6. Aufgabenblatt zu Funktionale Programmierung vom 25.11.2015.
Fällig: 09.12.2015 / 16.12.2015 (jeweils 15:00 Uhr)

Themen: *Typklassen und Überladung, Funktionen höherer Ordnung und Graphen*

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. Sie sollen für die Lösung dieses Aufgabenblatts also 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 noch einmal den Datentyp `Nat` aus Aufgabenteil 1 von Aufgabenblatt 5 zur Modellierung des endlichen Ausschnitts natürlicher Zahlen von 0 bis $maxNat$ und gehen davon aus, dass dieser Typ entsprechend Aufgabenteil 1 von Aufgabenblatt 5 Instanz der Klassen `Eq`, `Ord`, `Show` und `Num` ist.

```
data Zeichen = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P
             | Q | R | S | T | U | V | W | X | Y | Z
                deriving (Eq,Ord,Show,Read,Enum,Bounded)
data Ziffer  = Null | Eins | Zwei | Drei | Vier | Fuenf | Sechs | Sieben
             | Acht | Neun deriving (Eq,Ord,Show,Read,Enum,Bounded)
newtype Nat  = Nat ((Zeichen,Zeichen,Zeichen),(Ziffer,Ziffer,Ziffer))
```

   Im Haskell-Prelude sind neben den Typklassen `Eq`, `Ord`, `Show` und `Num` auch die Typklassen `Bounded`, `Enum`, `Real` und `Integral` zusammen mit den Signaturen der Rechenvorschriften definiert, die für jede Instanz dieser Typklassen implementiert sein müssen. Da für einige Rechenvorschriften bereits Proto-Implementierungen in den Klassendefinitionen angegeben sind, reicht es bei Instanzbildungen oft aus, für eine Teilmenge der in einer Klasse vorgesehenen Rechenvorschriften typspezifische Implementierungen bei der Instanzbildung anzugeben, wenn sich die so aus den Proto-Implementierungen automatisch ergebenden Vollimplementierungen das gewünschte Verhalten aufweisen. Machen Sie davon bei den folgenden Teilaufgaben Gebrauch und implementieren Sie lediglich die Mininalmenge von Rechenvorschriften und übernehmen überall sonst, wo möglich, die Proto-Implementierung unverändert, d.h. überschreiben Sie diese nicht.

   Machen Sie den Datentyp `Nat` explizit, d.h. ohne Rückgriff auf eine *deriving*-Klausel, zu je einer Instanz der Typklassen

   a) `Bounded` (Minimale Vervollständigung: Festlegung der Werte für `minBound` und `maxBound`.)
   b) `Enum` (Minimale Vervollständigung: Implementierung der Rechenvorschriften `toEnum` und `fromEnum`.)
   c) `Real` (Minimale Vervollständigung: Implementierung der Rechenvorschrift `toRelational`.)
   d) `Integral` (Minimale Vervollständigung: Implementierung der Rechenvorschriften (`quotRem` und `toInteger`.)

   Die Bedeutung aller in den verschiedenen Klassen vorgesehenen Rechenvorschriften soll — soweit nicht anders angegeben, etwa für die unter- und überlauffreie Addition, Multiplikation und Subtraktion für Werte des Datentyps `Nat` — in analoger Weise wie für den ebenfalls eingeschränkten Zahlbereich `Int` sein. Im einzelnen gilt folgende weitere Abweichung:

   - Die Rechenvorschrift `toEnum` liefert für echt negative Argumente das Resultat 0 als Wert vom Typ `Nat`.

2. Wir betrachten folgenden algebraischen Datentyp für ternäre Bäume:

```
data Num a => Tree a = Nil
                     | Node (a -> a) (a -> Bool) [a] (Tree a) (Tree a) (Tree a)
```

   Schreiben Sie eine Haskell-Rechenvorschrift für einen sich "selbst modifizierenden" Ternärbaum

```
smt :: Num a => (Tree a) -> (Tree a)
```

die folgendes leistet:

Angewendet auf einen Baum t, liefert die Funktion `smt` einen zu t strukturgleichen Baum t' zurück, wobei jeder Knoten in t' mit derselben Transformationsfunktion f für Marken und demselben Prädikat p auf Marken wie der entsprechende Knoten in t markiert ist, die Liste l von Marken jedoch durch eine i.a. von l verschiedene Liste l' ersetzt ist. Dabei entsteht die Liste l' aus l dadurch, dass die Elemente von l ähnlich wie bei dem Funktional `map` auf Listen von links nach rechts durchgegangen werden und jedes Element, das die Eigenschaft p erfüllt mittels f transformiert und in die Ergebnisliste aufgenommen wird. Elemente aus l, die p nicht erfüllen, werden verworfen.

