8. Aufgabenblatt zu Funktionale Programmierung vom 07.12.2016.
Fällig: 14.12.2016 (15:00 Uhr)

Themen: *Vermischte Aufgaben zur Anwendung funktionaler Programmierung*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe8.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.

Zur Frist der Zweitabgabe: Siehe allgemeine Hinweise auf Aufgabenblatt 1.

1. Lifestyle-Events erfreuen sich oft großen Zuspruchs von *Schickeria* und *Adabeis*. Einen Adabei auf einem solchen Event erkennt man daran, dass er ausnahmlos jedes anwesende Mitglied der Schickeria (aus Film, Funk, Fernsehen und anderen Quellen) kennt und möglicherweise auch andere Adabeis. Ein Schickeria-Mitglied erkennt man daran, dass es ausnahmslos alle anwesenden Mitglieder der Schickeria kennt, einschließlich sich selbst, aber keinen einzigen Adabei. Ob ein Adabei sich selbst (im Spiegelbild er-) kennt oder nicht, ist für unsere Aufgabe nicht erheblich. Manche zu einem solchen Event Geladene nehmen ihre Einladung nicht an und erscheinen nicht. Das sind die *Nidabeis*.

   Für die Seitenblickereporterin sind Events mit Schickeria-Beteiligung besonders reizvoll für die nächste Ausgabe, also Events mit mindestens einem teilnehmenden Schickeria-Angehörigen. Helfen Sie ihr, die schicken Events zu besuchen, in dem Sie ihr ein Haskell-Programm schreiben, das ihr für gegebene Listen von Eingeladenen, Teilnehmern und wer wen kennt, bestimmt, ob mindestens ein Schickeria-Angehöriger am Event teilnimmt.

   Für die Modellierung dieses Problems in Haskell gehen wir von einer 20-elementigen Menge geladener Gäste aus, deren Mitglieder durch die Werte des Aufzählungstyps `GeladenerGast` gegeben sind:

```
data GeladenerGast = A | B | C | D | E | F | G | H | I | J | K  | L | M
                   | N | O | P | Q | R | S | T deriving (Eq,Ord,Enum,Show)
```

   Weiters führen wir folgende Typsynonyme ein:

```
type Schickeria = [GeladenerGast]                    -- Aufsteigend geordnet
type Adabeis    = [GeladenerGast]                    -- Aufsteigend geordnet
type Nidabeis   = [GeladenerGast]                    -- Aufsteigend geordnet
type NimmtTeil  = GeladenerGast -> Bool                   -- Total definiert
type Kennt      = GeladenerGast -> GeladenerGast -> Bool  -- Total definiert
```

   Ein Wert vom Typ `NimmtTeil` gibt für jeden geladenen Gast an, ob er am Event teilnimmt oder nicht. Ein Wert vom Typ `Kennt` angewendet auf zwei geladene Gäste g_1 und g_2 gibt an, ob g_1 g_2 kennt oder nicht.

   Schreiben Sie Haskell-Rechenvorschriften

```
istSchickeriaEvent :: NimmtTeil -> Kennt -> Bool
istSuperSchick :: NimmtTeil -> Kennt -> Bool
istVollProllig :: NimmtTeil -> Kennt -> Bool
schickeria :: NimmtTeil -> Kennt -> Schickeria
adabeis    :: NimmtTeil -> Kennt -> Adabeis
nidabeis   :: NimmtTeil -> Kennt -> Nidabeis
```

   mit folgenden Funktionalitäten: Angewendet auf zwei Werte f und g der Typen `NimmtTeil` und `Kennt` liefert die Funktion

   (a) `istSchickeriaEvent` den Wert `True`, wenn f und g einen Event beschreiben, zu dem sich mindestens ein Schickeria-Angehöriger hingezogen fühlt und teilnimmt und den Event so zu einem Schickeria-Event adelt, sonst `False`.

(b) `istSuperSchick` den Wert `True`, wenn f und g einen Schickeria-Event beschreiben, auf dem die Schickeria ungestört von Adabeis unter sich ist, sonst `False`.

(c) `istVollProllig` den Wert `True`, wenn f und g einen Event beschreiben, den die Schickeria meidet und die Seitenblickereporterin deshalb nicht zu besuchen braucht, sonst `False`.

(d) `schickeria` die aufsteigend geordnete Liste der Schickeria-Angehörigen, die an dem durch f und g beschriebenen Event teilnehmen.

(e) `adabeis` die aufsteigend geordnete Liste der Adabeis, die an dem durch f und g beschriebenen Event teilnehmen.

(f) `nidabeis` die aufsteigend geordnete Liste der Nidabeis, die an dem durch f und g beschriebenen Event nicht teilnehmen.

Für die Implementierung dürfen Sie davon ausgehen, dass alle Funktionen ausschließlich mit total definierten Argumentfunktionen aufgerufen werden.

