3. Aufgabenblatt zu Funktionale Programmierung vom 26.10.2011. Fällig: 02.11.2011 / 09.11.2010 (jeweils 15:00 Uhr)

Themen: *Funktionen auf ganzen Zahlen, Listen und Matrizen*

Für dieses Aufgabenblatt sollen Sie die zur Lösung der unten angegebenen Aufgabenstellungen zu entwickelnden Haskell-Rechenvorschriften in einer Datei namens `Aufgabe3.lhs` im home-Verzeichnis Ihres Gruppenaccounts auf der Maschine g0 ablegen. **Anders** als bei der Lösung zu den ersten beiden Aufgabenblättern sollen Sie dieses Mal also ein "**literate Script**" schreiben. Versehen Sie 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.

Auf diesem Aufgabenblatt beschäftigen wir uns mit Operationen zur Vektor- und Matrizenrechnung. Wir benutzen folgende Bezeichnungen:

- (m,n)-Matrix $\mathcal{M}$ für $m,n \in \mathit{IN}$:

$$\mathcal{M} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{12} & \cdots & a_{mn} \end{pmatrix} = (a_{ik})_{(m,n)}$$

  Für $m \neq n$ heißt $\mathcal{M}$ *rechteckig* vom Typ (m,n), für $m = n$ heißt $\mathcal{M}$ *quadratisch* vom Typ (m,m).

- Zeilen- und Spaltenvektor vom Typ $(1,n)$ und $(m,1)$:

  Eine $(1,n)$-Matrix $\alpha^{(n)}$ heißt *Zeilenvektor* der Länge n; eine $(m,1)$-Matrix $\alpha_{(m)}$ heißt *Spaltenvektor* der Länge m.

$$\alpha^{(n)} = (a_1, a_2, \ldots, a_n), \quad \text{Zeilenvektor der Länge } n$$

$$\alpha_{(m)} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix}, \quad \text{Spaltenvektor der Länge } m$$

- Transponierte (n,m)-Matrix $\mathcal{T}$ einer (m,n)-Matrix $\mathcal{M}$ für $m,n \in \mathit{IN}$:

  Die (m,n)-Matrix $\mathcal{M}$ geht durch Vertauschen ihrer Zeilen und Spalten in ihre transponierte (n,m)-Matrix $\mathcal{T}$ über (und umgekehrt).

$$\text{Für} \quad \mathcal{M} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{12} & \cdots & a_{mn} \end{pmatrix} \quad \text{ergibt sich} \quad \mathcal{T} = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \cdots & \cdots & \cdots & \cdots \\ a_{1n} & a_{2n} & \cdots & a_{nm} \end{pmatrix}$$

- Skalarprodukt

  Das *Skalarprodukt* zweier Zeilen- oder Spaltenvektoren der Länge n ist die Summe der Produkte ihrer sich entsprechenden Komponenten.

$$\alpha^{(n)}\beta^{(n)} = \alpha^{(n)}\beta_{(n)} = \alpha_{(n)}\beta^{(n)} = \alpha_{(n)}\beta_{(n)} = \Sigma_{j=1}^{n} a_j b_j$$

In Haskell können wir Matrizen und (Zeilen-/Spalten-) Vektoren über ganzen Zahlen als Listen von Listen ganzer Zahlen implementieren:

```
type Matrix = [[Integer]]
```

Zeilen- und Spaltenvektoren ergeben sich dann als spezielle Matrizen(werte). So werden die Matrizen und Vektoren

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}, \quad (1,2,3,4), \quad \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

durch folgende Werte des Typs `Matrix` dargestellt:

```
m1 = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
m2 = [[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
zv = [[1,2,3,4]]
sp = [[1],[2],[3],[4]]
```

Schreiben Sie folgende Haskell-Rechenvorschriften:

1. Eine Haskell-Rechenvorschrift `anp1 :: [[Integer]] -> Matrix`. Angewendet auf ein Argument L liefert `anp1` als Resultat einen Wert M vom Typ `Matrix` ab, in dem alle Komponentenlisten von L bis zur Länge l der längsten Komponentenliste am Ende mit Nullen aufgefüllt sind. Die längste(n) Komponentenliste(n) bleibt/en unverändert. Auf diese Weise liefert `anp1` eine Matrix vom Typ (`length`L, l) ab. Ist L die leere Liste, ist das Result von `anp1` der Wert `[[1]]`.

   Folgende Beispiele illustrieren das gewünschte Ein-/Ausgabeverhalten:

   ```
   anp1 [] -> [[1]]
   anp1 [1,2,3],[1,2],[1,2,3,4,5],[1]] -> [1,2,3,0,0],[1,2,0,0,0],[1,2,3,4,5],[1,0,0,0,0]]
   anp1 [[1,2,3],[4,5,6]] -> [[1,2,3],[4,5,6]]
   ```

2. Eine Haskell-Rechenvorschrift `anp2 :: [[Integer]] -> Zeilen -> Spalten -> Fuellwert -> Matrix`. Angewendet auf ein Argument L, eine Zahl von Zeilen z und Spalten s und einen Füllwert w liefert `anp2` eine Matrix M vom Typ (z, s) ab. In der Matrix M sind die ersten z Komponentenlisten von L entweder auf die ersten s Elemente abgeschnitten bzw. auf s Elemente mit Wert w aufgefüllt. Enthält L weniger als z Komponentenlisten, so wird am Ende entsprechend mit aus Wert w bestehenden Zeilen der Länge s aufgefüllt. Enthält L mehr als z Komponentenlisten, werden die überzähligen Zeilen entfernt. Auf diese Weise liefert `anp2` eine Matrix vom Typ (z, s) ab.

