7. Aufgabenblatt zu Funktionale Programmierung vom Mi, 28.11.2018.
Fällig: Mi, 05.12.2018 (15:00 Uhr)

Themen: *Rechnen mit Funktionen, Funktionen als Argumente und Resultate, Typklassen*

Zur Frist der Zweitabgabe: Siehe „Hinweise zu Organisation und Ablauf der Übung“ auf der Homepage der LVA.

Aufgabe

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe7.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 benötigen, mit ihren Typdeklarationen und kommentieren Sie Ihre Programme aussagekräftig. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und (Wertvereinbarungen für) Konstanten.

1. Eine Liste ganzer Zahlen `[5,2,-4,1,9,8,9]` kann als kompakte Darstellung der Menge $\{(0,5),(1,2),(2,-4),(3,1),(4,9),(5,8),(6,9)\}$ aufgefasst werden. Dabei ist das erste Element jedes Paares durch seine Indexposition in der Liste, sein zweites Element durch den zugehörigen Listenwert gegeben. Die Menge der Paare $\{(0,5),(1,2),(2,-4),(3,1),(4,9),(5,8),(6,9)\}$ kann ihrerseits als Darstellung der Funktion $f : IN_0 \to \mathbb{Z}$ von der Menge der natürlichen Zahlen IN_0 in die Menge der ganzen Zahlen $\mathbb{Z}$ aufgefasst werden, als sog. *Graph* der folgendermaßen definierten Funktion f:

$$\forall\, n \in IN_0.\ f(n) =_{df} \begin{cases} 5 & \text{falls } n = 0 \\ 2 & \text{falls } n = 1 \\ -4 & \text{falls } n = 2 \\ 1 & \text{falls } n = 3 \\ 9 & \text{falls } n = 4 \\ 8 & \text{falls } n = 5 \\ 9 & \text{falls } n = 6 \\ 0 & \text{sonst} \end{cases}$$

Schreiben Sie Haskell-Rechenvorschriften:

```
fkt_zu_liste   :: Funktion -> Liste
fkt_zu_graph   :: Funktion -> Graph
liste_zu_graph :: Liste    -> Graph
liste_zu_fkt   :: Liste    -> Funktion
graph_zu_liste :: Graph    -> Liste
graph_zu_fkt   :: Graph    -> Funktion
```

über den Typ(synonym)en:

```
type Nat0     = Int
type Zett     = Int             -- Zusicherungen fuer Listen, Graphen, Funktionen:
type Liste    = [Zett]          -- Ausschl. 0-freie Listen.
type Graph    = [(Nat0,Zett)]   -- Ausschl. '0-freie-zweite-Paarelement'-Listen plus s.u.
type Funktion = (Nat0 -> Zett)  -- Ausschl. 0-determinierte Funktionen
```

die zwischen diesen drei Darstellungen von Funktionen vermitteln und unter den nachstehenden **Zusicherungen** für Listen-, Graph- und Funktionswerte folgende Invarianzeigenschaften erfüllen:

```
(fkt_zu_liste   . liste_zu_fkt)    =  λx. x
(fkt_zu_graph   . graph_zu_fkt)    =  λx. x
(liste_zu_graph . graph_zu_liste)  =  λx. x
(liste_zu_fkt   . fkt_zu_liste)    =  λx. x
(graph_zu_liste . liste_zu_graph)  =  λx. x
(graph_zu_fkt   . fkt_zu_graph)    =  λx. x
```

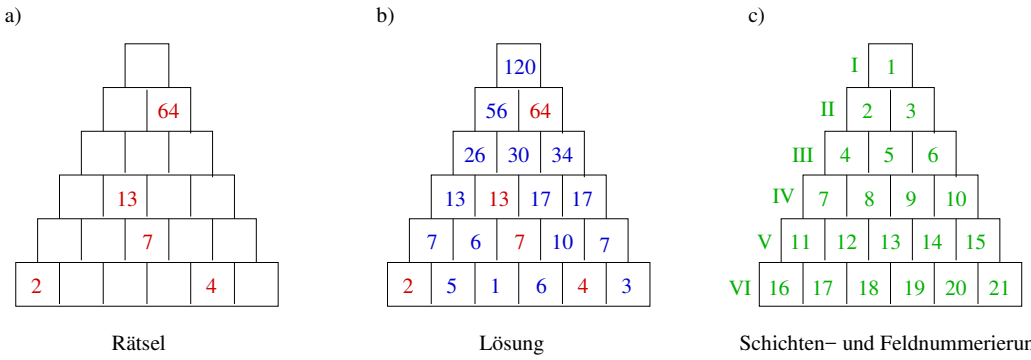
Zusicherungen: Für Listen-, Graph- und Funktionswerte gelten folgende Zusicherungen:

