5. Aufgabenblatt zu Funktionale Programmierung vom Mi, 15.11.2017.
Fällig: Mi, 22.11.2017 (15:00 Uhr)

Themen: *Funktionen über Graphen und Netzwerken, Listenkomprehension*

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 `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.

1. Gegeben sind folgende Typen zur Modellierung eines Melderegisters:

```
type Nat1           = Int
type Wahrheitswert  = Bool
type Name           = String
type Alter          = Nat1
data Geschlecht     = M | W | X deriving Show
type Gemeinde       = String
type Strasse        = String
type Hausnr         = Nat1
data Person         = P Name Alter Geschlecht Wohnsitze deriving Show
data Anschrift      = A Gemeinde Strasse Hausnr deriving Show
type Wohnsitze      = [Anschrift]
type Von_Anschrift  = Anschrift
type Nach_Anschrift = Anschrift
type Melderegister  = [Person]
```

   Implementieren Sie Haskell-Rechenvorschriften

```
einwohner :: Melderegister -> Gemeinde -> [(Name,Geschlecht,Alter)]
durchschnittsalter_mit_Geschlecht_in :: Melderegister -> Geschlecht -> Gemeinde -> Alter
ist_wohnhaft :: Melderegister -> Name -> Gemeinde -> Wahrheitswert
haben_ausschliesslich_als_Wohnsitz :: Melderegister -> Anschrift -> [Person]
ummelden :: Melderegister -> Von_Anschrift -> Nach_Anschrift -> Melderegister
bereinige_Melderegister :: Melderegister -> Melderegister
```

   die folgendes leisten: Die Funktion

   - `einwohner` liefert Name, Geschlecht und Alter aller Personen mit mindestens einem Wohnsitz in der angegebenen Gemeinde. Die Einträge der Ergebnisliste sollen dabei lexikographisch aufsteigend nach Namen geordnet sein, Einträge mit übereinstimmenden Namen aufsteigend nach Geschlecht (mit `M` ‘größer’ `W` ‘größer’ `X`) und Einträge mit übereinstimmenden Namen und Geschlecht aufsteigend nach Alter.
   - `durchschnittsalter_mit_Geschlecht_in` liefert abgerundet auf die nächstkleinere ganze Zahl das Durchschnittsalter aller Personen eines bestimmten Geschlechts in einer Gemeinde. Gibt es keine solchen Personen, liefert sie das fehleranzeigende Resultat 99999.
   - `ist_wohnhaft` überprüft, ob in der angegebenen Gemeinde eine oder mehrere Personen des angefragten Namens einen Wohnsitz haben.
   - `haben_ausschliesslich_als_Wohnsitz` liefert eine Liste aller Personen, die ausschließlich die angegebene Anschrift (möglicherweise mehrfach) als Wohnsitz haben. Das ‘Wohnsitze’-Feld ist dabei in jedem Eintrag der Ergebnisliste durch die leere Liste ersetzt.

- `ummelden` ersetzt bei allen im Melderegister gespeicherten Personen, die die 'Von_Anschrift' (möglicherweise mehrfach) als Wohnsitz haben, die Vorkommen dieser Anschrift durch die 'Nach_Anschrift'. Weitere Änderungen im Melderegister erfolgen nicht.
- `bereinige_Melderegister` löscht alle Duplikate im Melderegister. Ein Duplikat ist dabei ein Eintrag im Melderegister, der in allen Werten mit denen eines anderen Melderegistereintrags vollständig übereinstimmt und an einer größeren Indexposition als dieser steht. Für zwei Wohnsitzlisten heißt vollständig übereinzustimmen, dass die gespeicherten Wohnsitze in Zahl (gemessen in Anzahl der Einträge der Wohnsitzlisten) und Wert übereinstimmen, nicht notwendig jedoch in der Reihenfolge ihrer Anordnung.

Nehmen Sie weitere Typklasseninstanzbildungen vor, falls nötig oder hilfreich, um die Berechnung oder die Ausgabe von Ergebnissen zu erleichtern oder ermöglichen.

