3. Übungsaufgabe zu
Fortgeschrittene funktionale Programmierung
Thema: Rücksetzsuche, funktionale Felder, funktionale Perlen
Ausgegeben: Mi, 11.04.2018, abzugeben: Mi, 02.05.2018 (15:00 Uhr)

Nachträge vom 26.04.2018:

- Ergebnistyp von `search_dfso` von `node` zu `[node]`.
- Deklaration von `Schachbrett` von `data` zu `type`.
- Tippfehler bei `raetsel1` bis `raetsel4` berichtigt.

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften zur Lösung der im folgenden angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `AufgabeFFP3.hs` in Ihrem Gruppenverzeichnis ablegen, wie gewohnt auf oberstem Niveau. Kommentieren Sie Ihre Programme aussagekräftig und benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

- Modifizieren Sie das Funktional `search_dfs` der Rücksetzsuche aus Kapitel 3.2 der Vorlesung zu einem Funktional `search_dfso` ('depth-first-search_first-solution-only')

  ```
  search_dfso :: (Eq node) =>
                 (node -> [node]) -> (node -> Bool) -> node -> [node]
  ```

  das i.w. mit `search_dfs` übereinstimmt, aber anders als `search_dfs` nicht alle Lösungen im Suchraum bestimmt, sondern nach dem Finden der ersten Lösung die Suche abbricht und die gefundene Lösung als Ergebnis liefert.

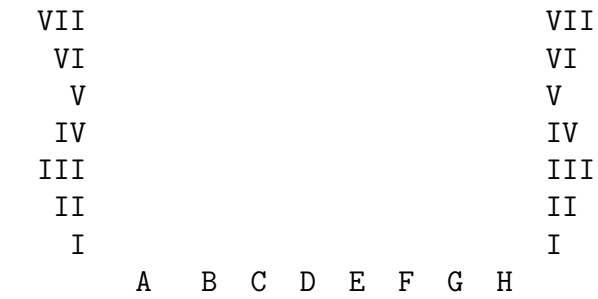
  Das Funktional `search_dfso` soll sich für die Implementierung des Datentyps `Stack` und der stapelmanipulierenden Funktionen auf Implementierung B aus Kapitel 3.2 abstützen, die ebenfalls in der Datei `AufgabeFFP3.hs` erfolgen soll, um ein abgeschlossenes (sich nicht auf selbstdefinierte Module abstützendes) Programm zu erhalten:

  ```
  newtype Stack a = Stk Stack a
  ```

- Betrachte folgende Schachaufgabe: Kann ein Turm unter Einhaltung seiner Zugregeln in genau n Zügen, $0 \leq n \leq 5$, von einem vorgegebenen Start- zu einem vorgegebenen Zielfeld gelangen? Ein Zug darf weder über ein besetztes Feld hinweggehen noch auf einem besetzten Feld beginnen oder enden.

  Lösen Sie diese Aufgabe mithilfe der modifizierten Rücksetzsuche aus dem vorigen Aufgabenteil, wobei ein Schachbrett wie folgt gedacht ist:

  ```
          A   B  C  D  E  F  G  H
  VIII                          VIII
  ```



Zur Modellierung von Turmaufgaben verwenden wir folgende Typen:

```
import Array

data Zahl  = Null | Eins | Zwei | Drei | Vier | Fuenf
                                          deriving (Eq,Ord,Enum,Show)
data Zeile = I | II | III | IV | V | VI | VII | VIII
                                          deriving (Eq,Ord,Enum,Show)
data Reihe = A | B | C | D | E | F | G | H
                                          deriving (Eq,Ord,Enum,Show)
type Besetzt = Bool

instance Ix Zeile where...
instance Ix Reihe where...

type Schachbrett = Array (Reihe,Zeile) Bool

type Zugzahl   = Zahl
type Feld      = (Reihe,Zeile)
type Von       = Feld
type Nach      = Feld
type Startfeld = Von
type Zielfeld  = Nach
type Aufgabe   = (Startfeld,Zielfeld,Zugzahl)
type Zug       = (Von,Nach)
type Zugfolge  = [Zug]
```

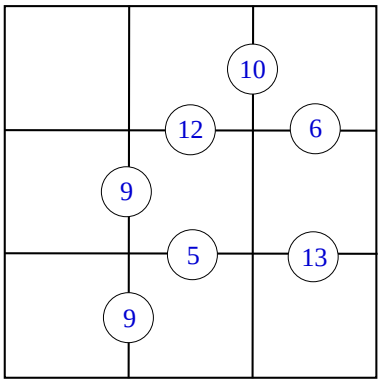
Gesucht sind nun Implementierungen für:

```
data Knoten = ...            -- Informationsreich genug, die Auf-
                             -- gabe zu loesen.

nachf :: Knoten -> [Knoten]  -- Berechnung der lokalen Suchraumum-
nachf k = ...                -- gebung, d.h. der Nachfolgerknoten.

lsg :: Knoten -> Bool        -- Loesung gefunden? Kann Suche ab-
lsg  = ...                   -- gebrochen werden?
```

```
suche :: Schachbrett -> Aufgabe -> Zugfolge
suche sb ag = ...search_dfso...
```

