Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
L. Noschinski, L. Hupel, Dr. J. Blanchette, M. Eberl

WS 2014/15
28.11.2014
Abgabe: 05.12.2014

Einführung in die Informatik 2
8. Übung

Aufgabe G8.1 Den Wald vor lauter Bäumen nicht finden

Wir wollen Teilbäume von Bäumen selektieren. Gesucht ist daher eine Funktion, die einen Baum nimmt sowie Instruktionen zum Finden eines Knotens und jenen Knoten zurückgibt.

1. Mögliche Navigationsinstruktionen in einem binären Baum sind "go left" und "go right", die dafür sorgen, dass der linke bzw. rechte Teilbaum selektiert wird. Definieren Sie einen Datentyp `Direction`, welcher die Konstruktoren `L` und `R` hat.
2. Implementieren Sie die Funktion `navigate :: [Direction] -> Tree a -> Maybe (Tree a)`, die einen Satz von Instruktionen befolgt und ggf. den entsprechenden Teilbaum zurückgibt. Für den Fall, dass dieser Teilbaum nicht existiert, so soll `Nothing` zurückgegeben werden.

   Beispiele:

   ```
   navigate [] Empty
     == Just Empty

   navigate [] (Node "hello" Empty Empty)
     == Just (Node "hello" Empty Empty)

   navigate [R] (Node "hi" Empty (Node "hello" Empty Empty))
     == Just (Node "hello" Empty Empty)

   navigate [R, L] (Node "hi" Empty (Node "hello" Empty Empty))
     == Just Empty

   navigate [L, R] (Node "hi" Empty (Node "hello" Empty Empty))
     == Nothing
   ```

   Hinweis: Sie können bei einer Funktionsdefinition mehrere Parameter gleichzeitig matchen. Nutzen Sie dies.
3. *(optional)* Wie könnte eine Datenstruktur aussehen, die eine „aktuelle Position" in einem Baum speichert und neben den Operationen `goLeft` und `goRight` zusätzlich auch `goParent` unterstützt?

Aufgabe G8.2 Sortieren mit Bäumen

In der Vorlesung wurde ein Datentyp `Tree` eingeführt. Die dazu gehörige Funktion `insert :: Ord a => a -> Tree a -> Tree a` fügt Elemente geordnet in einen Baum ein, sofern sie noch nicht im Baum vorhanden sind.

Eine *Invariante* ist eine Eigenschaft, die immer, insbesondere also vor und nach einer Operation auf einer Datenstruktur, gilt.

In diesem Beispiel ist das die *Suchbaumeigenschaft*, die wie folgt definiert ist: Wenn t ein Knoten mit dem Wert a, dem linken Teilbaum l und dem rechten Teilbaum r ist, dann sind alle in l enthaltenen Werte kleiner als a und alle in r enthaltenen Werte größer als a.

1. Schreiben Sie eine Funktion `isOrderedTree :: Ord a => Tree a -> Bool`, die prüft, ob der gegebene Baum die Suchbaumeigenschaft erfüllt.

   Warum hält sich `insert` an die Suchbaumeigenschaft? Welche Funktionen benötigen die Suchbaumeigenschaft?
2. Benutzen Sie `insert`, um eine Funktion `treeSort :: Ord a => [a] -> [a]` zu implementieren, die Listen sortiert und dabei Duplikate entfernt. Bauen Sie dazu zunächst einen geordneten Baum auf und wandeln Sie diesen dann in eine sortierte Liste um.
3. Welche Tests sind sinnvoll, um sicherzustellen, dass `treeSort` wirklich sortiert?

Aufgabe G8.3 Nichtleere Listen

In vielen Situationen benötigt man Listen, von denen man annehmen kann, dass sie mindestens ein Element enthalten. Man kann dafür einfach „gewöhnliche“ Listen nehmen und stets diese Eigenschaft abfragen, z.B. so:

```
if length xs > 0 then
  ... head xs ...
else
  error "something went utterly wrong"
```

Dabei können sich aber sehr leicht Programmierfehler einschleichen, so dass das Programm zur Laufzeit abstürzt. Der übliche Ansatz in Haskell ist daher, einen eigenen Datentyp so zu konstruieren, bei dem ungültige Werte *nicht existieren*.

1. Definieren Sie einen Datentyp `NonEmptyList`, welcher eine Liste darstellen soll, die mindestens ein Element enthält.
2. Schreiben Sie Konvertierungsfunktionen zwischen `[a]` und `NonEmptyList a`.

