Ord a) => Heap a -> Int
rank EmptyHP      = 0
rank (HP _ r _ _) = r

makeHP :: (Ord a) => a -> Heap a -> Heap a -> Heap a
makeHP x a b | rank a >= rank b = HP x (rank b + 1) a b
             | otherwise        = HP x (rank a + 1) b a

merge ::(Ord a) =>  Heap a -> Heap a -> Heap a
merge h EmptyHP = h
merge EmptyHP h = h
merge h1@(HP x _ a1 b1) h2@(HP y _ a2 b2)
      | x <= y    = makeHP x a1 (merge b1 h2)
      | otherwise = makeHP y a2 (merge h1 b2)
```