3. Wir betrachten erneut den Datentyp für Ternärbäume aus der vorigen Teilaufgabe, sowie folgenden vereinfachten Ternärbaumtyp:

```
data Num a => STree a = SNil
                      | SNode [a] (STree a) (STree a) (STree a) deriving (Eq,Show)
```

   - Schreiben Sie eine Haskell-Rechenvorschrift

     ```
     t2st :: Num a => (Tree a) -> (STree a)
     ```

     Angewendet auf einen Baum t vom Typ `(Tree a)` liefert `t2st` als Resultat einen Baum st vom Typ `(STree a)`, in dem die Knotenmarkierungen funktionalen Typs aus t fehlen und der ansonsten in Struktur und Markierung mit t übereinstimmt.

   - Schreiben Sie in Anlehnung an die Faltungsfunktionale auf Listen eine Haskell-Rechenvorschrift

     (a) `tsum :: Num a => STree a -> a`

     die angewendet auf einen Baum die Summe der Summen aller Knotenmarkierungen berechnet, d.h. an jedem Knoten wird die Summe der Listenelemente berechnet, die dann summativ in die Gesamtsumme eingeht.

     Schreiben Sie weiters eine Haskell-Rechenvorschrift

     b) `tdepth :: Num a => STree a -> Integer`

     die angewendet auf einen Baum, dessen Tiefe bestimmt. Dem Baumwert `SNil` ist die Tiefe 0 zugeordnet.

4. Wir betrachten folgende Datenstruktur zur Darstellung ungerichteter Graphen $G = (N, E)$ mit Knotenmenge N und Kantenmenge $E \subseteq N \times N$ in Haskell:

```
type Node     = Integer
type Edge     = (Node,Node)
type Source   = Node
type Sink     = Node
newtype Graph = Gph [(Source,[Sink])] deriving (Eq,Show)
```

   Die obige Adjazenzlistendarstellung eines Graphen legt seine Knotenmenge implizit fest. Ein Knoten ist Element der Knotenmenge des Graphen, wenn er in einem Eintrag der Liste `[(Source,[Sink])]` als Quelle oder Endpunkt einer in der Quelle beginnenden Kante aufgeführt wird. Zwischen zwei Knoten eines Graphen gibt es entweder eine ungerichtete Kante oder keine Kante. Eine ungerichtete Kante zwischen zwei Knoten m und n gibt es stets, wenn n in einer der Endpunktlisten mit m als Quelle angeführt ist oder/und umgekehrt. Die Adjazenzlistendarstellung ist also i.a. nicht minimal in dem Sinn, dass jeder Knoten des Graphen genau einmal als Quelle in einem Listeneintrag vorkommt oder dass von dort erreichbare Knoten nur einmal in der Endknotenliste auftauchen. Umgekehrt ist es nicht erforderlich, wenn n in einer Endknotenliste zur Quelle m aufgeführt ist, dass umgekehrt m in einer Endknotenliste zur Quelle n aufgeführt sein muss.

   Wir sagen, dass eine Kante $e = (i, j)$ in *Normaldarstellung* vorliegt, wenn gilt: $i \leq j$.

   (a) Schreiben Sie eine Haskell-Rechenvorschrift

   ```
   ne :: Graph -> ([Node],[Edge])
   ```

Angewendet auf einen Graphen G liefert `ne` alle Knoten und Kanten aus G. Dabei sollen im Resultat Knoten und Kanten echt aufsteigend sortiert sein und Kanten in Normaldarstellung angegeben sein. Für Knoten gilt: Ein Knoten i ist echt kleiner als ein Knoten j, wenn gilt: $i < j$. Für Kanten gilt: Eine Kante (i, j) ist echt kleiner als eine Kante (k, l), wenn gilt:

$$i < k \vee (i = k \wedge j < l)$$

(b) Sei $M \subseteq N$ eine Teilmenge der Knotenmenge N von G. M heißt *unabhängig*, wenn keine zwei Knoten aus M durch eine Kante verbunden sind. M heißt Knotenhülle von G, wenn mindestens ein Endpunkt jeder Kante e aus G aus M ist.

Schreiben Sie zwei Haskell-Wahrheitswertrechenvorschriften

i. `independent :: Graph -> [Node] -> Bool`

ii. `cover :: Graph -> [Node] -> Bool`

die, angewendet auf einen Graphen G und eine Knotenliste L berechnen, ob die in L enthaltene Menge von Knoten unabhängige Teilmenge der Knotenmenge von G ist bzw. Knotenhülle der Menge von Knoten von G ist. Enthält L einzelne Elemente mehrfach, so wird dies genauso behandelt als kämen diese Elemente nur genau einmal in L vor. Sind die in L enthaltenen Knoten keine Teilmenge der Knotenmenge von G liefern die Funktionen `independent` und `cover` den Wahrheitswert `False`.

Wichtig: Wenn Sie einzelne Rechenvorschriften aus früheren Lösungen für dieses oder spätere Aufgabenblätter wieder verwenden möchten, so kopieren Sie diese in die neue Abgabedatei ein. Ein `import` schlägt für die Auswertung durch das Abgabeskript fehl, weil Ihre frühere Lösung, aus der importiert wird, nicht mit abgesammelt wird. Deshalb: Kopieren statt importieren zur Wiederwendung!

Haskell Live

Der nächste *Haskell Live*-Termin findet am Freitag, den 27.11.2015, statt.