   Folgende Beispiele illustrieren das gewünschte Ein-/Ausgabeverhalten:

   ```
   type Zeilen = Integer
   type Spalten = Integer
   type Fuellwert = Integer

   anp2 [] 2 3 0 -> [[0,0,0],[0,0,0]]
   anp2 [1,2,3],[1,2],[1,2,3,4,5],[1]] 3 3 9 -> [1,2,3],[1,2,9],[1,2,3]]
   anp2 [[1,2,3],[4,5,6]] 3 4 (-1) -> [[1,2,3,(-1)],[4,5,6,(-1)],[(-1),(-1),(-1),(-1)]]
   ```

3. Eine Haskell-Rechenvorschrift `transp :: [[Integer]] -> Zeilen -> Spalten -> Fuellwert -> Matrix`. Angewendet auf ein Argument L, eine Zahl von Zeilen z und Spalten s und einen Füllwert w liefert `transp` die transponierte Matrix T der im Sinne von Teilaufgabe 2) angepassten (z, s)-Matrix zu L. T ist also eine Matrix vom Typ (s, z).

   Folgende Beispiele illustrieren das gewünschte Ein-/Ausgabeverhalten:

   ```
   type Zeilen = Integer
   type Spalten = Integer
   type Fuellwert = Integer

   transp [] 2 3 0 -> [[0,0],[0,0],[0,0]]
   transp [[1,2,3],[1,2],[1,2,3,4,5],[1]] 3 3 9 -> [1,1,1],[2,2,2],[3,9,3]]
   transp [[1,2,3],[4,5,6]] 3 4 (-1) -> [[1,4,(-1)],[2,5,(-1)],[3,6,(-1)],[(-1),(-1),(-1)]]
   ```

4. Eine Haskell-Rechenvorschrift `sp :: [[Integer]] -> [[Integer]] -> Laenge -> Fuellwert -> Integer`. Angewendet auf zwei Argumente $L1$ und $L2$, die Vektorlänge l und einen Füllwert w liefert der Aufruf von `sp` den Wert des Skalarprodukts der im Sinne von Teilaufgabe 2) zu einem Zeilenvektor vom Typ $(1, l)$ angepassten $L1$ und des zu einem Spaltenvektor vom Typ $(l, 1)$ angepassten $L2$.

   Folgende Beispiele illustrieren das gewünschte Ein-/Ausgabeverhalten:

```
type Laenge = Integer
type Fuellwert = Integer

sp [[1,2,3]] [[4,5,6]] 3 1 -> 32
sp [[1,2,3]] [[4,5,6]] 4 1 -> 33
sp [[1,2,3],[6,6,7,8],[3,45]] [[4,5,6],[1,2]] 4 4 -> 34
sp [] [[2,3,4]] 2 5 -> 35
```

Werden die obigen Funktionen mit nicht positiven Argumenten für Zeilen oder/und Spalten bzw. Länge aufgerufen, endet die Auswertung der Funktionsaufrufe mit dem Aufruf `error "unzulaessig"`.

Denken Sie bitte daran, dass Sie für die Lösung dieses Aufgabenblatts ein “literate” Haskell-Skript schreiben sollen!

Haskell Live

Am Freitag, den 28.10.2011, werden wir uns in *Haskell Live* u.a. mit den Beispielen von Aufgabenblatt 1 beschäftigen, sowie an diesem oder einem der kommenden *Haskell Live*-Termine mit der Aufgabe *City-Maut*.

City-Maut

Viele Städte überlegen die Einführung einer City-Maut, um die Verkehrsströme kontrollieren und besser steuern zu können. Für die Einführung einer City-Maut sind verschiedene Modelle denkbar. In unserem Modell liegt ein besonderer Schwerpunkt auf innerstädtischen Nadelöhren. Unter einem *Nadelöhr* verstehen wir eine Verkehrsstelle, die auf dem Weg von einem Stadtteil A zu einem Stadtteil B in der Stadt passiert werden muss, für den es also keine Umfahrung gibt. Um den Verkehr an diesen Nadelöhren zu beeinflussen, sollen an genau diesen Stellen Mautstationen eingerichtet und Mobilitätsgebühren eingehoben werden.

In einer Stadt mit den Stadtteilen A, B, C, D, E und F und den sieben in beiden Richtungen befahrbaren Routen B–C, A–B, C–A, D–C, D–E, E–F und F–C führt jede Fahrt von Stadtteil A in Stadtteil E durch Stadtteil C. C ist also ein Nadelöhr und muss demnach mit einer Mautstation versehen werden.

Schreiben Sie ein Programm in Haskell oder in einer anderen Programmiersprache Ihrer Wahl, das für eine gegebene Stadt und darin vorgegebene Routen Anzahl und Namen aller Nadelöhre bestimmt.

Der Einfachheit halber gehen wir davon aus, dass anstelle von Stadtteilnamen von 1 beginnende fortlaufende Bezirksnummern verwendet werden. Der Stadt- und Routenplan wird dabei in Form eines Tupels zur Verfügung gestellt, das die Anzahl der Bezirke angibt und die möglichen jeweils in beiden Richtungen befahrbaren direkten Routen von Bezirk zu Bezirk innerhalb der Stadt. In Haskell könnte dies durch einen Wert des Datentyps `CityMap` realisiert werden:

```
type Bezirk     = Integer
type AnzBezirke = Integer
type Route      = (Bezirk,Bezirk)
newtype CityMap = CM (AnzBezirke,[Route])
```

Gültige Stadt- und Routenpläne müssen offenbar bestimmten Wohlgeformtheitsanforderungen genügen. Überlegen Sie sich, welche das sind und wie sie überprüft werden können, so dass Ihr Nadelöhrsuchprogramm nur auf wohlgeformte Stadt- und Routenpläne angewendet wird.