Aufgabe 4 (Datenstrukturen): (4+4+4+10+4+10+9+10+10 = 65 Punkte)

In dieser Aufgabe betrachten wir Binärbaume, deren innere Knoten einzelne Werte eines Typs `a` speichern, während die Blätter jeweils einen Wert des Typs `b` speichern. Als Beispiel betrachten wir den Baum in Abbildung 1.
Sie dürfen jederzeit Hilfsfunktionen aus vorherigen Teilaufgaben verwenden, auch wenn Sie diese nicht selbst implementiert haben.

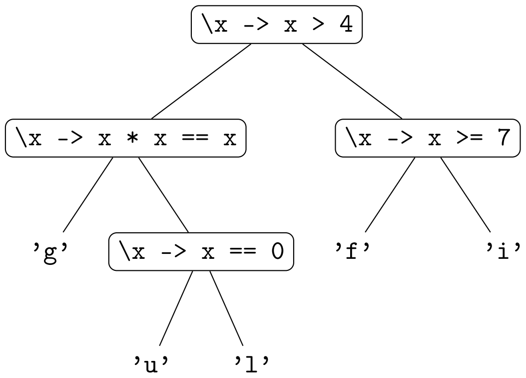

Abbildung 1: Ein beispielhafter Binärbaum.

Der Beispielbaum hat fünf Blätter mit den Zeichen `'g'`, `'u'`, `'l'`, `'f'`, `'i'` vom Typ `Char` und vier weiteren inneren Knoten, die eine einstellige Funktion vom Typ `Int -> Bool` enthalten.
Schreiben Sie zu jeder der im Folgenden zu implementierenden Funktionen auch eine Typdeklaration. Beachten Sie, dass auch die Wurzel eines Baums als innerer Knoten gilt.

a) Implementieren Sie in Haskell einen parametrisierten Datentyp `BinTree a b`, mit dem Binärbäume mit Werten vom Datentyp `a` in den inneren Knoten und mit Werten vom Datentyp `b` in den Blättern dargestellt werden können. Dabei soll sichergestellt werden, dass jeder innere Knoten genau zwei Nachfolger hat.

Hinweise:

- Ergänzen Sie `deriving Show` am Ende der Datentyp-Deklaration, damit `GHCi` die Bäume auf der Konsole anzeigen kann.

b) Definieren Sie den Beispielbaum aus Abb. 1 als `example`:

```
example :: BinTree (Int -> Bool) Char
example = ...
```

Hinweise:

- Importieren Sie das zusätzliche Modul `Text.Show.Functions` durch Einfügen der Zeile `import Text.Show.Functions` am Anfang ihrer Datei, damit `GHCi` auch Bäume wie in Abb. 1 auf der Konsole anzeigen kann, welche Funktionen enthalten.

c) Schreiben Sie die Funktion `countInnerNodes`, die einen Binärbaum vom Typ `BinTree a b` übergeben bekommt und die Anzahl der inneren Knoten als `Int` zurückgibt, d.h. die Anzahl der Knoten, die keine Blätter sind.

Für den Beispielbaum `example` soll der Aufruf `countInnerNodes example` also zu `4` auswerten.

d) Schreiben Sie die Funktion `decodeInt`. Diese bekommt als erstes Argument einen Binärbaum vom Typ `BinTree (Int -> Bool) b` und als zweites Argument einen Wert vom Typ `Int`. Der Rückgabewert dieser Funktion ist vom Typ `b`. Für einen Baum `bt` und eine Zahl `x` gibt `decodeInt bt x` das Zeichen zurück, an das man gelangt, wenn man ausgehend von der Wurzel in jedem inneren Knoten die Funktion vom Typ `Int -> Bool` des jeweiligen Knotens auf die Zahl `x` anwendet, wobei das linke Kind eines Knotens als Nachfolger gewählt wird, falls die Funktion zu `False` auswertet, und das rechte Kind gewählt wird, falls sie zu `True` auswertet. Wird `decodeInt` auf einem Blatt aufgerufen, wird dessen Wert zurückgegeben.

