Prof. Dr. Thomas Breuel
Dipl.-Inf. Patrick Michel (AG ST)
Dipl.-Ing. Ludwig Schmidt-Hackenberg

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

Übungsblatt 7: Software-Entwicklung 1 (WS 2010/11)

Ausgabe: in der Woche vom 06.12. bis zum 10.12.10
Abgabe: in der Woche vom 13.12. bis zum 17.12.10
Abnahme: max. zwei Tage nach der Übung

Aufgabe 1 Sortieren (Präsenzaufgabe)

In dieser Aufgabe sollen die in der Vorlesung vorgestellten Sortieralgorithmen am Beispiel geübt werden. Legen Sie sich selbst eine Reihenfolge fest, in der Sie die Beispiele lösen, so dass Sie frühzeitig nachfragen können bei Problemen.

a) Sortieren Sie die Liste `[7, 5, 9, 8, 4, 3, 6]` aufsteigend, einmal mit Hilfe des **Selectionsort** Algorithmus und einmal mit Hilfe des **Insertionsort** Algorithmus. Geben Sie Ihre Lösungen jeweils mit sinnvollen Zwischenschritten an, z.B. wie in der Vorlesung angedeutet!

b) Sortieren Sie die gleiche Liste aufsteigend mit Hilfe des **Bubblesort** Algorithmus. Geben Sie dabei alle Zwischenschritte an, bei denen sich etwas in der Liste geändert hat! Geben Sie die Liste nach einem rekursiven Durchlauf an, entfernen Sie das maximale Element und schreiben Sie es dahinter.

c) Sortieren Sie die Liste `[10, 55, 13, 42, 7]` aufsteigend, mit Hilfe des **Mergesort** Algorithmus. Verwenden Sie zur Darstellung Ihrer Lösung die Baum bzw. Graphdarstellung aus der Vorlesung.

d) Sortieren Sie die Liste `[12, 10, 55, 17, 13, 42, 7]` aufsteigend, mit Hilfe des **Quicksort** Algorithmus. Verwenden Sie zur Darstellung Ihrer Lösung eine Baumdarstellung wie aus der Vorlesung.

e) Sortieren Sie die Liste `[23, 1701, 42, 815, 13]` absteigend, mit Hilfe des **Heapsort** Algorithmus. Geben Sie dabei alle entstehenden Bäume an und markieren Sie welche die Heap-Eigenschaft verletzen.

Aufgabe 2 Sortieren (Einreichaufgabe)

a) Implementieren Sie die Sortieralgorithmen der Vorlesung noch einmal selbst in Haskell, der Einfachheit halber für Listen von ganzen Zahlen. Heapsort müssen Sie *nicht* neu schreiben. Schreiben Sie also die fünf `[Integer] -> [Integer]` Funktionen:

```
selectionsort, insertionsort, bubblesort, quicksort, mergesort
```

b) Überlegen Sie sich wie viele Vergleiche "`a < b`" (bzw. `leq` in den Folien) zwischen Elementen beim *Selectionsort* Algorithmus, auf einer Liste der Länge n, ausgeführt werden. Geben Sie eine möglichst geschlossene Form an.

c) Überlegen Sie sich wie viele Vergleiche "`a == b`" zwischen Elementen beim *Selectionsort* Algorithmus, auf einer Liste der Länge n, ausgeführt werden. Geben Sie eine maximale („worst-case") und eine minimale („best-case") Anzahl an.

d) Wie viele Vergleiche "`a < b`" fallen beim *Insertionsort* Algorithmus an, wiederum auf einer Liste der Länge n?

Aufgabe 3 Binäre Suche auf Listen (Einreichaufgabe)

In der Vorlesung haben wir die abstrakte Datenstruktur `Dictionary` mit Hilfe von Binärbäumen implementiert, um ihre Operationen – vor allem auch die Suche – zu optimieren. In dieser Aufgabe wollen wir die Datenstruktur mit Listen implementieren und schauen, ob wir die Suche trotzdem besser als linear implementieren können.

Bei der *binären Suche* auf einer Liste wird diese in zwei etwa gleich große Teillisten aufgespalten, danach entschieden, in welcher Teilliste das Element enthalten sein könnte und die binäre Suche rekursiv auf dieser Teilliste erneut ausgeführt. Dies wird solange wiederholt, bis das gewünschte Element gefunden ist oder aber klar ist, dass dieses nicht in der Liste vorkommt.

a) Welche Eigenschaft muss eine Liste erfüllen, damit eine binäre Suche durchgeführt werden kann?

b) Implementieren Sie die Schnittstelle von `Dictionary`, mit Hilfe von Haskell Listen. Sie können sich dazu die Datei "`Dictionary.hs`" von der Vorlesungsseite laden und erweitern. Achten Sie bei der `put` Operation darauf die in a) identifizierte Eigenschaft auf dieser Liste zu erhalten.