- Listenwerte sind 0-*frei*, d.h. 0 ist in keinem Listenwert enthalten.
- Graphwerte sind
  - 0-*frei* in den jeweils zweiten Paarkomponenten, d.h. (n, z) Element eines Graphwerts impliziert $z \neq 0$.
  - *paarweise verschieden* in den ersten Paarkomponenten, d.h. $(n, z), (n', z')$ Elemente eines Graphwerts impliziert $n \neq n'$.
  - *lückenfrei* in den jeweils ersten Paarkomponenten, d.h. n erste Komponente eines Elements eines Graphwerts impliziert n' ist ebenfalls erste Komponente eines Elements des Graphwerts für alle n' mit $0 \leq n' < n$.
  - *aufsteigend* nach den Werten der ersten Paarkomponenten angeordnet, d.h. $(n, z), (n', z')$ Elemente eines Graphwerts mit $n < n'$ impliziert (n, z) steht weiter links als (n', z').
- Funktionswerte sind 0-*determiniert*, d.h. für jeden Funktionswert f gibt es ein $n_0 \in I\!N_0$ mit $f(m) \neq 0$ für alle $m < n_0$ und $f(m) = 0$ für alle $m \geq n_0$.

Weiters gilt: Die leere Liste entspricht dem leeren Graphen entspricht der konstanten Funktion mit Funkionswert 0 ($\lambda n.\ 0$) und umgekehrt.

2. *Pyramidal*, ein Rätsel in Zahlen. Fülle die leeren Felder (s. Abbildung a)) so aus, dass

   - in der untersten Pyramidenschicht eine Permutation der Zahlen von 1 bis n steht, wobei n die Zahl der Felder der untersten Schicht angibt.
   - die Zahl in jedem Feld, das nicht in der untersten Schicht liegt, die Summe der in der unmittelbar tieferen Schicht schräg darunter stehenden Zahlen ist (s. Abbildung b)).



Rätsel | Lösung | Schichten− und Feldnummerierung

Wir nummerieren die *Schichten* eines pyramidalen Rätsels von oben nach unten (römisch), die Felder in den einzelnen Schichten von links nach rechts (arabisch) (s. Abbildung c)). Die Zahl der Schichten in einem pyramidalen Rätsel bezeichnen wir als *Schichtungstiefe* der Pyramide.