Für den Beispielbaum `example` soll der Aufruf `decodeInt example 0` also `'l'` zurückgeben.

e) Schreiben Sie die Funktion `decode`. Diese bekommt als erstes Argument einen Binärbaum vom Typ `BinTree (Int -> Bool) b` und als zweites Argument eine Liste vom Typ `[Int]` übergeben. Für einen Baum `bt` und eine Liste `xs` sucht `decode bt xs` zu jeder Zahl `x` aus der Liste `xs` den Wert `decodeInt bt x` und fügt die so erhaltenen Werte in einer Liste zusammen.

Für den Beispielbaum `example` soll der Aufruf `decode example [0,1,5,-4,7]` also den `String "lufgi"` zurückgeben.

f) Implementieren Sie eine Funktion `insertSorted`. Diese enthält als erstes Argument einen binären Suchbaum `bt` vom Typ `BinTree Int ()` und als zweites Argument eine Zahl vom Typ `Int`. Ein binärer Suchbaum ist ein Binärbaum, sodass für jeden inneren Knoten gilt, dass alle im linken Teilbaum gespeicherten Werte vom Typ `Int` echt kleiner sind als das im Knoten selbst gespeicherte Element, während alle im rechten Teilbaum gespeicherten Werte vom Typ `Int` nicht kleiner sind als der im Knoten selbst gespeicherte Wert. Der Typ `()` ist der Typ des leeren Tupels mit dem einzigen Wert `()::()` und dient hier dazu, zu verhindern, dass Werte vom Typ `Int` auch in den Blättern gespeichert werden. Ein binärer Suchbaum speichert also die in ihm enthaltenen Werte nur in den inneren Knoten, nicht jedoch in den Blättern. Der Aufruf `insertSorted bt x` soll nun den binären Suchbaum berechnen, welcher durch Einfügen des Werts `x` als neuen inneren Knoten aus `bt` entsteht.

Ein solcher binärer Suchbaum `bt` und das Ergebnis des Aufrufs `insertSorted bt 3` sind in Abb. 2 dargestellt.

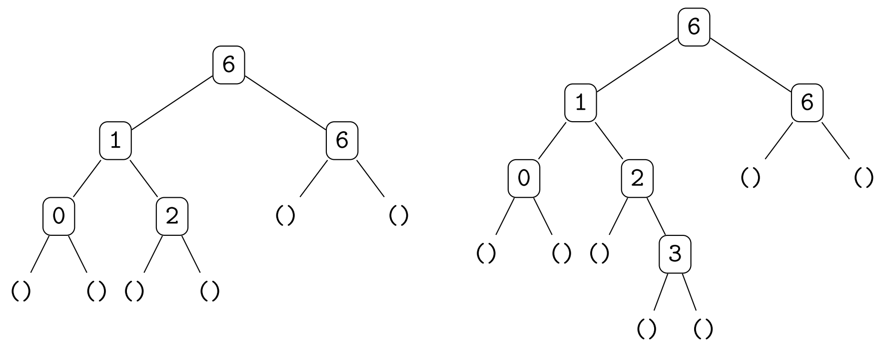

Abbildung 2: Ein binärer Suchbaum `bt` (links), welcher die Zahlen 0,1,2,6,6 speichert und das Ergebnis des Aufrufs `insertSorted bt 3` (rechts).

g) Schreiben Sie eine Funktion `treeSort`, welche eine Liste vom Typ `[Int]` durch geeignete Aufrufe der Funktion `insertSorted` sortiert.

Der Aufruf `treeSort [3,2,1,3,4,5]` soll zu der Liste `[1,2,3,3,4,5]::[Int]` auswerten.

h) Schreiben Sie eine Funktion `mergeTrees`, welche zwei binäre Suchbäume `bt1` und `bt2` vom Typ `BinTree Int ()` als Argumente erhält. Der Rückgabewert von `mergeTrees` ist ein binärer Suchbaum, welcher alle in `bt1` und `bt2` enthaltenen Werte speichert. Ist ein Wert `x` dabei n_1 mal in `bt1` und n_2 mal in `bt2` enthalten, so soll der Wert `x` anschließend $n_1 + n_2$ mal in dem Ergebnis des Aufrufs `mergeTrees bt1 bt2` enthalten sein.

