7. Aufgabenblatt zu Funktionale Programmierung vom 30.11.2011.
Fällig: 07.12.2011 / 14.12.2011 (jeweils 15:00 Uhr)

Themen: *Typklassen, Polymorphie und Überladung*

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. 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.

Im ersten Teil betrachten wir die folgenden Datentypen

```
type Vertex         = Integer
type Origin         = Vertex
type Destination    = Vertex
type Key            = Integer
type Name           = Integer

data BTree a        = BLeaf Key a |
                      BNode Key a (BTree a) (BTree a) deriving Show

data LTree a        = LNode Key a [(LTree a)] deriving Show

data ALgraph        = ALg [(Origin,[Destination])] deriving (Eq,Show)
```

und die Typklasse `Structure`:

```
class Structure s where
   noOfSources        :: s -> Integer
   noOfSinks          :: s -> Integer
   notSourceConnected :: s -> [Name]
   notSinkConnected   :: s -> [Name]
```

Strukturen sind für uns B-Bäume, L-Bäume und AL-Graphen. Kanten in AL-Graphen sind gerichtet, ebenso Kanten in B-Bäumen und L-Bäumen, hier durch Vater/Sohn-Beziehungen gegeben, gerichtet vom Vater zum Sohn. Eine *Quelle* (*Source*) in einer Struktur ist ein Knoten ohne eingehende Kanten. Eine *Senke* (*Sink*) in einer Struktur ist ein Knoten ohne ausgehende Kanten. In einem AL-Graphen ist der *Name* (*Name*) eines Knotens durch seine Knotennummer gegeben, in einem B- bzw. L-Baum durch seinen eindeutigen *Schlüssel* (*Key*).

Angewendet auf Werte w von Typen t der Typklasse `Structure` berechnet

- `noOfSources` die Anzahl der Quellen in w
- `noOfSinks` die Anzahl der Senken in w
- `notSourceConnected` aufsteigend nach Namen geordnet die Knoten, die nicht von einer Quelle in w aus über einen möglicherweise leeren gerichteten Pfad erreichbar sind (eine Quelle ist also stets von sich aus erreichbar).
- `notSinkConnected` aufsteigend nach Namen geordnet die Knoten, von denen aus keine Senke in w über einen möglicherweise leeren gerichteten Pfad erreichbar ist (von einer Senke ist also stets diese Senke selbst erreichbar).

1. Machen Sie die Typen `BTree a` und `LTree a` mithilfe expliziter Instanzdeklarationen `instance Eq a => Eq (BTree a)...` und `instance Eq a => Eq (LTree a)...` zu Instanzen der Typklasse `Eq`. B-Bäume bzw. L-Bäume sind gleich, wenn sie in Struktur und `a`-Wert übereinstimmen. Der `Key`-Wert soll für den Gleichheitstest keine Rolle spielen.

2. Machen Sie die Typen `BTree a`, `LTree a` und `ALgraph` mithilfe von Instanzdeklarationen `instance Structure...` zu Instanzen der Typklasse `Structure` mit obiger Bedeutung (Hinweis: für die hier zu bildenden Instanzen ist ein Kontext `Eq a` nicht erforderlich).

Sie können davon ausgehen, dass die Funktionen nur mit gültigen Bäumen bzw. Graphen aufgerufen werden; insbesondere, dass in B- und L-Bäumen die Schlüsselwerte eindeutig sind. Für AL-Graphen gilt, dass sie außer den explizit als Anfangs- bzw. in der Adjazenzliste eines Knoten als Endknoten genannten Knoten keine weiteren Knoten besitzen. Die Adjazenzliste eines Knotens kann auch leer sein.

Im zweiten Teil betrachten wir folgende Datentypen:

```
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
```

AM-Graphen stellen Graphen in Form einer Adjazenzmatrix dar. Die Knoten sind dabei von 0 bis Anzahl der Zeilen verringert um 1 durchnummeriert. Zwischen zwei Knoten kann es 0, 1 oder auch mehr gerichtete Kanten pro Richtung geben. Jede dieser Kanten ist mit einem `a`-Wert beschriftet. Enthält das j-te Element der i-ten Zeile z.B. die `a`-Liste `[x,y,z]`, so gibt es vom Knoten i zum Knoten j drei gerichtete Kanten, die mit möglicherweise gleichen `a`-Werten `x`, `y` und `z` beschriftet sind. Ist p ein gerichteter Pfad im Graph, so sind die aneindergefügten Kantenbeschriftungen von p die *Pfadbeschriftung* von p. Ein kantenloser Pfad heißt *leer*. Die Kantenbeschriftung des leeren Pfads ist das leere Wort, d.h. die leere (`a`)-Liste.

Wir können einen AM-Graphen als einen nichtdeterministischen endlichen Automaten ohne ε-Übergänge auffassen, den wir zur Spracherkennung verwenden wollen. Angesetzt auf ein Wort w von `a`-Werten, einen Anfangszustand s (`StartState`) und eine Menge von Endzuständen E (`AcceptingStates`) akzeptiert der Automat Wort w, wenn es im Automaten einen gerichteten Pfad von s zu einem Zustand e in E mit Pfadbeschriftung w gibt. Ein Anfangszustand kann zugleich auch akzeptierender Zustand sein.

3. Schreiben Sie eine Haskell-Rechenvorschrift `accept` mit Signatur

   ```
   accept :: Eq a => (Automaton a) -> StartState -> AcceptingStates -> (Word a) -> Bool
   ```

   die angesetzt auf einen Automaten A, einen Anfangszustand s, eine Menge von Endzuständen E und ein Wort w überprüft, ob w akzeptiert wird (Resultat `True`) oder nicht (Resultat `False`). Ist s kein Knoten von A, oder enthält E keinen Knoten von A, so ist das Resultat ebenfalls `False`.

   Sie können davon ausgehen, dass die Funktion `accept` nur mit gültigen Graphen aufgerufen wird, dass also jede Zeile genauso viele Spalten hat wie es Zeilen gibt.

Haskell Live

An einem der kommenden *Haskell Live*-Termine, der nächste ist am Freitag, den 02.12.2011, 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.