2. Sei K eine Menge und $Ka \subseteq K \times K$ eine (möglicherweise leere) Teilmenge des Kreuzprodukts von K.

   Dann heißt das Paar $G = (K, Ka)$

   - *gerichteter Graph* mit Knotenmenge K und gerichteter Kantenmenge Ka. Ein Knoten $k \in K$ ist über eine gerichtete Kante von einem Knoten $k' \in K$ über eine (gerichtete) Kante *erreichbar* gdw. $(k', k) \in Ka$ gilt. Für eine gerichtete Kante $(k', k) \in Ka$ heißt k' *Anfangspunkt* der Kante und k *Endpunkt*. Stimmen Anfangs- und Endpunkt einer Kante überein, so heißt die Kante *Kringel*.
   - *ungerichteter Graph* mit Knotenmenge K und ungerichteter Kantenmenge Ka gdw. für alle $(k, k') \in Ka$ auch $(k', k) \in Ka$ ist. In diesem Fall heißen die Knoten k und k' durch eine Kante in G verbunden.

   Sind in einem ungerichteten Graphen zwei Knoten k und k' durch eine Kante verbunden, so ist k von k' erreichbar und umgekehrt. Konzeptuell verschmelzen wir dabei die gerichteten Kanten von k nach k' und umgekehrt zu einer (ungerichteten) Kante.

   Ist in einem (gerichteten oder ungerichteten) Graphen Knoten k von Knoten k' über eine Kante erreichbar, so heißen die Knoten k und k' *benachbart* und k *Nachbar* von k' und umgekehrt. Ein Knoten ist sich selbst benachbart, wenn er Ausgangs- und Endpunkt einer Kringelkante ist.

   Gerichtete und ungerichtete Graphen über einer endlichen Knotenmenge K können wir in Haskell mithilfe von Abbildungen modellieren:

```
data Knoten = K1 | K2 | K3 | K4 | K5
              | K6 | K7 | K8 | K9 | K10 deriving (Eq,Show)
type Graph = Knoten -> [Knoten]   -- Total definiert
newtype G_Graph = GGr Graph
newtype U_Graph = UGr Graph
```

   Sei `g` ein total definierter Wert vom Typ `Graph`. Wir sagen:

   - `g` repräsentiert den ungerichteten Graphen

     $$UG = (\{\texttt{K1},\ldots,\texttt{K10}\}, \{(k, k') \mid k' \in \texttt{g}(k)\})$$

     gdw. $\forall\, k, k' \in \{\texttt{K1}, \ldots, \texttt{K10}\}.\ k' \in \texttt{g}(k) \Leftrightarrow k \in \texttt{g}(k')$.
   - `g` repräsentiert den gerichteten Graphen

     $$GG = (\{\texttt{K1}, \ldots, \texttt{K10}\}, \{(k, k') \mid k' \in \texttt{g}(k)\})$$

     gdw. `g` repräsentiert keinen ungerichteten Graphen.
   - `g` heißt *minimal* gdw. $\forall\, k \in \{\texttt{K1}, \ldots, \texttt{K10}\}$. $\texttt{g}(k)$ ist frei von Duplikaten.

Implementieren Sie über den obigen und folgenden Datentypen

```
data Klassifikation = GG         -- Fuer 'gerichteter Graph'
                    | UG         -- Fuer 'ungerichteter Graph'
                    | MGG        -- Fuer 'minimaler gerichteter Graph'
                    | MUG        -- Fuer 'minimaler ungerichteter Graph'
                    deriving (Eq,Show)
data Farbe = Tuerkis | Blau deriving (Eq,Show)
type Faerbung = Knoten -> Farbe  -- Total definiert
```

Haskell-Rechenvorschriften für folgende Aufgaben:

```
ist_minimal :: Graph -> Bool
klassifiziere :: Graph -> Klassifikation
erweitere :: Graph -> U_Graph
ist_zweifaerbbar :: U_Graph -> Bool
ist_zweifaerbung :: U_Graph -> Faerbung -> Bool
```