Abb. 3 stellt zwei Suchbäume `bt1` und `bt2` sowie ein mögliches Ergebnis des Aufrufs `mergeTrees bt1 bt2` dar.

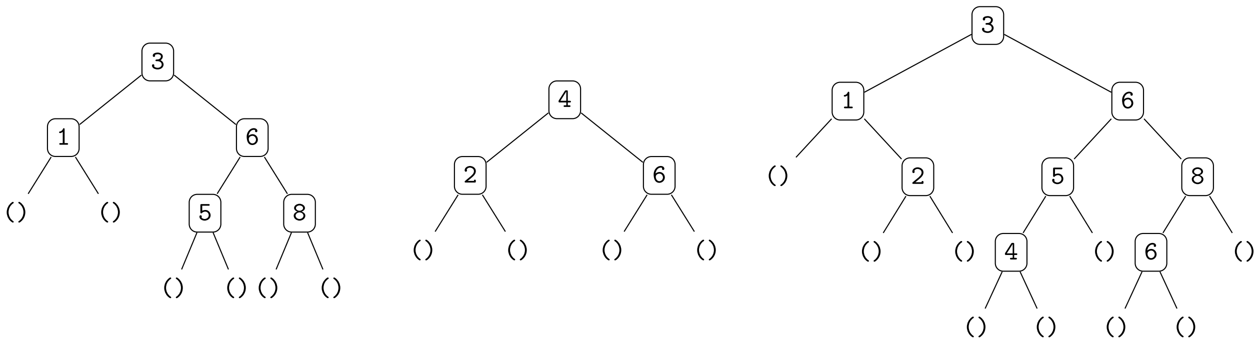

Abbildung 3: Ein Suchbaum `bt1` vom Typ `BinTree Int ()` (links), ein weiterer Suchbaum `bt2` vom Typ `BinTree Int ()` (Mitte) und ein mögliches Ergebnis des Aufrufs `mergeTrees bt1 bt2` (rechts).

Hinweise:

- Benutzen Sie die Funktion `insertSorted` aus Aufgabenteil f).

i) Implementieren Sie eine Funktion `numberOfOccurrences`, welche einen binären Suchbaum `bt` vom Typ `BinTree Int ()` als erstes Argument und einen Wert `x` vom Typ `Int` als zweites Argument erhält. Die Funktion berechnet dann die Anzahl der Vorkommen des Werts `x` im binären Suchbaum `bt`. Nutzen Sie dabei die Suchbaumeigenschaft von `bt` aus, um eine effiziente Vorgehensweise Ihrer Implementierung sicherzustellen.

Für den Suchbaum `bt` aus Abb. 2 wertet der Aufruf `numberOfOccurrences bt 6` zu `2` aus, da der Wert `6` zweimal im Suchbaum `bt` enthalten ist.

Lösung:

```
import Text.Show.Functions
```

a)
```
data BinTree a b = Node a (BinTree a b) (BinTree a b) | Leaf b
                   deriving Show
```

b)
```
example :: BinTree (Int -> Bool) Char
example = Node (\x -> x > 4)
              (Node (\x -> x * x == x)
                  (Leaf 'g')
                  (Node (\x -> x == 0) (Leaf 'u') (Leaf 'l'))
              )
              (Node (\x -> x >= 7) (Leaf 'f') (Leaf 'i'))
```

c)
```
countInnerNodes :: BinTree a b -> Int
countInnerNodes (Leaf _) = 0
countInnerNodes (Node _ bt1 bt2) =
  1 + countInnerNodes bt1 + countInnerNodes bt2
```

d)
```
decodeInt :: BinTree (Int -> Bool) b -> Int -> b
decodeInt (Leaf c)          x = c
decodeInt (Node f bt1 bt2) x | f x        = decodeInt bt2 x
                             | otherwise = decodeInt bt1 x
```