   ```
   fromList :: [a] -> Maybe (NonEmptyList a)
   toList :: NonEmptyList a -> [a]
   ```

3. Implementieren Sie die Funktionen `nHead`, `nTail` und `nAppend` analog zu `head`, `tail` und `(++)`. Welche Besonderheit gibt es beim Rückgabetyp von `nTail` zu beachten?

Wichtiger Hinweis: Alle Ihre Definitionen müssen folgende Kriterien erfüllen:

- Es dürfen keine vordefinierten Funktionen aus der Standardbibliothek verwendet werden.
- Sie dürfen für keine Eingabe mit einem Fehler abstürzen bzw. in eine Endlosschleife geraten.

Aufgabe H8.1 Bäume anzeigen

Definieren Sie eine Funktion `prettyTree :: Show a => Tree a -> String`, die einen Baum in eine „schöne“ Form bringt. Die Ausgabe soll zeilenweise erfolgen, d.h. jede Ebene des Baums hat eine eigene Zeile. Die einzelnen Knoten einer Ebene sollen mit dem Separator `" | "` getrennt werden. Fehlende Teilbäume (`Empty`) sollen einfach übersprungen werden. Die Ausgabe darf keine Leerzeilen enthalten.

Beispiele:

```
prettyTree Empty == ""
prettyTree (Node 1 Empty Empty) == "1"

prettyTree
  (Node 1 (Node 2 Empty Empty)
          (Node 3 (Node 4 Empty Empty) Empty)) ==
  "1\n2 | 3\n4"
```

Um sich die Ausgabe inkl. Zeilenumbrüche anzeigen zu lassen, verwenden Sie in GHCi die Funktion `putStrLn`:

```
Exercise_8> putStrLn (prettyTree ...)
1
2 | 3
4
```

Hinweis: Um Werte vom Typ `a` in einen `String` umzuwandeln, nutzen Sie die Funktion `show`. Konstruieren Sie zunächst jede Zeile einzeln und nutzen Sie dann die Funktion `intercalate`.

Aufgabe H8.2 Bäume traversieren

Bäume können auf verschiedene Arten in Listen konvertiert werden. In der Gruppenaufgabe ist bereits eine solche Funktion vorgekommen; in dieser Hausaufgabe soll dieses Problem allgemeiner gelöst werden.

Als gemeinsames Beispiel dient der Baum

```
t = Node "+"
       (Node "3" Empty Empty)
       (Node "*" (Node "4" Empty Empty) (Node "2" Empty Empty))
```

Für diese Aufgabe sollen folgende drei Arten betrachtet werden:

Name	**Beschreibung**	**Resultat**
`Postorder`	zuerst die Kinder, dann der Knoten	`["3", "4", "2", "*", "+"]`
`Inorder`	linkes Kind, Knoten, rechtes Kind	`["3", "+", "4", "*", "2"]`
`Preorder`	zuerst der Knoten, dann die Kinder	`["+", "3", "*", "4", "2"]`

Gegeben sei ein Datentyp `Traversal`, der diese drei Arten beschreibt (siehe Schablone). Definieren Sie eine Funktion

```
treeToList :: Traversal -> Tree a -> [a]
```

die einen Baum nach dem angegebenen Schema in eine Liste konvertiert.

Aufgabe H8.3 Bäume modifizieren

Wir wollen die Baumstruktur aus der Vorlesung verwenden um eine Map mit Schlüsseln vom Typ `a` und Werten vom Typ `b` zu implementieren. Dafür brauchen wir Funktionen zum Einfügen, Ändern und Löschen von Einträgen.

1. Diese Funktionen haben viel gemeinsam, darum sollen Sie die Funktion

   ```
   modify :: Ord a => (Maybe b -> Maybe b) -> a
          -> Tree (a,b) -> Tree (a,b)
   ```

   schreiben, mit der sich die gewünschten Funktionen sehr einfach implementieren lassen.

   `modify` f k t nimmt einen Suchbaum t (d.h., die Knoten sind nach dem Schlüssel geordnet) und gibt einen neuen Suchbaum zurück, bei dem der Knoten mit Schlüssel k durch die Funktion f modifiziert wurde.

   - f wird mit dem Wert des Knotens k aufgerufen (oder mit `Nothing`, wenn dieser Knoten nicht existiert).
   - Hat f den Rückgabewert `Nothing`, so wird der Knoten gelöscht
   - Hat f den Rückgabewert `Just x`, so wird der Wert des Knotens auf x geändert (bzw. eingefügt, wenn er vorher nicht existierte).

2. Implementieren Sie die folgenden Funktionen, indem Sie nur `modify` verwenden, aber keine anderen Baumfunktionen (oder -konstruktoren)!