   Implementieren Sie die `get` Operation mittels binärer Suche!

c) Wieviele Vergleiche mit Listenelementen sind bei einer `get` Operation auf einem `Dictionary` der Länge $n = 2^k$ mit $n, k \in \mathbb{N}$ durchzuführen, wenn das gesuchte Element nicht im `Dictionary` enthalten ist?

Aufgabe 4 Robosort (freiwillige Zusatzaufgabe)

Wie wir auf Blatt 3 schon gesehen haben, kann auch der Roboter einen unsortierten Eingabestapel sortieren! Insbesondere die Lösungen zu den Aufgaben 2b und 2d geben Ihnen ein Grundprimitiv, mit dem Sie direkt zwei der in der Vorlesung betrachteten Sortieralgorithmen implementieren können.

a) Schreiben Sie drei Roboterprogramme `selectionsort`, `insertionsort` und `bubblesort`, die jeweils einen unsortierten Anfangsstapel mit beliebig vielen Karten sortieren.

   Hinweis: Sie können für diese Aufgabe auch den vierten Stapel D verwenden.

b) Wir möchten dem Roboter nun auch `quicksort` und `mergesort` beibringen. Um dies einfacher zu gestalten wurden folgende Änderungen am Roboter vorgenommen:

   - Der Befehl `Hole` führt auf einem leeren Stapel nicht mehr zum Fehler, sondern wird ignoriert. Analog dazu führt nun auch der Befehl `Lege` mit leerem Arm nicht mehr zu einem Fehler. Die Vergleichsoperation funktioniert nun auch wenn der Roboter nur eine Karte in der Hand hält, wobei er eine fehlende Karte als "sehr groß" interpretiert.
   - Schließlich wurden zwei neue Operationen `Sperre Stapel` und `Entsperre Stapel` hinzugefügt, die eine Sperrkarte auf einen Stapel legen, bzw. eine an oberster Stelle liegende Sperrkarte entfernen. Alle anderen Operationen sehen einen Stapel, auf dem oben eine Sperre liegt, als leer an.

   Laden Sie sich die neue "`RoboLib.hs`" von der Vorlesungsseite und entwickeln Sie die beiden Roboterprogramme `quicksort` und `mergesort`.

   Kommen Sie auch ohne Sperren und/oder unendlich große Programme (Haskell-Rekursion) aus?

c) Testen und vergleichen Sie die verschiedenen Algorithmen und Implementierungen. Denken Sie dabei an die Kommandozeilen-Option "`-r`" und die Tasten `'<'`, bzw. `'>'`, mit denen Sie stabile Anfangsstapel erzeugen können. Die Schrittanzeige zeigt Ihnen nun auch die Anzahl der Vergleiche an. Die Funktion `stepcount :: Programm -> [Int] -> Maybe (Int, Int)` erlaubt es Ihnen für ein Programm und einen Anfangsstapel die Anzahl der Schritte und Vergleiche zu ermitteln, insofern das Programm nicht abstürzt oder endlos laufen würde.

   Warnung: Achten Sie auf Endlosschleifen, die immer eine neue Sperre erzeugen, da diese nicht erkannt werden.

Aufgabe 5 Map-Reduce (Einreichaufgabe)

Wir betrachten die Funktion `mapreduce :: (a -> b) -> (b -> b -> b) -> [a] -> b`, aus der Vorlesung. Laden Sie sich die Datei "`mapreduce.hs`" von der Vorlesungsseite, in der diese Funktion bereits definiert ist, alle Funktionsrümpfe stehen und einfache Tests definiert sind. Was Sie nun noch brauchen ist die Datei "`shakespeare-romeo-48.txt`", die Sie auf unserer Webseite verlinkt finden und ebenfalls herunterladen und abspeichern sollen.

Hinweis: Verwenden Sie `mapreduce` *um folgende Probleme zu lösen. Das heißt Sie sollen wenn möglich das Ergebnis von* `mapreduce` *nicht mehr verändern und nur die beiden Parameterfunktionen frei wählen.*

a) Schreiben Sie eine Funktion `countWord :: String -> [String] -> Int`, die berechnet wie oft ein Wort in einer Liste von Wörtern vorkommt, wobei Groß- und Kleinschreibung ignoriert wird.

b) Schreiben Sie eine Funktion `countChar :: Char -> [String] -> Int`, die berechnet wie oft ein Buchstabe in einer Liste von Wörtern vorkommt, wobei Groß- und Kleinschreibung ignoriert wird.

c) Schreiben Sie eine Funktion `wordsWithLength :: Int -> [String] -> [String]`, die alle Wörter mit gegebener Länge auflistet, wobei jedes Wort nur *einmal* im Ergebnis sein soll und auch hier Groß- und Kleinschreibung keine Rolle spielen sollen.

d) Schreiben Sie eine Funktion `wordsContaining :: Char -> [String] -> [String]`, die alle Wörter die ein bestimmtes Zeichen enthalten auflistet, wobei jedes Wort nur *einmal* im Ergebnis sein soll und auch hier Groß- und Kleinschreibung keine Rolle spielen sollen.

e) Schreiben Sie eine Funktion `upperCaseWords :: [String] -> [String]`, die alle Wörter, die nur aus Großbuchstaben bestehen, auflistet, wobei alle Wörter nur *einmal* im Ergebnis auftauchen sollen.

f) Schreiben Sie eine Funktion `longestWord :: [String] -> String`, die das längste Wort einer Liste von Wörtern ermittelt.

g) *(freiwillige Zusatzaufgabe)* Schreiben Sie die Funktion `histogram :: [String] -> [(String, Int)]`, welche eine Histogramm für die Eingabe berechnet, sprich eine Liste aller Worte zusammen mit ihrer Häufigkeit.