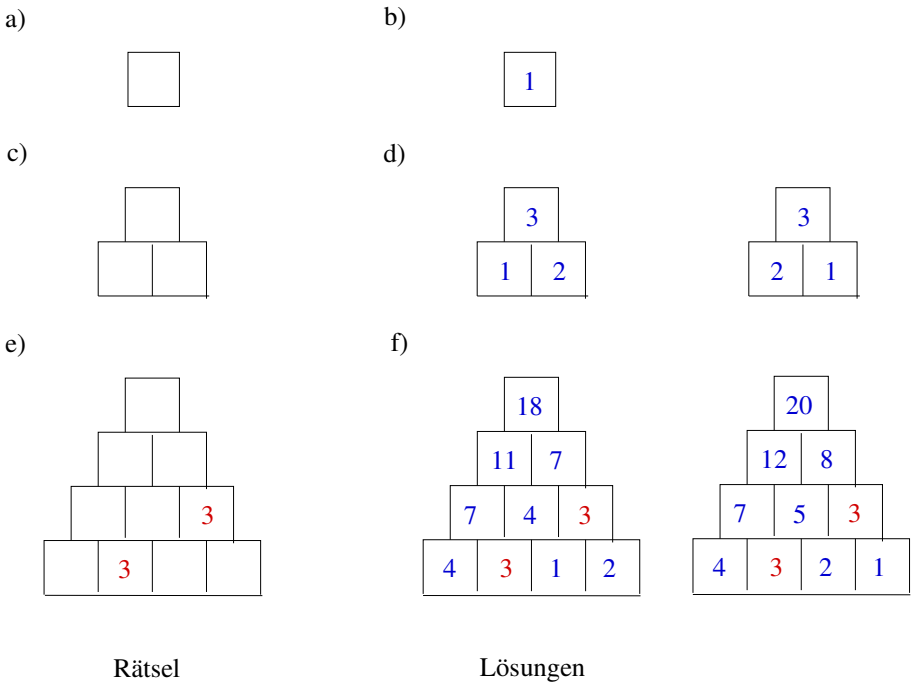
In dieser Aufgabe wollen wir eine Haskell-Rechenvorschrift `loese` zur Lösung pyramidaler Rätsel schreiben, wenn sie eine Lösung haben. Zur Modellierung führen wir folgende Datenstrukturen ein:

```
type Nat0 = Int
type Nat1 = Int

data Schicht            = Eins | Zwei | Drei | Vier | Fuenf | Sechs
                          deriving (Eq,Ord,Enum,Show)
type Schichtungstiefe   = Schicht
type Feldnummer         = Nat1
type Feldinhalt         = Nat0       -- Wert 0 repraesentiert freies, unbelegtes Feld

type Feldbelegung       = Feldnummer -> Feldinhalt
type Pyramidenraetsel   = Feldbelegung
type Pyramidenloesung   = Feldbelegung
type Pyramidenloesungen = [Pyramidenloesung]

loese :: Schichtungstiefe -> Pyramidenraetsel -> Pyramidenloesungen
```

Hat ein Pyramidenrätsel keine Lösung, so ist die Lösungsliste leer; hat ein Rätsel mehr als eine Lösung, so enthält die Lösungsliste jede dieser Lösungen in beliebiger Reihenfolge genau einmal.

Folgende *Aufrufbeispiele* illustrieren das Verhalten der Funktion `loese`:

```
fb1 = \n -> 0
fb2 = \n -> if (n==6 || n==8) then 3 else 0

lsg1 = \n -> if n==1 then 1 else 0

lsg2 = \n -> case n of 1 -> 3
                       2 -> 1
                       3 -> 2
                       _ -> 0

lsg3 = \n -> case n of 1 -> 3
                       2 -> 2
                       3 -> 1
                       _ -> 0

lsg4 = \n -> case n of  1 -> 18
                        2 -> 11
                        3 ->  7
                        4 ->  7
                        5 ->  4
                        6 ->  3
                        7 ->  4
                        8 ->  3
                        9 ->  1
                       10 ->  2
                        _ ->  0

lsg5 = \n -> case n of  1 -> 20
                        2 -> 12
                        3 ->  8
                        4 ->  7
                        5 ->  5
                        6 ->  3
                        7 ->  4
                        8 ->  3
                        9 ->  2
                       10 ->  1
                        _ ->  0

loese Eins fb1 ->> [lsg1]          -- vgl. Abbildungen a) und b).
loese Zwei fb1 ->> [lsg2,lsg3]     -- vgl. Abbildungen c) und d).
loese Vier fb2 ->> [lsg4,lsg5]     -- vgl. Abbildungen e) und f).
```



Ohne Abgabe: Welche Komplexität hat die Funktion `loese` gemessen in Anzahl der Additionen (zu Einheitskosten), die sie zur Berechnung aller Lösungen in Abhängigkeit der Schichttiefe n, $n \geq 1$, eines Pyramidenrätsels im schlechtesten Fall durchführt (vgl. Kap. 7.4 der Vorlesung)? Analog: Welche Komplexität hat die Funktion `determinante` von Aufgabenblatt 5 in Anzahl der Multiplikationen und Additionen (zu Einheitskosten) im schlechtesten Fall?

3. Schreiben Sie eine Haskell-Rechenvorschrift

```
zeige_loesungen :: Schichtungstiefe -> Pyramidenloesungen -> [[[Feldinhalt]]]
```

die angewendet auf eine Schichtungstiefe und eine Liste von Pyramidenrätsellösungen die Lösungen in Form von Listen von Listen von Feldinhalten in der Reihenfolge der Schichten und Feldnummern ausgibt.