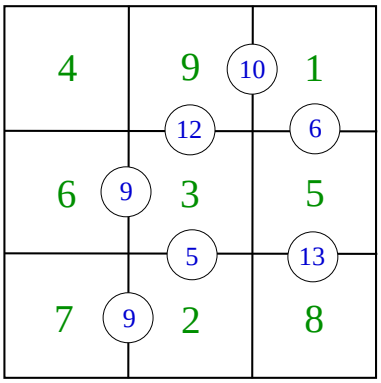
Die Funktion `suche` angewendet auf ein Schachbrett und eine Turmaufgabe liefert die erstgefundene Zugfolge, die den Turm in genau der vorgegebenen Zahl von Zügen vom Start- zum Zielfeld bringt, wenn es (mindestens) eine solche gibt. Ansonsten liefert `suche` die leere Zugfolge als Ergebnis, auch dann, wenn Start- oder/und Zielfeld der Turmaufgabe auf dem Schachbrett besetzt sind. Start- und Zielfeld dürfen in einer Zugfolge mehrfach berührt werden, d.h. als Ausgangs- oder Endpunkt eines Zugs auftreten.

- Betrachte Zahlenrätsel der Form:

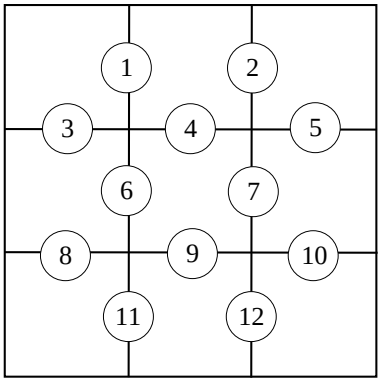


Die Spielregeln sind wie folgt: Trage die Zahlen von 1 bis 9 je einmal in die Felder des Spielfelds ein. Die Summe der Einträge benachbarter Felder, die durch einen wertbenannten Kreis zu einem Paar zusammengefasst sind, muss dabei gleich dem im Kreis vorgegebenen Wert sein.

Das obige Zahlenrätsel hat genau eine Lösung:



Zur Modellierung von Rätseln dieser Art nummerieren wir die möglichen Feldpaare fortlaufend von links oben nach rechts unten durch:



Rätsel können damit als Vektoren, Lösungen als Matrizen modelliert werden:

```
type Raetsel = Array Int Int deriving (Eq,Show)
type Loesung = Array (Int,Int) Int deriving (Eq,Show)

loese :: Raetsel -> Loesung
```

Versuchen Sie, eine performante Implementierung fuer `loese` durch schrittweise Verbesserung einer einfachen, offensichtlich richtigen Lösung nach dem Vorbild der Entwicklung funktionaler Perlen zu gewinnen (s. Kapitel 4 der Vorlesung).

Überlegen Sie sich bei jedem Verbesserungsschritt, warum Ihre Lösung korrekt bleibt.

Gibt es mindestens eine gültige Lösung für ein Rätsel, ist es egal, welche dieser Lösungen `loese` liefert; gibt es keine gültige Lösung, liefert `loese` ein Feld, dessen Einträge alle auf 0 gesetzt sind. In jedem Fall ist das Ergebnis von `loese` ein Feld der Dimension 3×3, dessen Zeilen- und Spaltenindizes von 1 bis 3 laufen.

Für den Vektor, der ein Rätsel beschreibt, treffen wir folgende Vereinbarung, um jeden Vektorwert als Rätselbeschreibung auffassen zu können. Wir nutzen dabei aus, dass als Summenvorgaben nur Werte zwischen 3 und 17 in Frage kommen:

- Die Einträge des Vektors werden unabhängig vom Anfangsindex aufsteigend gemäß der in der Abbildung illustrierten Nummerierung als Summenvorgaben für die entsprechenden Feldpaare interpretiert.
- Vektoreinträge mit Werten w zwischen 3 und 17 ($3 \leq w \leq 17$), werden als feste Summenvorgabe interpretiert; Vektoreinträge mit Werten außerhalb dieses Bereichs ($w \leq 2 \vee w \geq 18$) werden als freie Vorgabe interpretiert, d.h. jeder Summenwert ist ür solche Feldpaare erlaubt.
- Enthält der Vektor mehr als 12 Einträge, werden die überzähligen Einträge ignoriert.
- Enthält der Vektor weniger als 12 Einträge, gilt für Feldpaare, für die der Vektor keine Vorgabe trifft, die freie Vorgabe.

Anwendungsbeispiel (`raetsel1` und `loesung1` entsprechen obigem Beispiel):

```
raetsel1 = listArray (1,12) [0,10,0,12,6,9,0,0,5,13,9,0]
raetsel2 = listArray (1,12) [0,10,0,12,3,9,0,0,5,13,9,0]
raetsel3 = listArray (5,15) [0,10,99,12,6,9,(-5),42,5,13,9]
raetsel4 = listArray (10,24) [2,10,18,12,6,9,(-5),42,5,13,9,0,5,2,73]
loesung1 = listArray ((1,1),(3,3)) [4,9,1,6,3,5,7,2,8]
loesung2 = listArray ((1,1),(3,3)) [0,0,0,0,0,0,0,0,0]
loese raetsel1 == loesung1
loese raetsel2 == loesung2
loese raetsel3 == loesung1
loese raetsel4 == loesung1
```