7. Aufgabenblatt zu Funktionale Programmierung vom 23.11.2010.
Fällig: 30.11.2010 / 07.12.2010 (jeweils 15:00 Uhr)

Themen: *Funktionen höherer Ordnung, Polymorphie und Typklassen*

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 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 akzeptierende Automaten wie auf Blatt 6

   ```
   type State           = Integer
   type StartState      = State
   type AcceptingStates = [State]
   type Word a          = [a]
   type Row a           = [[a]]

   data AMgraph a       = AMg [(Row a)] deriving (Eq,Show)

   type Automaton a     = AMgraph a
   ```

   sowie den Typ

   ```
   type Postfix a = Word a
   ```

   Schreiben Sie eine Haskell-Rechenvorschrift `isPostfix` mit der Signatur `isPostfix :: Eq a => (Automaton a) -> StartState -> AcceptingStates -> (Postfix a) -> Bool`. Angewendet auf einen Automaten A, einen Anfangszustand s, eine Menge von Endzuständen E und ein Wort p, ist das Resultat von `isPostfix True`, falls p Postfix eines von A bezüglich s und E akzeptierten Wortes ist, sonst `False`.

2. Wir erweitern die Typdeklarationen aus dem ersten Aufgabenteil wie folgt:

   ```
   type Prefix a         = Word a
   ```

   Schreiben Sie eine Haskell-Rechenvorschrift `givePrefix` mit der Signatur `givePrefix :: Eq a => (Automaton a) -> StartState -> AcceptingStates -> (Postfix a) -> (Maybe (Prefix a))`. Angewendet auf einen Automaten A, einen Anfangszustand s, eine Menge von Endzuständen E und ein Wort p, ist das Resultat von `givePrefix Nothing`, falls p kein Postfix eines von A bzgl. s und E akzeptierten Wortes ist, ansonsten `Just` q, so dass die Konkatenation qp ein von A bzgl. s und E akzeptierten Wortes ist. Beachten Sie, dass q i.a. nicht eindeutig bestimmt ist. Es reicht, wenn Ihre Funktion ein gültiges Prefix q zu einem Postfix p bestimmt.

3. Wir betrachten folgende Variante mit a-Werten benannter AL-Graphen.

   ```
   type Vertex          = Integer
   type Origin          = Vertex
   type Destination     = Vertex

   data ALbgraph a      = ALbg [(Origin,a,[Destination])] deriving (Eq,Show)
   ```

   Wir gehen davon aus, dass alle Knoten explizit genannt sind. D.h., für jeden Knoten, der in einer Zielknotenliste (`Destination`-Liste) auftaucht, gibt es auch einen Eintrag, in dem dieser als Ursprungsknoten (`Origin`) auftritt. Kein Knoten tritt zweimal als Urprungsknoten auf.

   Schreiben Sie eine Haskell-Rechenvorschrift `traverse` mit Signatur `traverse :: Eq a => (a -> a) -> (a -> Bool) -> (ALbgraph a) -> (ALbgraph a)`. Angewendet auf eine Funktion f, ein

Prädikat p und einen ALb-Graphen G besucht die Funktion `traverse` jeden Knoten k in G und transformiert den a-Wert von k mit f, falls der a-Wert das Prädikat p erfüllt, ansonsten lässt sie den a-Wert unverändert.

Angewendet auf die Nachfolgerfunktion (+1) und das Prädikat *isOdd* liefert die Funktion `traverse` also einen ALb-Graphen zurück, in dem alle Schlüsselwerte gerade sind.

Sie können davon ausgehen, dass die Funktion `traverse` nur mit ALb-Graphen aufgerufen wird, die der eingangs genannten Wohlgeformtheitsbedingung genügen.

4. Wir betrachten den Aufzählungstyp

```
data Color        = Red | Blue | Green | Yellow deriving (Eq,Show)
```

und den folgenden Typ ungerichteter Graphen:

```
data Ugraph         = Ug [(Origin,Color,[Destination])] deriving (Eq,Show)
```

Dabei sind `Origin` und `Destination` wie oben definiert. Weiters gehen wir davon aus, dass alle Knoten explizit genannt sind. D.h., für jeden Knoten, der in einer Zielknotenliste (`Destination`-Liste) auftaucht, gibt es auch einen Eintrag, in dem dieser als Ursprungsknoten (`Origin`) auftritt. Kein Knoten tritt zweimal als Urprungsknoten auf. Für keine Kante stimmen Anfangs- und Endknoten überein.

Ein ungerichteter Graph G heißt *wohlgefärbt* genau dann, wenn benachbarte Knoten verschiedene Farben tragen. Zwei Knoten sind *benachbart* genau dann, wenn sie durch eine Kante miteinander verbunden sind.

Schreiben Sie eine Haskell-Rechenvorschrift `isWellColored` mit Signatur `isWellColored :: Ugraph -> Bool`. Angewendet auf einen ungerichteten Graphen G, ist das Resultat von `isWellColored` `True`, falls G wohlgefärbt ist, sonst `False`.

Sie können davon ausgehen, dass die Funktion `isWellColored` nur mit U-Graphen aufgerufen wird, die der oben genannten Wohlgeformtheitsbedingung genügen.

Haskell Live

An einem der kommenden *Haskell Live*-Termine, der nächste ist am Freitag, den 26.11.2010, 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, uws., 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.