Aufrufbeispiele:

```
zeige_loesungen Eins (loese Eins fb1) ->> [[[1]]]
zeige_loesungen Zwei (loese Zwei fb1) ->> [[[3],[1,2]],[[3],[2,1]]]
zeige_loesungen Vier (loese Vier fb2)
  ->> [[[18],[11,7],[7,4,3],[4,3,1,2]],[[20],[12,8],[7,5,3],[4,3,2,1]]]
```

Für die Implementierung von `zeige_loesungen` dürfen Sie davon ausgehen, dass bei Aufrufen die Pyramidenlösungen des zweiten Arguments für mindestens so viele Feldnummern definiert sind, wie es aufgrund des Wertes des Schichtungstiefenarguments erforderlich ist. Ist das nicht der Fall, so bleibt das Verhalten von `zeige_loesungen` unbestimmt und kann beliebig sein.

4. Um eine schönere Ausgabe zu erzielen, führen wir einen neuen Datentyp `Schoen` zusammen mit einer 'mache'-Funktion ein:

   ```
   newtype Schoen = Schoen [[[Feldinhalt]]]
   mache :: [[[Feldinhalt]]] -> Schoen
   ```

   Machen Sie den Typ `Schoen` zu einer Instanz der Typklasse `Show`. Die Überführung in eine Zeichenreihe soll dabei wie durch folgende Beispiele illustriert sein, wobei die Funktion `mache` einen `[[[Feldinhalt]]]`-Wert in offensichtlicher Weise in einen `Schoen`-Wert umwandelt:

   ```
   show (mache [[[1]]])                    ->> "<[1]>"
   show (mache [[[3],[1,2]],[[3],[2,1]]]) ->> "<[3|1,2],[3|2,1]>"
   show (mache [[[18],[11,7],[7,4,3],[4,3,1,2]],[[20],[12,8],[7,5,3],[4,3,2,1]]])
     ->> "<[18|11,7|7,4,3|4,3,1,2],[20|12,8|7,5,3|4,3,2,1]>"
   ```

5. **Wir ergänzen wertungsmäßig Teilaufgabe 1 von Aufgabenblatt 3 um die Aufgabe Schluchtenfinder 2.0 (für Aufgabenblatt 7 sind deshalb 100+25 Punkte zu erreichen):** Zwischen zwei verschiedenen ganzen Zahlen klafft eine *Schlucht* der Breite n, $n \in IN_0$, wenn zwischen ihnen n ganze Zahlen fehlen.

   Ist l eine Liste ganzer Zahlen, so wollen wir alle aufsteigend sortierten, aufsteigend angeordneten Teillisten maximaler Länge ≥ 2 aller Zahlen aus l bestimmen, so dass für jede der Teillisten gilt:

   - Die Elemente der Teilliste sind paarweise verschieden und aufsteigend angeordnet.
   - Sind m und m' das kleinste und größte Element der Teilliste, so sind alle Elemente k aus l mit $m < k < m'$ ebenfalls Element der Teilliste.
   - Zwischen je zwei benachbarten Elementen der Teilliste klafft eine Schlucht der Breite n.

   Schreiben Sie eine Haskell-Rechenvorschrift `schluchtenfinde_2_0` über den Typsynonymen `Nat0`, `Zett`, `Terrain`, `Schluchtenbreite` und `Schluchtenliste`, die diese Aufgabe erfüllt:

   ```
   type Nat0             = Int
   type Zett             = Int
   type Terrain          = [Zett]
   type Schluchtenbreite = Nat0
   type Schluchtenliste  = [Terrain]

   schluchtenfinder_2_0 :: Terrain -> Schluchtenbreite -> Schluchtenliste
   ```

   Aufrufbeispiele:

   ```
   schluchtenfinder_2_0 [11,4,6,2,42,3,7,13,1,42,2,10] 0  ->> [[1,2],[2,3,4],[6,7],[10,11]]
   schluchtenfinder_2_0 [11,4,6,2,42,3,7,13,1,42,2,10] 1  ->> [[4,6],[11,13]]
   schluchtenfinder_2_0 [14,25,2,17,29,6,16,8,29,21,12] 3 ->> [[2,6],[8,12],[17,21,25,29]]
   ```

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 alte Lösung, aus der importiert wird, nicht mit abgesammelt wird. Deshalb: Kopieren statt importieren zur Wiederwendung!

