Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Lars Noschinski, Lars Hupel, Dr. Jasmin Blanchette

WS 2013/14
03.11.2012
Abgabe: 10.12.2012, 15:30

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]
   ```

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 Haufenweise Bäume (3 Punkte)

Abgesehen von der Suchbaumeigenschaft gibt es noch eine andere wichtige Invariante auf binären Bäumen. Ist bei einem Baum der Wert eines Knotes kleiner als der Wert jedes Kindknotens, so spricht man von einem *Heap*. Definieren Sie eine Funktion `isHeap :: Ord a => Tree a -> Bool`, die prüft, ob ein Baum die Heap-Eigenschaft erfüllt.

Aufgabe H8.2 Den Wald vor lauter Bäumen nicht finden II (4 Punkte)

Die `navigate`-Funktion hat einen Teilbaum in einem Baum lokalisiert. Nun wollen wir einen Teilbaum *ersetzen*.

Definieren Sie dazu eine Funktion `replace :: [Direction] -> Tree a -> Tree a -> Maybe (Tree a)`. Für einen Aufruf `replace` ds t' t soll im Baum t der Teilbaum an der Position ds durch den Baum t' ersetzt werden. Falls solch ein Baum nicht existiert, dann soll `Nothing` zurückgegeben werden.

Beispiele:

```
replace [] Empty (Node "hello" Empty Empty)
  == Just Empty

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

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

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

Aufgabe H8.3 Law and Ordering (4 Punkte)

Die Standardbibliothek bietet in der Typklasse `Ord` die gängigen Vergleichsoperatoren wie `(<)`. Für einen 3-Wege-Vergleich („a ist kleiner, gleich, oder größer als b“?) ist das aber ungeeignet, weil man zwei `if then else` schachteln muss.

Dafür gibt es eine Operation mit eigenem Datentyp:

```
data Ordering = LT | EQ | GT

compare :: Ord a => a -> a -> Ordering
```

Implementieren Sie `find` für `Tree a` erneut, aber ohne Verwendung von `(==)`, `(<)` o.ä., sondern nur mit Verwendung von `compare`.

Aufgabe H8.4 To be or not to be (4 Punkte)

1. Definieren Sie eine Funktion `maybeMap :: (a -> Maybe b) -> [a] -> Maybe [b]` mit folgendem Verhalten:

   $$\texttt{maybeMap}\ f\ [x_1, x_2, \ldots, x_m] = \texttt{Just}\ [y_1, y_2, \ldots, y_m]$$

   falls $f\ x_i = \texttt{Just}\ y_i$ für alle $1 \leq i \leq m$ und

   $$\texttt{maybeMap}\ f\ [x_1, x_2, \ldots, x_m] = \texttt{Nothing}$$

   sonst.

2. Schreiben Sie eine Funktion `inverses :: [Integer] -> Maybe [Rational]`, die zu einer Liste von Integern die Liste ihrer Kehrwerte berechnet. Ist einer der Integer 0, so soll `Nothing` zurückgegeben werden. Verwenden Sie die Funktion `maybeMap`.

   Beispiele:

   ```
   inverses [1,2,3] = Just [1 % 1, 1 % 2, 1 % 3]
   inverses [3,0,4] = Nothing
   ```

Aufgabe H8.5 `http://xkcd.com/208/` (Wettbewerbsaufgabe, 5 Punkte)

Reguläre Ausdrücke (kurz: *Regexes*) sind aus der Automatentheorie bekannt. Sie beschreiben Mengen von Strings. Unix-Programme wie `grep` und `sed` basieren auf Regexes, jedoch jeweils mit einer leicht abweichenden Syntax. Nach Donald Knuth: „I define UNIX as 30 definitions of regular expressions living under one roof.“

Diese Aufgabe führt die 31. Definition ein. Für beliebige Regexes r und s sei:

1. Das Sonderzeichen `.` (Punkt) ist ein Regex, der ein beliebiges Zeichen erkennt.
2. Ein normales Zeichen c ist ein Regex, der c erkennt.
3. Ein Zeichenbereich `[`$s_1 \ldots s_n$`]` ($n \geq 0$) ist ein Regex, der ein beliebiges Zeichen aus einem beliebigen Teilbereich s_j erkennt. Ein *Teilbereich* ist als c`-`d angegeben und steht für alle Zeichen e, für die $c \leq e \leq d$ gilt. Der Teilbereich `a-z` steht beispielsweise für `a` oder `b` oder ... oder `y` oder `z`.
4. Der Regex rs (Verkettung) erkennt Strings, deren Beginn von r und deren Ende von s erkannt wird (ohne etwas dazwischen).
5. Der Regex r`|`s (Auswahl) erkennt Strings, die von r oder von s erkannt werden.
6. Der Regex r`+` erkennt ein oder mehrere Vorkommen von Strings, die jeweils von r erkannt werden.

Beim Parsen hat der Wiederholungsoperator `+` Vorrang vor dem Verkettungsoperator, der wiederum Vorrang vor dem Auswahloperator hat. Zum Beispiel ist der Regex `ab|cd+` als `(ab)|(c(d+))` zu verstehen und erkennt genau die Wörter aus der Menge {`ab`, `cd`, `cdd`, `cddd`, ...}.

In Haskell ist es natürlich, die regulären Ausdrücke als Datentyp zu implementieren:

```
data RegEx =
  Any |
  One Char |
  OneIn [(Char, Char)] |
  Concat RegEx RegEx |
  Alt RegEx RegEx |
  Repeat RegEx
```

Ihre Aufgabe besteht darin, eine Funktion

```
match :: RegEx -> String -> Bool
```

zu implementieren, die prüft, ob eine Regex ein Wort erkennt.

Die folgenden Beispiele sollten zu `True` auswerten:

```
match Any "x"
not $ match Any ""
not $ match Any "xy"
match (One 'x') "x"
not $ match (One 'y') "x"
match (OneIn [('a', 'c'), ('x', 'z')]) "x"
not $ match (OneIn [('a', 'c'), ('x', 'z')]) "d"
not $ match (OneIn [('a', 'c'), ('z', 'x')]) "x"
match (Concat (One 'x') (One 'y')) "xy"
not $ match (Concat (One 'x') (One 'y')) "x"
not $ match (Concat (One 'x') (One 'y')) "yx"
match (Alt (One 'x') (One 'y')) "x"
match (Alt (One 'x') (One 'y')) "y"
not $ match (Alt (One 'x') (One 'y')) ""
not $ match (Alt (One 'x') (One 'y')) "z"
match (Repeat (Concat (One 'x') (One 'y'))) "xy"
match (Repeat (Concat (One 'x') (One 'y'))) "xyxyxyxy"
not $ match (Repeat (Concat (One 'x') (One 'y'))) ""
```

(Zur Erinnerung: `f $ bla bla bla` ist das gleiche wie `f (bla bla bla)`.)

Für den Wettbewerb zählt die Effizienz, vor allem die Skalierbarkeit. Ihre Implementierung sollte mit großen Ausdrücken, langen Wörtern und absurd vielen Duplikaten möglichst effizient vorgehen. Lösungen, die sich effizienztechnisch ähnlich verhalten, werden bezüglich ihrer Lesbarkeit verglichen, wobei Formatierung und Namensgebung besonders gewichtet werden.

Die Lösung muss innerhalb der Kommentare {-WETT-} und {-TTEW-} abgegeben werden, um als Wettbewerbsbeitrag zu zählen.

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.