e)
```
decode :: BinTree (Int -> Bool) b -> [Int] -> [b]
decode _ []      = []
decode bt (x:xs) = decodeInt bt x : decode bt xs
```

f)
```
examplebt :: BinTree Int ()
examplebt = Node 6 (Node 1
                     (Node 0 (Leaf ()) (Leaf ()))
                     (Node 2 (Leaf ()) (Leaf ())))
                  (Node 6 (Leaf ()) (Leaf ()))

insertSorted :: BinTree Int () -> Int -> BinTree Int ()
insertSorted (Leaf ())    v = Node v (Leaf ()) (Leaf ())
insertSorted (Node x l r) v
  | v < x     = Node x (insertSorted l v) r
  | otherwise = Node x l (insertSorted r v)
```

g)
```
treeSort :: [Int] -> [Int]
treeSort keys = toList (buildTree keys)
  where
    buildTree []     = Leaf ()
    buildTree (x:xs) = insertSorted (buildTree xs) x
    toList (Leaf ())    = []
    toList (Node x l r) = toList l ++ x:toList r
```

h)
```
examplebt1 :: BinTree Int ()
examplebt1 = Node 3
               (Node 1 (Leaf ()) (Leaf ()))
               (Node 6
                 (Node 5 (Leaf ()) (Leaf ()))
                 (Node 8 (Leaf ()) (Leaf ())))

examplebt2 :: BinTree Int ()
examplebt2 = Node 4
               (Node 2 (Leaf ()) (Leaf ()))
               (Node 6 (Leaf ()) (Leaf ()))


mergeTrees :: BinTree Int () -> BinTree Int () -> BinTree Int ()
mergeTrees bt (Leaf ())    = bt
mergeTrees bt (Node x l r) =
  mergeTrees (mergeTrees (insertSorted bt x) l) r
```

i)
```
numberOfOccurrences :: BinTree Int () -> Int -> Int
numberOfOccurrences (Leaf ()) _ = 0
numberOfOccurrences (Node x l r) v
  | v < x     = numberOfOccurrences l v
  | v > x     = numberOfOccurrences r v
  | otherwise = 1 + numberOfOccurrences r v
```

Aufgabe 6 (Typen): (8 + 7 + 9 + 11 = 35 Punkte)

Bestimmen Sie zu den folgenden Haskell-Funktionen `f`, `g`, `h`, `i`, `j` und `k` den jeweils allgemeinsten Typ. Geben Sie den Typ an und begründen Sie Ihre Antwort. Gehen Sie hierbei davon aus, dass alle Zahlen den Typ `Int` haben, die Funktion `+` den Typ `Int -> Int -> Int` hat, die Funktion `head` den Typ `[a] -> a`, und die Funktion `==` den Typ `a -> a -> Bool` hat.

a)
```
f xs     y []     = []
f (x:xs) y (z:zs) = if x then y + z : f xs z zs else z : f xs z zs
```

b)
```
g x y = g (head y) y
g x y = (\x y -> x) x x
```

c)
```
h w x []     z = if x == [] then head z else w x []
h w x (y:ys) z = if w x ys then head z else y
```

d)
```
i x y z = x z y
j x = x
k = i j
```

Hinweise:

- Versuchen Sie diese Aufgabe ohne Einsatz eines Rechners zu lösen. Bedenken Sie, dass Sie in einer Prüfung ebenfalls keinen Rechner zur Verfügung haben.

Lösung:

a)
```
f xs     y []     = []
f (x:xs) y (z:zs) = if x then y + z : f xs z zs else z : f xs z zs
```

Da `f` drei Parameter bekommt, hat der Typ die Form `f :: a -> b -> c -> d`.

- Aus der ersten Gleichung folgt, dass `c` und `d` Listen sein müssen.
- Aus `(x:xs)` können wir folgern, dass auch `a` eine Liste ist.
- Aus `if x` folgt, dass `x` den Typ `Bool` hat.
- Weiter folgt aus `y + z`, dass `y` und `z` beide den Typ `Int` haben. Damit ergibt sich `f :: [Int] -> Int -> [Bool] -> [e]`.
- Aus `y + z : f xs z zs` bzw. `z : f xs z zs` folgt, dass der Ergebnistyp eine Liste mit Elementen des Typs von `z::Int` ist.

