5. Aufgabenblatt zu Funktionale Programmierung vom 14.11.2012.
Fällig: 21.11.2012 / 28.11.2012 (jeweils 15:00 Uhr)

Themen: *Funktionen und Funktionen höherer Ordnung auf algebraischen Datentypen*

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. Wir betrachten folgenden algebraischen Datentyp für Bäume:

```
data Tree  = Null | Tree Label Tree Tree deriving (Eq,Show)
type Label = Integer
```

Schreiben Sie Haskell-Rechenvorschriften

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

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

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

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 Listen mit Elementtyp `Label`) wiederholt:

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

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

foldr :: (Label -> Label -> Label) -> Label -> [Label] -> Label
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 = Tree 2 (Tree 3 Null Null) (Tree 5 Null Null)
t3 = Tree 2 (Tree 3 (Tree 5 Null Null) Null) (Tree 7 Null Null)

tmap (+1) t1 ->> Null
tmap (+1) t2 ->> Tree 3 (Tree 4 Null Null) (Tree 6 Null Null)
tmap (+1) t3 ->> Tree 3 (Tree 4 (Tree 6 Null Null) Null) (Tree 8 Null Null)

tzw (+) t1 t2 ->> Null
tzw (+) t2 t3 ->> Tree 4 (Tree 6 Null Null) (Tree 12 Null Null)
```

```
tfold f1 0 t1 ->> 0
tfold f2 1 t1 ->> 1
tfold f1 0 t2 ->> 10
tfold f1 0 t3 ->> 17
tfold f2 1 t2 ->> 30
tfold f2 1 t3 ->> 210
```

2. Paul Erdös (1913-1996) ist einer der berühmtesten Mathematiker des 20. Jahrhunderts. Wissenschaftler, die gemeinsam mit Paul Erdös eine Arbeit veröffentlicht haben, sind hoch angesehen. Das Beste, was Mathematiker erreichen konnten, die nicht unmittelbar gemeinsam mit Paul Erdös arbeiten und veröffentlichen konnten, war, gemeinsam mit einem Co-Autoren von Paul Erdös zu forschen und zu publizieren. Dies hat zur Einführung der sog. *Erdös-Zahlen* geführt, die inzwischen viele Wisschenschaftler (meist aus Spaß) in ihrem Lebenslauf anführen.

   Dabei gilt: Die Erdös-Zahl von Wissenschaftlern, die mit Paul Erdös mindestens eine gemeinsame Veröffentlichung haben, ist 1. Wissenschaftler, die mindestens eine gemeinsame Veröffentlichung mit einem Wissenschaftler haben, dessen Erdös-Zahl 1 ist, haben selbst die Erdös-Zahl 2, usw. Wissenschaftler ohne einen (Veröffentlichungs-) Bezug zu Paul Erdös erhalten für die Zwecke dieser Aufgabe die Erdös-Zahl -1; Paul Erdös selbst die Erdös-Zahl 0.

   Schreiben Sie eine Haskell-Rechenvorschrift `erdosNum :: Database -> Scientist -> ErdosNumber`, die für jeden Wissenschaftler s die zugehörige Erdös-Zahl gemäß der in der Datenbank db gespeicherten Publikationseinträge berechnet.

   Dabei gilt:

   ```
   type ErdosNumber = Integer
   data Scientist = Sc Initial SurName
   type Initial = Char
   type SurName = String
   type Author = Scientist
   newtype Database = Db [([Author],PaperTitle)]
   type PaperTitle = String
   ```

   Folgende Beispiele zeigen das gewünschte Ein-/Ausgabeverhalten (Umlaute wie "ö"werden ersetzt durch den entsprechenden Grundbuchstaben, "ö" etwa durch "o", usw.):

   ```
   db = Db [([Sc 'M' "Smith",Sc 'G' "Martin",Sc 'P' "Erdos"],"Newtonian Forms of Prime Factors"),
            ([Sc 'P' "Erdos",Sc 'W' "Reisig"],"Stuttering in Petri Nets") ,
            ([Sc 'M' "Smith",Sc 'X' "Chen"],"First Order Derivates in Structured Programming"),
            ([Sc 'T' "Jablonski",Sc 'Z' "Hsueh"],"Selfstabilizing Data Structures"),
            ([Sc 'X' "Chen",Sc 'L' "Li"],"Prime Numbers and Beyond")]

   erdosNum db (Sc 'P' "Erdos") ->> 0
   erdosNum db (Sc 'M' "Smith") ->> 1
   erdosNum db (Sc 'X' "Chen") ->> 2
   erdosNum db (Sc 'L' "Li") ->> 3
   erdosNum db (Sc 'Z' "Hsueh") ->> (-1)
   erdosNum db (Sc 'K' "Tochterle") ->> (-1)
   ```

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 16.11.2012, 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.