Ohne Abgabe: Überlegen Sie sich, dass es auf einem Event höchstens eine nichtleere Menge von Schickera-Angehörigen geben kann.

2. Wir definieren folgenden Strom s natürlicher Zahlen:

   - Die Elemente von s sind echt anwachsend.
   - s beginnt mit der Zahl 1.
   - Wenn s die Zahl n enthält, so enthält s auch die Zahlen $2n$, $3n$ und $5n$.
   - s enthält keine anderen Zahlen.

   Der Strom s beginnt mit diesen Festlegungen mit den Zahlen $1, 2, 3, 4, 5, 6, 8, 9, 10, 12, ...$

   Schreiben Sie eine (0-stellige) Haskell-Rechenvorschrift

   ```
   stream :: [Integer]
   ```

   so dass die Auswertung von `stream` die Zahlen des oben beschriebenen Stroms s liefert.

3. Schreiben Sie eine Haskell-Rechenvorschrift

   ```
   type Quadrupel = (Integer,Integer,Integer,Integer)
   quadrupel :: Integer -> [Quadrupel]
   ```

   die angewendet auf eine natürliche Zahl n, $n \geq 1$, in lexikographisch aufsteigender Ordnung den (Anfangs-) Strom der Quadrupel (a, b, c, d) über den natürlichen Zahlen bestimmt mit $a^3 + b^3 = c^3 + d^3$ sowie $1 \leq a, b, c, d \leq n$, $a \leq b$ und $a < c \leq d$. Ist $n \leq 0$, so liefert die Funktion `quadrupel` die leere Liste.

   Implementieren Sie `quadrupel` mithilfe eines Generators `gen` und eines Filters `fil`. Der Generator `gen` erzeugt den (Anfangs-) Strom der Quadrupel (a, b, c, d) über den natürlichen Zahlen im Bereich von 1 bis n mit $1 \leq a, b, c, d \leq n$, $a \leq b$ und $a < c \leq d$. Der Filter `fil` entfernt aus einem Strom beliebiger Quadrupel (a, b, c, d) über den natürlichen Zahlen all diejenigen Quadrupel, die die Gleichung $a^3 + b^3 = c^3 + d^3$ verletzen. Implementieren Sie zusätzlich einen Selektor `sel`, der aus einem Strom das $|n|$-te Element, $n \in \mathbb{Z}$, $|n|$ Betrag von n, herausgreift. Für $n = 0$ wird also das Kopfelement des Stroms herausgegriffen.

   ```
   gen :: Integer -> [Quadrupel]
   fil :: [Quadrupel] -> [Quadrupel]
   sel :: Int -> [Quadrupel] -> Quadrupel
   ```

   Implementieren Sie `quadrupel`, `gen`, `fil` und `sel` auf globalem Niveau, so dass sie separat getestet werden können.

Hinweis:

- Verwenden Sie *keine* Module. Wenn Sie Funktionen wiederverwenden möchten, kopieren Sie diese in die Abgabedatei `Aufgabe8.hs`. Andere Dateien als diese werden vom Abgabeskript ignoriert.

Haskell Live

An einem der kommenden *Haskell Live*-Termine, der nächste ist am Freitag, den 09.12.2016, werden wir uns, wenn es die Zeit erlaubt, u.a. mit der Aufgabe *Tortenwurf* beschäftigen.

Tortenwurf

Wir betrachten eine Reihe von $n + 2$ nebeneinanderstehenden Leuten, die von paarweise verschiedener Größe sind. Eine größere Person kann stets über eine kleinere Person hinwegblicken. Demnach kann eine Person in der Reihe so weit nach links bzw. nach rechts in der Reihe sehen bis dort jemand größeres steht und den weitergehenden Blick verdeckt.

In dieser Reihe ist etwas Ungeheuerliches geschehen. Die ganz links stehende 1-te Person hat die ganz rechts stehende $n + 2$-te Person mit einer Torte beworfen. Genau p der n Leute in der Mitte der Reihe hatten während des Wurfs freien Blick auf den Tortenwerfer ganz links; genau r der n Leute in der Mitte der Reihe hatten freien Blick auf das Opfer des Tortenwerfers ganz rechts.

Wieviele Permutationen der n in der Mitte der Reihe stehenden Leute gibt es, so dass gerade p von ihnen freie Sicht auf den Werfer und r von ihnen auf das Tortenwurfopfer hatten?

Schreiben Sie in Haskell oder einer anderen Programmiersprache ihrer Wahl eine Funktion, die zu einer vorgegebenen Zahl $n \leq 10$ von Leuten in der Mitte der Reihe, davon p mit $1 \leq p \leq n$ mit freier Sicht auf den Werfer und r mit $1 \leq r \leq n$ mit freier Sicht auf das Opfer, diese Anzahl von Permutationen berechnet.