Haskell Live

An einem der kommenden *Haskell Live*-Termine, der nächste ist am Freitag, den 30.11.2018, werden wir uns u.a. mit der Aufgabe *Wer bekommt den Job?* beschäftigen.

Wer bekommt den Job?

Im Oktober 2004 hat Google mit einer Anzeigenkampagne zu Bewerbungen eingeladen. Erfolgreich konnten nur Bewerbungen sein, denen die Lösung einer Reihe von Aufgaben beigelegt war. Eine dieser Aufgaben lautete im Wortlaut wie folgt:

> *Consider a function which, for a given whole number n, returns the number of ones required when writing out all numbers between* 0 *and n. For example, f(13)=6. Notice that* $f(1) = 1$. *What is the next largest n such that* $f(n) = n$*?*

Von einer Funktion identisch, also auf sich selbst abgebildete Argumentwerte heißen *Fixpunkte* der Funktion. 1 ist also ein Fixpunkt von f; 1 ist sogar der kleinste echt positive Fixpunkt von f. Die Lösung der von Google gestellten Aufgabe ist es also, den kleinsten von 1 verschiedenen echt positiven Fixpunkt von f zu bestimmen. In der Folge nennen wir die positiven Fixpunkte von f *Google-Zahlen*.

Schreiben Sie Haskell-Rechenvorschriften für folgende Aufgaben: Eine Funktion `kleinste_google_zahl_groesser_als`, die die kleinste Google-Zahl echt größer einer vorgegebenen Zahl bestimmt; eine Funktion `kleinste_google_zahl_zwischen`, die die kleinste Google-Zahl zwischen zwei vorgegebenen Zahlen bestimmt (jeweils einschließlich dieser Zahlen); eine Funktion `strom_der_google_zahlen`, die so wie die Funktion `sieve` den Strom der Primzahlen berechnet (s. Kapitel 1.1.1), den Strom der Google-Zahlen beginnend mit 0 berechnet.

```
type Nat0 = Integer

kleinste_google_zahl_groesser_als :: Integer -> Nat0
kleinste_google_zahl_groesser_als n = ...

kleinste_google_zahl_zwischen :: Nat0 -> Nat0 -> Int
kleinste_google_zahl_zwischen n m = ...

strom_der_google_zahlen :: [Nat0]
strom_der_google_zahlen = ...
```

Gibt es keine Google-Zahl zwischen n und m oder ist $n > m$, liefert `kleinste_google_zahl_zwischen` angewendet auf n und m den Fehler anzeigenden Wert -1. Es gilt: `kleinste_google_zahl_groesser_als (-5) == 0`, `kleinste_google_zahl_groesser_als 0 == 1`; `kleinste_google_zahl_groesser_als 1` und `strom_der_google_zahlen!!2` sind Antwort auf die Bewerbungsfrage.

Nur wer mitmacht, kann gewinnen! Schlagen Sie Ihre Kolleginnen und Kollegen!

Wer kann am 07.12.2018 in *Haskell Live* die schnellsten Lösungen für diese beiden Aufgaben präsentieren?

Statt mit einem Haskell-Programm dürfen Sie auch mit einem Programm in einer anderen Sprache zur Lösung dieser Aufgaben an den Start gehen! Dem Schnellsten winkt ein Schokoladenweihnachtsmann als Preis!

Die Feststellung über die schnellste Lösung obliegt im Zweifel dem strengen, doch gerechten Blick des oder der leitenden Tutoren! Die Tutor(en)entscheidung ist maßgeblich und unanfechtbar, der Rechtsweg ausgeschlossen.

Interesse gefunden, sich auf eine Bewerbung bei Google vorzubereiten? Die vollständige Bewerbungseinladung aus dem Jahr 2004 mit allen Fragen finden Sie der Druckfassung der Oktoberausgabe des Jahres 2004 des Flaggschiff-Magazins der *Association for Computing Machinery* (*ACM*) beigeheftet (*Communications of the ACM 47(10), 2004*).

Haskell Private

Anmeldungen zu *Haskell Private* sind noch möglich. Nutzen Sie die Möglichkeit. Nähere Hinweise und die URL zur Anmeldungsseite finden Sie auf der Homepage der Lehrveranstaltung.