4. Aufgabenblatt zu Funktionale Programmierung vom 26.10.2010.
Fällig: 02.11.2010 / 09.11.2010 (jeweils 15:00 Uhr)

Themen: *Funktionen auf algebraischen Datentypen*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe4.hs` ablegen. Wie etwa für die Lösung zum ersten Aufgabenblatt sollen Sie dieses Mal 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. Im einzelnen sollen Sie die im folgenden beschriebenen Problemstellungen bearbeiten.

1. Binärbäume lassen sich in Haskell durch folgenden algebraischen Typ repräsentieren:

```
data Tree = Leaf Integer |
            Node Integer Tree Tree deriving (Eq,Show)
```

   (a) Unter einer *Schicht* eines Binärbaums verstehen wir die Menge aller Knoten und Blätter eines Binärbaums, die gleich weit von der Wurzel entfernt sind. Die erste Schicht eines Binärbaums ist also die Wurzel selbst. Die zweite Schicht die Menge aller Knoten und Blätter, die unmittelbare Nachfolger der Wurzel sind. Allgemein: Die i-te Schicht eines Binärbaums, $i > 1$, ist die Menge aller Knoten und Blätter, die Nachfolger eines Knotens der $i-1$-ten Schicht sind. Für $i = 1$ besteht die i-te Schicht aus der Wurzel des Baums.

   Schreiben Sie eine Haskell-Rechenvorschrift `writeLayer` mit der Signatur `writeLayer :: Tree -> Ord -> [Layer]`, die die Markierungen des Argumentbaums schichtenweise als Liste von Schichten ausgibt. Die Knoten- und Blattmarkierungen in jeder Schicht werden von links nach rechts aufgeführt, d.h., der am weitesten linke Knoten der Schicht ist das erste Element der Liste, der am weitesten rechte Knoten das letzte Element der Liste. Das Argument vom Typ `Ord` gibt an, ob die Schichten in aufsteigender Folge von der kleinsten zur größten Schicht (`TopDown`) oder in absteigender Folge von der größten zur kleinsten Schicht (`BottomUp`) ausgegeben werden sollen.

```
type Layer = [Integer]
data Ord   = BottomUp | TopDown
```

   (b) Ein Binärbaum ist ein *Suchbaum*, wenn alle Knoten- und Blattmarkierungen paarweise verschieden sind und die Knoten- und Blattmarkierungen in linken Teilbäumen alle kleiner als die der Wurzel und die in rechten Teilbäumen alle größer als die der Wurzel sind.

   Schreiben Sie eine Haskell-Rechenvorschrift `transform` mit der Signatur `transform :: Tree -> STree`, die einen Argumentbaum in einen Suchbaum verwandelt, wobei im Argumentbaum ggf. mehrfach vorkommende Knoten- bzw. Blattmarkierungen gestrichen werden, ansonsten aber alle im Argumentbaum vorkommenden Knoten- und Blattmarkierungen auch im Resultatbaum vorkommen. Beachten Sie, dass das Resultat nicht eindeutig festgelegt ist. Es reicht, dass das Resultat ein Suchbaum ist, der die obigen Anforderungen erfüllt.

```
data STree = Nil |
             SNode Integer STree STree deriving (Eq,Show)
```

2. Adjazenzlisten sind eine weitverbreitete Form zur Darstellung von Graphen. Im Prinzip ist eine Adjazenzlistendarstellung eines Graphen eine Liste von Knoten, wobei jedem Knoten die Liste der von ihm aus über Kanten erreichbaren Nachbarknoten zugeordnet ist, wobei zugleich dadurch die Richtung der Kanten kodiert ist: Vom Knoten jeweils zu den Knoten in seiner Nachbarknotenliste. Wir erlauben dabei, dass Knoten mehrfach in einer Nachbarknotenliste auftauchen können und interpretieren dies als unterschiedliche Wege zwischen zwei Knoten. Eine Kostenkomponente gibt dabei die (möglicherweise negativen) Kosten an, die entlang dieser Kante entstehen.

   In Haskell können wir zur Darstellung folgenden algebraischen Typ verwenden:

```
type Cost     = Integer
type Vertex   = Integer
newtype Graph = Graph [(Vertex,[(Vertex,Cost)])]
```

Für diese Aufgabe nehmen wir nun an, dass die Knotenmenge eines Graphen implizit definiert ist, d.h. durch ein oder mehrere Vorkommen im Argument des Konstruktors `Graph`, also als Knoten mit einer Adjazenzliste bzw. in der Adjazenzliste eines Knotens. Knoten, die ausschließlich in einer Adjazenzliste auftauchen, besitzen also keine Nachfolger.

Unter einem Pfad in einem Graphen verstehen wir eine nichtleere gerichtete Kantenfolge, die zwei Knoten des Graphens verbindet. Die Knotenfolge des Pfades ist die Folge der auf diesem Pfad liegenden Knoten einschließlich des Start- und Endknotens des Pfads.

(a) Schreiben Sie eine Haskell-Rechenvorschrift `path` mit der Signatur `path :: Graph -> Vertex -> Vertex -> Cost -> Result`, die bestimmt, ob es zwischen zwei Knoten eines Graphen einen Pfad gibt, dessen Kosten kleiner oder gleich der vom dritten Argument vorgegebenen Kosten sind. Entsprechend wird das Resultat `Yes` bzw. `No` ausgegeben. Die Kosten eines Pfades ist dabei die Summe der entsprechenden Kantenkosten. Ist einer der Argumentknoten nicht im Graphen enthalten, wird das Resultat `Invalid` ausgegeben.

```
data Result = Yes | No | Invalid deriving (Eq,Show)
```

(b) In dieser Teilaufgabe nehmen wir an, dass Kanten mit negativen Kosten oder Kosten 0 nicht durchlaufen werden können. Nur Kanten mit echt positiven Kosten können durchlaufen werden. Schreiben Sie unter dieser Voraussetzung eine Haskell-Rechenvorschrift `minpath` mit der Signatur `minpath :: Graph -> Vertex -> Vertex -> ([Vertex],Cost)`, die angewendet auf einen Graphen g, einen Knoten v und einen Knoten w einen Pfad mit minimalen Kosten von v nach w bestimmt, falls es einen solchen gibt. Gibt es keinen solchen Pfad, oder ist einer der Argumentknoten nicht im Graph enthalten, ist das Resultat der Funktion das Tupel `([],0)`. Gibt es mehr als einen Pfad mit minimalen Kosten, so ist es egal, welchen Pfad die Funktion bestimmt. In jedem Fall enthält dann das Resultattupel die Knotenfolge des Pfades und die Kosten dieses Pfades.

Haskell Live

Am Freitag, den 29.10.2010, werden wir uns in *Haskell Live* u.a. mit den Beispielen der Aufgabenblätter 1 und 2 beschäftigen.