5. Aufgabenblatt zu Funktionale Programmierung vom 13.11.2013.
Fällig: 20.11.2013 / 27.11.2013 (jeweils 15:00 Uhr)

Themen: *Funktionen höherer Ordnung und Polymorphie*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe5.hs` ablegen. Sie sollen für die Lösung dieses Aufgabenblatts 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. "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) Zeigen Sie, 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` darstellen lässt. 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:

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

   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` in Ihrer Abgabedatei insgesamt wie folgt implementiert ist:

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

   Alle Funktionen sollen dabei als globale, nicht als lokale Rechenvorschriften implementiert sein, um auch einzeln testbar zu sein.

2. Wir betrachten folgenden algebraischen Datentyp für Bäume:

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

Schreiben Sie Haskell-Rechenvorschriften

(a) `tmap :: (a -> a) -> (Tree a) -> (Tree a)`

(b) `tzw :: (a -> a -> a) -> (Tree a) -> (Tree a) -> (Tree a)`

(c) `tfold :: (a -> a -> a -> a) -> a -> (Tree a) -> a`

die auf Bäumen das leisten, was die Funktionale `map`, `zipWith` und für assoziative Argumentfunktionen die Funktionale `foldl` und `foldr` auf Listen leisten. Zur Erinnerung seien hier die Implementierungen dieser drei Funktionale (eingeschränkt auf die Typvariable `a`) wiederholt:

```
map :: (a -> a) -> [a] -> [a]
map f [] = []
map f (x:xs) = f x : map f xs

zipWith :: (a -> a -> a) -> [a] -> [a] -> [a]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _      _      =  []

foldr :: (a -> a -> a) -> a -> [a] -> a
foldr _ v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Sie können bei der Implementierung voraussetzen, dass die Funktion `tfold` aussschließlich mit kommutativen und assoziativen Funktionsargumenten aufgerufen wird.

Folgende Beispiele zeigen das gewünschte Ein-/Ausgabeverhalten:

```
f1 = \x y z -> x+y+z
f2 = \x y z -> x*y*z

t1 = Null
t2 = Node 2 (Node 3 Nil Nil) (Tree 5 Nil Nil)
t3 = Node 2 (Node 3 (Node 5 Nil Nil) Nil) (Node 7 Nil Nil)

tmap (+1) t1 ->> Nil
tmap (+1) t2 ->> Node 3 (Node 4 Nil Nil) (Node 6 Nil Nil)
tmap (+1) t3 ->> Node 3 (Node 4 (Node 6 Nil Nil) Nil) (Node 8 Nil Nil)

tzw (+) t1 t2 ->> Nil
tzw (+) t2 t3 ->> Node 4 (Node 6 Nil Nil) (Node 12 Nil Nil)

tfold f1 0 t1 ->> 0
tfold f2 1 t1 ->> 1
fold f1 0 t2 ->> 10
tfold f1 0 t3 ->> 21
fold f2 1 t2 ->> 30
tfold f2 1 t3 ->> 210
```

Hinweis: Keine früheren Lösungen als Module importieren!

Wenn Sie zur Lösung einzelne Funktionen früherer Lösungen wiederverwenden möchten, so kopieren Sie diese unbedingt explizit in Ihre neue Programmdatei ein. Importieren schlägt im Rahmen der automatischen Programmauswertung fehl. Es wird nicht nachgebildet. Deshalb: Wiederverwendung ja, aber durch kopieren, nicht durch importieren!

Haskell Live

An einem der kommenden *Haskell Live*-Termine, der nächste ist am Freitag, den 22.11.2013, werden wir uns u.a. mit der Aufgabe *City-Maut* beschäftigen.

City-Maut

Viele Städte überlegen die Einführung einer City-Maut, um die Verkehrsströme kontrollieren und besser steuern zu können. Für die Einführung einer City-Maut sind verschiedene Modelle denkbar. In unserem Modell liegt ein besonderer Schwerpunkt auf innerstädtischen Nadelöhren. Unter einem *Nadelöhr* verstehen wir eine Verkehrsstelle, die auf dem Weg von einem Stadtteil A zu einem Stadtteil B in der Stadt passiert werden muss, für den es also keine Umfahrung gibt. Um den Verkehr an diesen Nadelöhren zu beeinflussen, sollen an genau diesen Stellen Mautstationen eingerichtet und Mobilitätsgebühren eingehoben werden.

In einer Stadt mit den Stadtteilen A, B, C, D, E und F und den sieben in beiden Richtungen befahrbaren Routen B–C, A–B, C–A, D–C, D–E, E–F und F–C führt jede Fahrt von Stadtteil A in Stadtteil E durch Stadtteil C. C ist also ein Nadelöhr und muss demnach mit einer Mautstation versehen werden.

Schreiben Sie ein Programm in Haskell oder in einer anderen Programmiersprache Ihrer Wahl, das für eine gegebene Stadt und darin vorgegebene Routen Anzahl und Namen aller Nadelöhre bestimmt.

Der Einfachheit halber gehen wir davon aus, dass anstelle von Stadtteilnamen von 1 beginnende fortlaufende Bezirksnummern verwendet werden. Der Stadt- und Routenplan wird dabei in Form eines Tupels zur Verfügung gestellt, das die Anzahl der Bezirke angibt und die möglichen jeweils in beiden Richtungen befahrbaren direkten Routen von Bezirk zu Bezirk innerhalb der Stadt. In Haskell könnte dies durch einen Wert des Datentyps `CityMap` realisiert werden:

```
type Bezirk     = Integer
type AnzBezirke = Integer
type Route      = (Bezirk,Bezirk)
newtype CityMap = CM (AnzBezirke,[Route])
```

Gültige Stadt- und Routenpläne müssen offenbar bestimmten Wohlgeformtheitsanforderungen genügen. Überlegen Sie sich, welche das sind und wie sie überprüft werden können, so dass Ihr Nadelöhrsuchprogramm nur auf wohlgeformte Stadt- und Routenpläne angewendet wird.