Praktische Informatik 3 WS 20/21

Christoph Lüth
Thomas Barkowsky
Tobias Brandt
Alexander Krug
Tarek Soliman
Robert Sachtleben

6. Übungsblatt

Ausgabe: 14.12.2020
Abgabe: 21.12.2020 12:00

In dieser Woche wollen wir uns ein wenig mit literarischen Texten befassen. Wir wollen die in einem gegebenen Text vorkommenden Wörter erfassen und organisieren, und wir wollen verzeichnen, in welchen Zeilen eines Textes diese Wörter jeweils vorkommen. Am Ende können wir dann einen alphabetischen Index erstellen, der uns darüber informiert, wo im Text die jeweiligen Wörter zu finden sind.

6.1 *Von Texten und Bäumen* *4 Punkte*

Wir wollen zunächst Wörter und deren Auftreten in bestimmten Zeilen eines Textes verwalten und dann den gesamten Text in einem binären Suchbaum organisieren.

Wir definieren uns hierfür zunächst den folgenden Datentyp:

```
data WEntry = WEntry { word :: String
                     , occurrences :: [Int]
                     } deriving Show
```

Die Wörter unseres Textes wollen wir in einem lexikographisch sortierten Binärbaum speichern. Hierbei sollen die Wörter in den rechten Teilbäumen jeweils kleiner sein als die in den linken.

Wir nutzen dafür folgenden Datentyp:

```
data TTree a = WNode { rightbranch :: TTree a
                     , entry :: a
                     , leftbranch :: TTree a}
                     | Empty
                     deriving Show
```

Jetzt schreiben wir uns zwei Funktionen, mit denen wir das Auftreten von Wörtern mit ihren Zeilennummern erfassen können, sowie eine Funktion, die ein Wort in Verbindung mit einer Zeilennummer in unseren Textbaum einfügt:

```
addOccurrence :: WEntry → Int → WEntry
insertWord :: String → Int → TTree WEntry → TTree WEntry
```

Die erste Funktion erzeugt einen Eintrag für ein Wort oder fügt einem existierenden Wort eine weitere Zahl hinzu.

```
addOccurrence (WEntry "Hund" []) 5 ⇝ WEntry {word = "Hund", occurrences = [5]}
addOccurrence WEntry {word = "Hund", occurrences = [5]} 13 ⇝
   WEntry {word = "Hund", occurrences = [5,13]}
```

Die Funktion `insertWord` nimmt ein Wort zusammen mit einer Zeilennummer in unseren Text-Baum auf. Wenn das Wort bereits im Baum enthalten ist, dann wird nur die neue Zeilennummer hinzugefügt. Ansonsten wird das Wort mit der übergebenen Zeilennummer in den Baum aufgenommen, und zwar so, dass lexikographisch kleinere Wörter nach rechts, größere nach links eingeordnet werden:

```
insertWord "Hund" 4 Empty ⇝
   WNode {rightbranch = Empty, entry = WEntry {word = "Hund", occurrences = [4]},
   leftbranch = Empty}
insertWord "Hund" 9 it ⇝
   WNode {rightbranch = Empty, entry = WEntry {word = "Hund", occurrences = [4,9]},
   leftbranch = Empty}
insertWord "Alf" 13 it ⇝
   WNode {rightbranch = WNode {rightbranch = Empty,
   entry = WEntry {word = "Alf", occurrences = [13]}, leftbranch = Empty},
   entry = WEntry {word = "Hund", occurrences = [4,9]}, leftbranch = Empty}
```

Jetzt können wir uns vornehmen, einen kompletten Text (d.h. einen String mit `\n` als Zeilenseparator) in unseren Baum einzulesen:

```
textToTTree :: String → TTree WEntry
```

Hierfür schreiben wir uns zunächst zwei Hilfsfunktionen:

```
wordsWithLine :: String → [(String, Int)]
wordListToTree :: [(String, Int)] → TTree WEntry → TTree WEntry
```

Die erste Funktion erzeugt aus einem Text eine Liste aus Paaren von Wörtern und Zeilennummern:

```
wordsWithLine "eins␣zwei\ndrei␣vier\nfuenf␣sechs" ⇝
   [("eins",1),("zwei",1),("drei",2),("vier",2),("fuenf",3),("sechs",3)]
```

In `wordsWithLine` sollen die Wörter des Textes auch gleich in Kleinbuchstaben gewandelt werden und alle nichtalphabetischen Zeichen (bspw. Interpunktionszeichen) sollen entfernt werden.

`wordListToTree` wandelt dann diese Liste in einen Textbaum um. Jetzt können wir auch leicht die Funktion `textToTTree` realisieren.

6.2 *Text-Origami* *6 Punkte*

Unseren Textbaum wollen wir nun für die Analyse des zugrundeliegenden Textes nutzen. Hierfür wäre es natürlich schön, wenn wir auf der Struktur *mapping* und strukturelle Rekursion zur Verfügung hätten.

Wir machen unseren `TTree` also zunächst zu einer Instanz der Typklasse `Functor`:

```
instance Functor TTree where...
```

Und da wir unseren Datentyp auch falten können wollen, definieren wir uns die Funktion `foldTTree` analog zu `foldr` für Listen:

```
foldTTree :: (b → a → b → b) → b → TTree a → b
```

Uff! Jetzt wird es aber Zeit für ein paar kleine Anwendungen von `fmap` und `foldTTree`.

Wir schreiben uns folgende Funktionen:

```
sizeOfVocabulary :: TTree WEntry → Int
wordFrequency :: TTree WEntry → String
whereDoesThisWordOccur :: TTree WEntry → String → [Int]
```

Die erste Funktion liefert uns den Umfang des Vokabulars eines Textes zurück (also die Anzahl der *verschiedenen* verwendeten Wörter):

```
sizeOfVocabulary $ textToTTree raven ⇝ 443
```

(Den Text von Edgar Allan Poes Gedicht *The Raven* findet ihr in der Vorlage.)

Die zweite Funktion listet uns die verwendeten Wörter in alphabetischer Reihenfolge (unser Baum ist ja sortiert...) zusammen mit der Häufigkeit ihres Auftretens:

```
putStr $ wordFrequency $ textToTTree raven ⇝
a 15
above 7
adore 1
again 1
agreeing 1
ah 2
aidenn 1
air 1
all 4
allan 1...
```

Die dritte Funktion gibt aus, wo überall im Text ein bestimmtes Wort zu finden ist:

```
whereDoesThisWordOccur (textToTTree raven) "raven" ⇝ [1,46,55,57,66,80,99,106,113,120,122]
```

Zu guter Letzt wollen wir noch unseren Index ausgeben können, also die alphabetische Liste aller Wörter zusammen mit den Zeilen, in denen diese zu finden sind:

```
putStr $ makeIndex $ textToTTree raven ⇝
...
but[26,33,48,66,80,87,91]
by[2,53,73,95,104,109]
came[5,26,27]
cannot[61]
caught[75]
censer[94]
chamber[6,7,20,21,27,38,48,49,62,63,123]
clasp[111,112]
core[88]
countenance[53]...
```

Viel Spaß – und gruselt euch nicht gar zu sehr...