Somit ergibt sich insgesamt

`f :: [Bool] -> Int -> [Int] -> [Int]`.

b)
```
g x y = g (head y) y
g x y = (\x y -> x) x x
```

Da `g` zwei Argumente hat, nehmen wir den Typ `g :: a -> b -> c` an.

- Aus der ersten Gleichung folgt, dass `b` eine Liste mit Elementen des Typs `a` ist.
- Der Typ des Lambda-Ausdrucks `\x y -> x` ist `d -> e -> f`. Aus dem Ausdruck folgt weiterhin, dass die Typvariable `f` mit der Typvariablen `d` übereinstimmt.

- Da dieser Lambda-Ausdruck zuerst auf den ersten Parameter der Funktion angewendet wird, folgt aus der zweiten Gleichung, dass dieser Parametertyp dem Ergebnistyp entspricht: `g :: a -> [a] -> a`.

c)
```
h w x []     z = if x == [] then head z else w x []
h w x (y:ys) z = if w x ys then head z else y
```

Die Funktion `h` hat 4 Parameter. Wir gehen also erstmal vom Typ `h :: a -> b -> c -> d -> e` aus.

- Aus der ersten Gleichung erfahren wir, dass der zweite Parameter (`x`) einen Listentyp (`[f]`) hat.
- Aus `head z` folgt, dass der vierte Parameter ebenfalls einen Listentyp `[g]` hat und dass der Elementtyp `g` dem Ergebnistyp `e` der Funktion `h` entspricht.
- Aus `w x []` folgt, dass `w` eine Funktion mit 2 Parametern ist. Wir nehmen hier also eine Funktion `w :: b -> i -> e` an, da der erste Parameter von `w` der zweite Parameter der ersten Gleichung ist und der Ergebnistyp von `w` mit dem Ergebnistyp von `h` übereinstimmt. Der erste Parameter der Funktion `w` ist wie `x` also ebenfalls vom Typ `[f]`.
- Aus `if w x ys` in der zweiten Gleichung ergibt sich, dass der Rückgabetyp von `w`, und damit auch von `h`, `Bool` ist.
- Damit folgern wir, dass die Elemente der Liste `z` und `y` vom Typ `Bool` sein müssen. Das dritte Argument von `h` (und das zweite Argument von `w`) ist damit selbst vom Typ `[Bool]`.

Also ergibt sich insgesamt der Typ

`h :: ([f] -> -> [Bool] -> Bool) -> [f] -> [Bool] -> [Bool] -> Bool`.

d)
```
i x y z = x z y
j x = x
k = i j
```

Da die erste Gleichung für die Funktion `i` 3 Parameter besitzt, nehmen wir `i :: a -> b -> c -> d` an.

- Der Parameter `x` ist jedoch selbst eine Funktion, welche mit den beiden Parametern `z` und `y` von `i` aufgerufen wird und dessen Rückgabetyp dem Ergebnistyp von `i` entspricht. Wir nehmen daher `x :: c -> b -> d` an.

Damit ergibt sich `i :: (c -> b -> d) -> b -> c -> d`.

Mithilfe der zweiten Gleichung stellen wir fest, dass `j` einen Parameter erwartet, dessen Typ auch gleich wieder der Rückgabetyp von `j` selbst ist. Wir erhalten daher `j :: e -> e`.

Da nach der dritten Gleichung `k` keine Parameter erwartet, nehmen wir `k :: f` an, wobei `f` ebenfalls der Rückgabetyp von `i j` ist.

- Um den Typ `f` von `i j` zu bestimmen, stellen wir fest, dass die Typvariable `e` dem Typen `b -> d` entsprechen muss, denn `(->)` assoziert nach rechts. Damit ergibt sich auch, dass `e` und `c` identisch sein müssen.

Wir erhalten daher `k :: b -> (b -> d) -> d` als allgemeinsten Typ von `k`.