   ```
   insert :: Ord a => b         -> a -> Tree (a,b) -> Tree (a,b)
   update :: Ord a => (b -> b) -> a -> Tree (a,b) -> Tree (a,b)
   delete :: Ord a =>              a -> Tree (a,b) -> Tree (a,b)
   ```

   Die Funktionen sollen sich wie folgt verhalten:

   `insert` v k: Setzt den Wert des Knotens k auf v (und fügt den Knoten ggfs. ein).

   `update` f k: Wendet f auf den Wert des Knotens k an (falls dieser existiert).

   `delete` k: Löscht den Knoten k.

   Technischer Hinweis: Wir werden diese Funktionen mit *unserer* Version von `modify` testen. Falls also das Testsystem andere Ergebnisse Ihrer Funktion ausgibt als bei Ihnen lokal, prüfen Sie bitte, ob Ihre `modify`-Funktion korrekt implementiert ist.

Aufgabe H8.4 Game of Stones (Wettbewerbsaufgabe)

Wenn Sie nicht am Wettbewerb teilnehmen wollen, bearbeiten Sie bitte diese Aufgabe wie gewohnt und geben Sie sie bis zum 05.12.2014 ab. Falls Sie teilnehmen wollen, lesen Sie bitte weiter, andernfalls können Sie den nächsten Absatz ignorieren.

Diese Wettbewerbsaufgabe ist zweiwöchig. Sie haben bis zum 12.12.2014 Zeit, einen cleveren Algorithmus zu implementieren. Der MC wird nach einer Woche Bearbeitungszeit bereits eine Zwischenauswertung vornehmen und Ihnen die Ergebnisse mitteilen. Hierfür wird er alle Abgaben berücksichtigen, die bis zur regulären Abgabefrist (05.12.2014) eingegangen sind. Um Hausaufgabenpunkte auf diese Aufgabe zu erhalten, müssen Sie ebenfalls bereits bis zum 05.12.2014 eine funktionstüchtige Implementation abgeben.

Nachdem der MC Sr. die Wettbewerbsteilnehmer letztes Jahr Othello hat spielen lassen, geht es dieses Jahr um ein ähnlich einfaches Spiel:

Das Spielfeld ist dabei eine Raute, die in 11×11 sechseckige Felder unterteilt ist. Die Spieler (rot und blau) setzen abwechselnd Steine ihrer Farbe auf ein leeres Feld mit dem Ziel, zwei gegenüberliegende Seiten des Spielfelds durch einen Pfad ihrer Farbe zu verbinden. Der rote Spieler muss die linke Seite mit der rechten verbinden, der blaue die untere mit der oberen.

Gegeben sind folgende Definitionen:

```
type Position :: (Int, Int)
data Player = Red | Blue
data Board = ...
playerAt :: Position -> Board -> Maybe Player
```

Ihre Aufgabe ist es nun, eine KI für dieses Spiel zu schreiben:

```
nextMove :: Player -> Board -> Position
```

Der erste Parameter sagt Ihnen hierbei, welcher Spieler Sie sind. Der zweite Parameter ist das aktuelle Spielfeld, das sie mit der Funktion `playerAt` inspizieren können. Die Rückgabe ist die Position, an die Sie einen Stein Ihrer Farbe setzen wollen, wobei natürlich an dieser Position noch kein anderer Stein liegen darf. Sie können hierbei davon ausgehen, dass `nextMove` nur aufgerufen wird, wenn noch mindestens eine freie Position existiert. Ein Beispiel für ein Spiel, in dem der rote Spieler gewonnen hat, sowie eine Skizze der korrekten Nummerierung sehen Sie in Abb. 1.

Sie können hierbei auf ein Repertoire aus vordefinierten Hilfsfunktionen aus dem Module `Game` zurückgreifen. Dokumentation hierzu finden Sie auf `http://www21.in.tum.de/teaching/info2/WS1415/game/`; insbesondere können Sie in GHCi gegen Ihre eigene Strategie spielen (hierfür müssen Sie Ihre Abgabe und die Datei `GameUtils.hs` zusammen in GHCi laden):

```
ghci Exercise_8.hs GameUtils.hs
*Exercise_8> import GameUtils
*Exercise_8 GameUtils> playGameInteractive
                            (simpleStrategy nextMove) Red
```

Sie sehen dann eine Repräsentation des Spielfelds. Ausgefüllte Kreise stehen für rote Steine, gestrichelte Steine für blaue Steine, leere Kreise für ein freies Feld. Sie werden aufgefordert, einen Zug einzugeben (also wohin Sie Ihren Stein legen wollen); hier geben Sie einfach eine Position im Format `x y` ein, mit $1 \leq x, y \leq 11$. Sie können so verfolgen, wie Ihre Strategie auf Ihre Züge reagiert. Ein Beispiel hierfür sehen Sie in Abb. 2.