Die Funktion

- `ist_minimal` überprüft, ob das Argument einen minimalen Graphen repräsentiert und gibt entsprechend den Wert `True` oder `False` zurück.
- `klassifiziere` überprüft, ob das Argument einen gerichteten, ungerichteten, minimalen gerichteten oder minimalen ungerichteten Graphen repräesentiert und gibt den entsprechenden Wert vom Typ `Klassifikation` zurück.
- `erweitere` erzeugt aus einem Wert `g` vom Typ `Graph` einen Wert `(UGr g')` vom Typ `U_Graph`, so dass `g'` minimal ist und $\forall\, k, k' \in \{K1, ..., K10\}.\ k' \in \mathtt{g'}(k) \Leftrightarrow (k' \in \mathtt{g}(k) \vee k \in \mathtt{g}(k'))$.
- `ist_zweifaerbbar` überprüft, ob der Argumentgraph zweifärbbar ist und gibt entsprechend den Wert `True` oder `False` zurück. Ein ungerichteter Graph G heißt *zweifärbbar* gdw. jeder Knoten in G ist derart mit einer von zwei Farben färbbar, so dass keine zwei benachbarten Knoten mit derselben Farbe gefärbt sind.
- `ist_zweifaerbung` überprüft, ob der Argumentgraph mit der Argumentfärbung in korrekter Weise zweigefärbt ist und gibt entsprechend `True` oder `False` zurück.

Bei der Implementierung dürfen Sie davon ausgehen, dass alle in Aufrufen verwendeten Graph- und Färbungsargumente total definert sind und die Funktionen `ist_zweifaerbbar` und `ist_zweifaerbung` nur auf solche Graphargumente angewendet werden, die einen minimalen ungerichteten Graphen repräsentieren.

Überlegen Sie sich (ohne Abgabe!), auf welche Weise Sie die korrekte Arbeitsweise der Funktion `erweitere` überprüfen können; eine naive Ausgabe des funktionalen Resultatwerts von `erweitere` ist nicht möglich, zum Testen aber möglicherweise auch nicht nötig.

Nehmen Sie weitere Typklasseninstanzbildungen vor, falls nötig oder hilfreich.

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 17.11.2017, werden wir uns u.a. mit der Aufgabe *World of Perfect Towers* beschäftigen.

World of Perfect Towers

In diesem Spiel konstruieren wir Welten perfekter Türme. Dazu haben wir n Stäbe, die senkrecht auf einer Bodenplatte befestigt sind und auf die mit einer entsprechenden Bohrung versehene Kugeln gesteckt werden können. Diese Kugeln sind ebenso wie die Stäbe beginnend mit 1 fortlaufend nummeriert.

Die auf einen Stab gesteckten Kugeln bilden einen Turm. Dabei liegt die zuerst aufgesteckte Kugel ganz unten im Turm, die zu zweit aufgesteckte Kugel auf der zuerst aufgesteckten, usw., und die zuletzt aufgesteckte Kugel ganz oben im Turm. Ein solcher Turm heißt *perfekt*, wenn die Summe der Nummern zweier unmittelbar übereinanderliegender Kugeln eine Zweierpotenz ist. Eine Menge von n perfekten Türmen heißt *n-perfekte Welt*.

In diesem Spiel geht es nun darum, n-perfekte Welten mit maximaler Kugelzahl zu konstruieren. Dazu werden die Kugeln in aufsteigender Nummerierung, wobei mit der mit 1 nummerierten Kugel begonnen wird, so auf die n Stäbe gesteckt, dass die Kugeln auf jedem Stab einen perfekten Turm bilden und die Summe der Kugeln aller Türme maximal ist.

Schreiben Sie in Haskell oder einer anderen Programmiersprache ihrer Wahl eine Funktion, die zu einer vorgegebenen Zahl n von Stäben die Maximalzahl von Kugeln einer n-perfekten Welt bestimmt und die Türme dieser n-perfekten Welt in Form einer Liste von Listen ausgibt, wobei jede Liste von links nach rechts die Kugeln des zugehörigen Turms in aufsteigender Reihenfolge angibt.

Haskell Private

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