Berthold Hoffmann <hof>
Christian Maeder <maeder>
Klaus Lüttich <luettich>
Pascal Schmidt <pharao90>
Felix Beckwermert <foetus>
Dennis Walter <dw>
Franck Ngueuleu <nngf>

Bonusblatt

Ausgabe: 15.12.2003

Bearbeitungszeit: bis 3./4.2.2004

Eine Typklasse für Graphen *20 Punkte*

Einfache Graphen bestehen aus einer Menge von Knoten und einer Menge von gerichteten Kanten, die durch Paare von Knoten repräsentiert werden. Als Typ für Knoten legen wir hier `Integer` fest.

Statt einen abstrakten Datentypen für Graphen zu definieren, benutzen wir nun eine Klasse `Graph` mit Methoden, die ein abstrakter Datentyp exportieren würde. Einen benutzbaren konkreten Datentypen erhalten wir später, indem wir eine Instanz der Klasse `Graph` definieren, durch die die Typvariable `g` (aus der Klasse) instanziiert wird.

```
type Node = Integer
type Edge = (Node, Node)

class Graph g where
   empty :: g
   addNode, delNode :: Node -> g -> g
   isolated :: Node -> g -> Bool
   addEdge, delEdge :: Edge -> g -> g
   nodes :: g -> [Node]
   successors, predecessors :: Node -> g -> [Node]
```

Im Wesentlichen erlaubt es die Klasse `Graph`, auf die direkten Nachfolger und Vorgänger eines Knotens zuzugreifen. Alle Knotenlisten sollten immer streng aufsteigend sortiert sein. Die Knoten eines Graphs sind über die Funktion `nodes` erreichbar und nur solche Knoten können das Ergebnis von `successors` bzw. `predecessors` sein. Ein Knoten $k2$ ist ein Nachfolger von $k1$ genau dann wenn $k1$ ein Vorgänger von $k2$ ist. Dieses ist wiederum äquivalent dazu, dass $(k1, k2)$ eine Kante im Graphen ist. Ein Knoten kann durch `delNode` nur gelöscht werden, wenn er `isolated` ist, d.h. wenn er keine Vorgänger- und Nachfolgerknoten hat. Die Funktionen `addEdge` und `delEdge` verändern die Knotenmenge nicht. Mit `addEdge` kann nur eine Kante hinzugefügt werden, wenn beide Knoten der Kante im Graphen vorkommen.

Mit Hilfe der obigen Klassenmethoden implementieren Sie die folgenden generischen Funktionen:

- Test auf leeren Graphen: `isEmpty :: Graph g => g -> Bool`
- Die Anzahl der ein- und abgehenden Kanten eines Knotens:
  `inDregee, outDegree :: Graph g => Node -> g -> Int`
- Alle Kanten eines Graphs: `edges :: Graph g => g -> [Edge]`
- Alle Kanten von oder zu einem Knoten löschen, so dass er danach isoliert ist: `isolate :: Graph g => Node -> g -> g`
- Der Umkehrgraph: `revGraph :: Graph g => g -> g`
  Im Umkehrgraphen ist die Richtung aller Kanten umgekehrt.
- Alle transitiven Nachfolger eines Knotens:
  `transitiveSuccessors :: Graph g => Node -> g -> [Node]`
  Ein transitiver Nachfolger ist entweder ein direkter Nachfolger oder ein transitiver Nachfolger eines direkten Nachfolgers. Dabei muss man selbstverständlich berücksichtigen, dass Graphen zyklisch sein dürfen!

Wir modellieren nun einen Graphen als eine Abbildung von Knoten auf Nachfolgerknoten mit Hilfe des abstrakten Datentyps `Map` vom vorherigen Übungsblatt. (Die Vorgängerknoten sind dadurch nur implizit durch die Umkehrabbildung gegeben und den größten Aufwand wird die Funktion `predecessors` haben.) Implementieren Sie also für den folgenden Datentyp `GraphByMap` eine Instanz der `Graph`-Klasse.

```
data GraphByMap = GraphByMap (Map Node [Node])

instance Graph GraphByMap where ...
```

Innerhalb der Instanzdeklaration dürfen leider nicht die (instanziierten) Signaturen der Klassenmethoden angegeben werden.

Für `GraphByMap` importieren Sie Ihr Modul `MapBySortedList` vom vorherigen Übungsblatt 4 (Aufgabe 9) oder `Data.FiniteMap` aus der Haskell Bibliothek. Dort heißt der Datentyp allerdings `FiniteMap`:

`haskell.org/ghc/docs/latest/html/libraries/base/Data.FiniteMap.html`

Dieses ist Version 2.0 vom 17. Dezember 2003.