6. Aufgabenblatt zu Funktionale Programmierung vom 13.11.2006. Fällig: 28.11.2006 / 05.12.2006 (jeweils 15:00 Uhr)

Themen: *Funktionen auf polymorphen Typen, insbesondere Bäumen und Graphen*

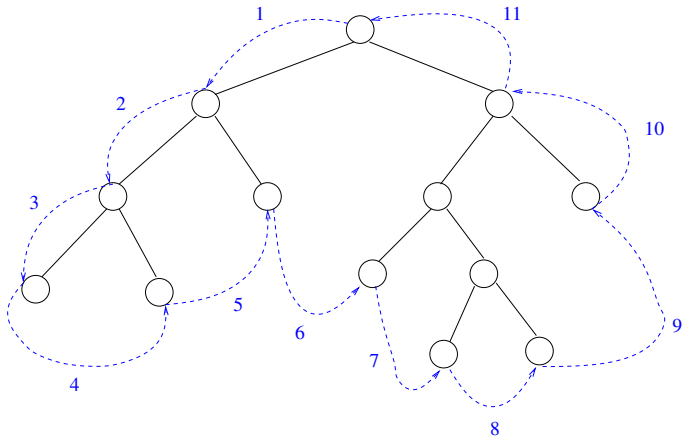
Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe6.lhs` ablegen. Wie für die Lösung zum zweiten Aufgabenblatt sollen Sie dieses Mal also wieder ein "literate" Haskell-Skript 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. Denken Sie bei der Kommentierung daran, dass Ihre Lösungen auch Grundlage für das Abgabegespräch am Semesterende sind.

Im einzelnen sollen Sie die im folgenden beschriebenen Problemstellungen bearbeiten.

1. In dieser Aufgabe betrachten wir noch einmal die Typklasse `GeoFigur` von Aufgabenblatt 5 sowie den folgenden polymorphen algebraischen Datentyp `Tree a b c`:

   ```
   data Tree a b c = Leaf c |
                     Node a b (Tree a b c) (Tree a b c)
   ```

   Erweitern Sie die Typklasse `GeoFigur` um den polymorphen Typ `Tree a b c`. Der Umfang eines nur aus einem Blatt bestehenden Baumes sei dabei mit 1 festgelegt, ansonsten sei der Umfang eines Wertes vom Typ `Tree a b c` durch die Länge des entlang des Randes des Baumes verlaufenden Weges gegeben wie in der folgenden Abbildung illustriert. Der Umfang des dort dargestellten Baums beträgt 11 (Längeneinheiten):

   Beachte: Knoten– und Blattbenennungen sind weggelassen.

   Die Fläche eines Baumes vom Typ `Tree a b c` sei durch die Anzahl der Benennungen im Baum gegeben. Pro Blatt gibt es in Bäumen vom Typ `Tree a b c` eine Benennung, pro innerem Knoten zwei.

2. Neben Adjazenzmatrizen sind Adjazenzlisten 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. In unserer Aufgabe wollen wir jedem Knoten in einer Nachbarknotenliste eine natürliche Zahl als Gewicht zuordnen. Dieses Gewicht kann als Weglänge zwischen den Knoten gedeutet werden. Beachte, dass Knoten wechselweise und auch mehrfach in ihren Nachbarknotenlisten auftauchen können. Dabei kann das Gewicht jeweils verschieden sein, was als Wege unterschiedlicher Länge zwischen diesen Knoten aufgefasst werden kann. Auch kann der Rückweg kürzer oder länger als der Hinweg sein, z.B. aufgrund von Einbahnregelungen. In jedem Fall lassen wir als Gewichte in dieser Aufgabe nur echt positive Zahlen zu.

   In Haskell können wir zur Darstellung folgenden algebraischen Typ verwenden, wobei die Knotenliste des Graphen eine geschachtelte Struktur annimmt:

```
data Graph a = Nil |
               Node a [(a,Int)] (Graph a) deriving (Eq,Show)
```

Ein Wert des Datentyp `Graph a` heißt *wohlgeformt*, wenn jeder Knoten, der in einer Nachbarknotenliste auftritt, genau einmal auch in der Liste der Knoten des Graphen auftritt (also als Argument des Konstruktors `Node`; ggf. mit leerer Nachbarknotenliste) und als Gewichte nur echt positive ganze Zahlen auftreten.

Schreiben Sie eine Haskell-Rechenvorschrift `isWellFormed` mit der Signatur `isWellFormed :: (Graph a) -> Bool`, die als Resultat den Wahrheitswert `True` liefert, wenn der Argumentgraph wohlgeformt ist, `False` sonst.

3. In dieser Aufgabe betrachten wir wieder den Datentyp `Graph a` aus der vorigen Teilaufgabe. Schreiben Sie eine Haskell-Rechenvorschrift `esGibtWeg` mit der Signatur `esGibtWeg :: (Graph a) -> a -> a -> Int -> Bool`. Angewendet auf einen Graphen g, zwei Knoten m und n und eine Weglänge k mit $k > 0$ liefert die Funktion `esGibtWeg` den Wahrheitswert `True`, wenn es einen Weg der Länge höchstens k von m nach n gibt, sonst `False`. Angewendet auf einen Graphen g, zwei Knoten m und n und eine Weglänge k mit $k \leq 0$ liefert die Funktion `esGibtWeg` den Wahrheitswert `True`, wenn es überhaupt einen Weg von m nach n gibt, sonst `False`. Bei der Implementierung können Sie davon ausgehen, dass die Funktion nur mit wohlgeformten Graphen getestet wird.

4. Wir definieren auf der Menge der ganzen Zahlen $\mathbb{Z}$ eine eingeschränkte Potenzfunktion $\odot : \mathbb{Z} \times \mathbb{Z} \to I\!N_0$ wie folgt:

$$\forall\, m, n \in \mathbb{Z}.\ m \odot n =_{df} \begin{cases} m^n & \text{falls } m, n \geq 0 \\ 0 & \text{sonst} \end{cases}$$

Schreiben Sie eine Haskell-Rechenvorschrift `powMin3` mit der Signatur `powMin3 :: String -> String -> String`, die die Potenzoperation $\odot$ für (-3)-adische Zahlen realisiert. Die Rechnung soll dabei wie für die Funktion `addMin3` von Aufgabenblatt 5 ausschließlich im (-3)-adischen System ausgeführt werden, nicht also etwa durch Umwandlung der Argumente in ein anderes b-adisches Zahlsystem, Ausführung der Berechnung in diesem System und Rückumwandlung des Resultats in die entsprechende (-3)-adische Repräsentation.

Denken Sie bitte daran, dass für die Lösung von Aufgabenblatt 6 ein “literate” Haskell-Skript gefordert ist!