8. Aufgabenblatt zu Funktionale Programmierung vom 30.11.2010.
Fällig: 07.12.2010 / 14.12.2010 (jeweils 15:00 Uhr)

Themen: *Funktionen auf Graphen und Feldern, Knobeleien*

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 noch einmal den Typ ungerichteter Graphen von Aufgabenblatt 7, die denselben Anforderungen wie dort genügen. Ebenso übernehmen wir von Aufgabenblatt 7 den Begriff der Wohlgefärbtheit solcher Graphen.

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

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

   Schreiben Sie eine Haskell-Rechenvorschrift `color` mit Signatur `color :: Ugraph -> Maybe Ugraph`. Das Resultat von `color` angewendet auf einen Graphen G ist `Nothing`, falls sich G nicht wohlgefärbt färben lässt; anderenfalls ist das Resultat `Just g`, so dass g wohlgefärbt ist, d.h. die Funktion `isWellColored` von Aufgabenblatt 7 angewendet auf `g` lieferte `True`. Beachten Sie, dass das Resultat nicht eindeutig festgelegt ist, wenn sich der Graph wohlfärben lässt. In diesem Fall reicht es, dass Ihre Funktion einen wohlgefärbten Graphen bestimmt.

2. In vielen Tageszeitungen finden sich als Knobelei Sudokus folgender Art.

	3	7	8		6			5
		5	2	7			3	
				3	5		6	8
		1					9	3
		2		5		4		
5	7					8		
2	1		5	6				
	4			2	1	5		
6			3		7	2	4	

   Wir nennen solch ein Sudoku ein 9×9-*Sudoku*. Ein 9×9-Sudoku setzt sich aus 9 Unterquadraten zusammen. Ein vollständig (mit den Zahlen 1 bis 9 ausgefülltes) 9×9-Sudoku heißt *total korrekt* gdw. gilt:

   Die Zahlen von 1 bis 9 kommen genau einmal vor

   - in jeder Zeile
   - in jeder Spalte
   - in jedem Unterquadrat

   Ein total korrektes 9×9-Sudoku heißt

   - *total korrektes Kreuz-Sudoku*, wenn zusätzlich in jeder der beiden Diagonalen
   - *total korrektes Farb-Sudoku*, wenn zusätzlich an allen einander entsprechenden Positionen der Unterquadrate

   jede der Zahlen von 1 bis 9 genau einmal vorkommt.

Analog nennen wir teilausgefüllte 9×9-Sudokus (wie das obige) *anfangskorrekt*, *anfangskorrektes Kreuz-Sudoku* bzw. *anfangskorrektes Farb-Sudoku*, wenn im teilausgefüllten Sudoku gegen keine der obigen Bedingungen verstoßen ist.

Das obige teilausgefüllte 9×9-Sudoku ist anfangskorrekt, anfangskorrektes Farb-Sudoku, aber nicht anfangskorrektes Kreuz-Sudoku.

In Haskell können wir 9×9-Sudokus durch folgenden Datentyp realisieren:

```
type Row    = [Integer]
type Sudoku = [Row]
```

Zusätzlich betrachten wir folgenden Typ zur Codierung der Sudoku-Variante:

```
data Variant = Basic | Cross | Color deriving (Eq,Show)
```

(a) Schreiben Sie eine Wahrheitswertfunktion `isValid` mit der Signatur `isValid :: Sudoku -> Variant -> Bool`. Die Funktion `isValid` liefert den Wert `True`, wenn das Argument-Sudoku ein anfangs- oder ein total korrektes 9×9-Sudoku, Kreuz- oder Farb-Sudoku entsprechend der vom zweiten Argument vorgegebenen Variante ist, sonst `False`.

(b) Schreiben Sie eine Rechenvorschrift `solve :: Sudoku -> Variant -> Maybe Sudoku`, die angewendet auf ein 9×9-Sudoku, ein total korrektes 9×9-Sudoku, Kreuz- oder Farb-Sudoku entsprechend der vom zweiten Argument vorgegebenen Variante zurückgibt, also einen Wert vom Typ `Just Sudoku`, wenn möglich; ansonsten den Wert `Nothing`. Ist mehr als eine Ergänzung zu einem total korrekten 9×9 (Kreuz-, Farb-) Sudoku möglich, reicht es, wenn ihre Funktion eines dieser total korrekten zurückliefert.

Sie können davon ausgehen, dass beide Funktionen nur mit passend dimensionierten 9×9-Feldern aufgerufen werden. Felder, die im Argument nicht mit Zahlen von 1 bis 9 besetzt sind, gelten als unbesetzt bzw. leer. Anders als die mit 1 bis 9 besetzten Felder dürfen diese im Resultat der Funktion `solve` überschrieben werden.

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 03.12.2010, werden wir uns 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.