8. Aufgabenblatt zu Funktionale Programmierung vom 07.12.2011.
Fällig: 14.12.2011 / 21.12.2011 (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 `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.

Im einzelnen sollen Sie die im folgenden beschriebenen Problemstellungen bearbeiten.

1. Wir betrachten akzeptierende Automaten wie auf Blatt 7

```
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`.

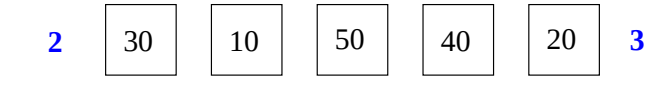
2. Wir erweitern die Typdeklarationen aus dem ersten Aufgabenteil wie folgt:

```
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. *Hochhauszeile* ist eine vereinfachte Variante der Knobelei *Skyline*. In einer Hochhauszeile der Länge n, $n > 1$, steht je ein Hochhaus mit genau 10, 20, 30,...,$n * 10$ Etagen. Die Zahlen am Rand der Hochhauszeile geben an, wieviele Hochhäuser man sieht, wenn man von diesem Punkt in die Hochhauszeile hineinschaut. Dabei gilt: Höhere Hochhäuser verdecken niedrigere.

   Die folgende Abbildung zeigt ein Beispiel für eine Hochhauszeile der Länge 5 (bzw. Länge $(5 + 2)$ einschließlich Sichtbarkeitsinformation).



   Wir modellieren Hochhauszeilen und Sichtbarkeitsinformation durch folgenden Datentyp:

```
type Skyscraperline = [Integer]
```

Zusätzlich führen wir folgende Typsynonyme ein:

```
type Length       = Integer
type VisFromLeft  = Integer
type VisFromRight = Integer
```

In einer Liste der Länge $(n + 2)$ beschreiben der 1-te und $(n + 2)$-te Wert der Liste die Anzahl der sichtbaren Hochhäuser, wenn man von links bzw. rechts aus in die Hochhauszeile hineinschaut. Das 1-te und $(n+2)$-te Feld der Liste bezeichnen wir als *Sichtbarkeitsfelder*, die die Sichtbarkeitsinformation tragen. Als *Hochhauszeile* bezeichnen wir die innere Liste der Länge n, auf denen Hochhäuser stehen ohne die Sichtbarkeitsfelder.

(a) Schreiben Sie eine Haskell-Funktion `isValid` mit der Signatur `isValid :: Skyscraperline -> Bool`. Angewendet auf einen Wert vom Typ `Skyscraperline` überprüft `isValid`, ob das Argument eine *gültige* Hochhauszeile einschließlich Sichtbarkeitsinformation beschreibt, d.h., das Argument eine Liste der Länge $(n + 2)$ ist, auf den inneren n Elementen der Liste genau je ein Hochhaus mit 10, 20, 30,...,$n * 10$ Etagen steht und die Sichtbarkeitsangaben im 1-ten und $n + 2$-ten Listenelement korrekt sind. In diesem Fall ist das Resultat des Aufrufs `True`, sonst `False`.

(b) Schreiben Sie eine Haskell-Funktion `computeVisibility` mit der Signatur `computeVisibility :: Skyscraperline -> Skyscraperline`, die angewendet auf eine Liste der Länge n, in der auf jedem Feld je ein Hochhaus mit genau 10, 20, 30,...,$n * 10$ Etagen steht, diese Hochhauszeile erweitert um die beiden Sichtbarkeitsfelder mit korrekter Sichtbarkeitsinformation zurückliefert. Sie können davon ausgehen, dass die Funktion nur mit solchen Listen aufgerufen wird.

(c) Schreiben Sie eine Haskell-Funktion `buildSkyscrapers` mit der Signatur `buildSkyscrapers :: Length -> VisFromLeft -> VisFromRight -> Maybe Skyscraperline`, die angewendet auf die Länge der Hochhauszeile, die Anzahl der von links und von rechts sichtbaren Hochhäuser eine passende Hochhauszeile mit entsprechenden Sichtbarkeitsfeldern zurückliefert, also einen Wert vom Typ `Just Skyscraperline`, wenn möglich; ansonsten den Wert `Nothing`. Ist die Hochhauszeile durch Länge und Sichtbarkeitsinformation nicht eindeutig festgelegt, reicht es, wenn ihre Funktion eine den Anforderungen genügende Bebauung zurückliefert.

(d) Schreiben Sie eine Haskell-Funktion `noOfSkyscraperLines` mit der Signatur `noOfSkyscraperLines :: Length -> VisFromLeft -> VisFromRight -> Integer`, die die Anzahl zulässiger Hochhausbebauungen für bestimmte Hochhauszeilenlänge und Sichtbarkeitsinformation berechnet.

(e) Schreiben Sie eine Haskell-Funktion `allSkyscraperLines` mit der Signatur `allSkyscraperLines :: Length -> VisFromLeft -> VisFromRight -> [Skyscraperline]`, die alle zulässigen Hochhausbebauungen einschließlich Sichtbarkeitsinformationen für gegebene Hochhauszeilenlänge und Sichtbarkeitsinformation zurückliefert. Dabei seien die verschiedenen Permutationen in der Resultatliste lexikographisch aufsteigend sortiert.

Sie können davon ausgehen, dass die Funktionen aus c), d) und e) nur für Hochhauszeilen bis zur Länge 5 einschließlich getestet werden.

Haskell Live

An einem der kommenden *Haskell Live*-Termine, der nächste ist am Freitag, den 09.12.2011, 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.