Sie können auch zwei Strategien gegeneinander spielen lassen (oder eine Strategie gegen sich selbst) mit

```
*Exercise_8> playGame (simpleStrategy nextMove,
                       simpleStrategy nextMove)
```

Für die Hausaufgabe ist es bereits ausreichend, wenn Ihre Funktion immer einen gültigen Zug zurückzugibt. Für den Wettbewerb wird der MC Ihre Strategie gegen die Strategien der anderen Studenten antreten lassen und ein Ranking erstellen. Für den Wettbewerb gelten zusätzlich die folgenden Hinweise:

- Sie sollten Ihre Züge schnell berechnen. Der MC wird nicht mehr als einige Sekunden pro Zug warten.
- Anstatt von `nextMove` können Sie auch ein erweitertes Interface verwenden, das Ihnen erlaubt, Zufall zu benutzen und einen Zustand mitzuführen. Details hierzu finden Sie im Aufgabentemplate. Bei Fragen hierzu fragen Sie im Piazza oder wenden Sie sich an den MC Jr. Manuel Eberl.

Wichtig: Wenn Sie diese Aufgabe als Wettbewerbsaufgabe abgeben, stimmen Sie zu, dass Ihr Name ggf. auf der Ergebnisliste auf unserer Internetseite veröffentlicht wird. Sie können diese Einwilligung jederzeit widerrufen, indem Sie eine Email an `fp@fp.in.tum.de` schicken. Wenn Sie nicht am Wettbewerb teilnehmen, sondern die Aufgabe allein im Rahmen der Hausaufgabe abgeben möchten, lassen Sie bitte die {`-WETT-`}...{`-TTEW-`} Kommentare weg. Bei der Bewertung Ihrer Hausaufgabe entsteht Ihnen hierdurch kein Nachteil.

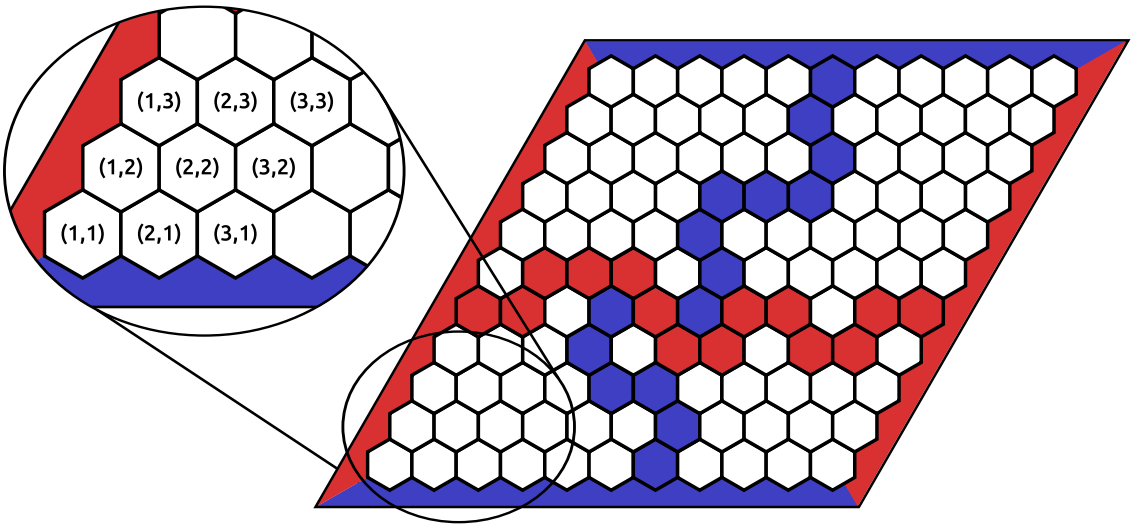

Abbildung 1: Ein Spiel, in dem der rote Spieler gewonnen hat (rechts) sowie ein Ausschnitt aus der linken unteren Ecke des Spielfelds mit korrekter Nummerierung (links).

```
Ok, modules loaded: Exercise_8, Game, GameUtils.
*Exercise_8> playGameInteractive (simpleStrategy nextMove) Red
          ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
         ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
        ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
       ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
      ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
     ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
    ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
   ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
  ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
 ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
● ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
Please enter your move.
2 3
          ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
         ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
        ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
       ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
      ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
     ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
    ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
   ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
  ○ ◍ ○ ○ ○ ○ ○ ○ ○ ○ ○
 ● ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
● ○ ○ ○ ○ ○ ○ ○ ○ ○ ○
Please enter your move.
```

Abbildung 2: Der Anfang eines interaktiven Spiels gegen eine Strategie.