In dieser Aufgabe wird es sehr hilfreich sein verschiedene Funktionen aus `Data.List` und `Data.Char` zu verwenden. Es lohnt sich insgesamt sich mit diesen Modulen vertraut zu machen. Hilfreich für diese Aufgabe könnten zum Beispiel sein:

```
and  or  any  all  isUpper  isLower  toUpper  toLower
map  nub  union  intersect  reverse  concat  filter  lookup
```

Aufgabe 6 Parameterinduktion (Präsenzaufgabe)

Geben Sie die Implementierung der Haskell-Funktion `multiply` an, welche die Multiplikation zweier *natürlicher* Zahlen auf Addition / Subtraktion zurückführt. Zeigen Sie für Ihre Implementierung unter Verwendung der *vollständigen Induktion*, dass die folgende Gleichheit gilt:

$$\texttt{multiply x y} = x \cdot y \qquad \forall x, y \in \mathbb{N}$$

Aufgabe 7 Parameterinduktion (Einreichaufgabe)

a) Geben Sie die Implementierung der Haskell-Funktion `sumquad` an, welche für gegebenen Parameter `n` die Summe der Quadratzahlen von 0 bis n^2 durch rekursive Berechnung ermittelt. Zeigen Sie für Ihre Implementierung unter Verwendung der *vollständigen Induktion*, dass die folgende Gleichheit gilt:

$$\texttt{sumquad n} = \frac{n(n+1)(2n+1)}{6}$$

b) Beweisen Sie, dass die Haskell-Funktionen

```
g :: Integer -> Integer
g n = h n 0 1  where

  h 0 x _ = x
  h n x y = h (n-1) (x + y) x
```

und

```
f :: Integer -> Integer
f 0 = 0
f 1 = 1
f n = f (n - 1) + f (n - 2)
```

für natürliche Zahlen gleich sind, also `g n = f n` für alle $n \in \mathbb{N}$. Zeigen Sie hierzu zunächst mittels *Induktion* über k, dass

```
h (n-k) (f k) (f (k-1)) = f n
```

für $n, k \in \mathbb{N}$ mit $1 \leq k \leq n$ und $n > 0$ gilt. Nutzen Sie dieses Teilergebnis, um die eigentliche Aussage zu beweisen.

Hinweis: Im Fall obiger Induktion *über k ist es ratsam, bei k = n zu beginnen und den Induktionsschluß* $k \to (k-1)$ *durchzuführen!*

Aufgabe 8 Begriffe der prozeduralen Programmierung

Kreuzen Sie an, ob folgende Aussagen wahr oder falsch sind. Bereiten Sie diese Aufgabe bis zu Ihrer nächsten Übungsstunde vor, so dass Sie bei Unklarheiten nachfragen und die Antworten diskutieren können.

wahr	falsch	
☐	☐	Algorithmen beschreiben Abläufe.
☐	☐	Aktionen werden auf Zuständen ausgeführt.
☐	☐	Die Ausführung eines Algorithmus kann im Prinzip unendlich lange dauern.
☐	☐	Der Zustand einer Variable wird durch eine Zuweisung nicht verändert.
☐	☐	Ein Algorithmus kann in verschiedenen Sprachen beschrieben werden.
☐	☐	Der Ausführungszustand wird durch die Position im Algorithmus bestimmt.
☐	☐	Beim Ausführen *eines* Schrittes des Algorithmus können sich sowohl Speicher- als auch Steuerungszustand ändern.
☐	☐	Beim Ausführen einer Aktion verändert sich der Steuerungszustand.
☐	☐	Es gibt Algorithmen bei deren Ausführung unabhängig von den Eingaben immer der gleiche Ablauf entsteht.
☐	☐	Die Beschreibung eines effizienten Algorithmus ist in der Regel kürzer als die eines weniger effizienten.
☐	☐	Auch ein Java-Programm mit Fallunterscheidungen ist deterministisch.
☐	☐	Die Auswertung einer Haskell-Funktion ist deterministisch.
☐	☐	Es gibt deterministische determinierte Algorithmen.
☐	☐	Es gibt nicht-deterministische determinierte Algorithmen.
☐	☐	Ein für alle Eingaben nicht-terminierender Algorithmus ist nicht-determiniert.
☐	☐	Im prozeduralen Programmierparadigma werden Prozeduren als Abstraktionsmittel verwendet.