RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

Lehrstuhl für Hochleistungsrechnen
RWTH Aachen · D-52074 Aachen · GERMANY
http://programmierung.informatik.rwth-aachen.de

Prof. Christian Bischof, Ph.D.
Andre Vehreschild, Jakob T. Valvoda

Übung *Programmierung WS 06/07* – Blatt 11

Lösungen müssen bis zum **22. Januar 2007, 17:00 Uhr** in den Kasten Ihrer Übungsgruppe eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55). Bitte vergessen Sie nicht, die **Nummer Ihrer Übungsgruppe**, sowie die **Namen** und **Matrikelnummern** der Mitglieder Ihrer 2er-Gruppe auf jedes Lösungsblatt Ihrer Abgabe zu schreiben. Bitte heften Sie ihre Blätter zusammen.

Alle erstellten Haskell-Programme sind sowohl in gedruckter Form abzugeben, als auch per E-Mail an den jeweiligen Tutor zu senden. Vergessen Sie auch bei der elektronischen Abgabe per E-Mail nicht die Angabe der **Nummer Ihrer Übungsgruppe**, sowie die jeweiligen **Namen** und **Matrikelnummern**.

Aufgabe 1 (0.5 + 0.5 + 0.5 + 0.5 + 0.5 + 0.5 = 3 Punkte)

Seien `x` und `y` ganze Zahlen und `xs` und `ys` Listen, in welchen ganze Zahlen gespeichert sind. Geben Sie an, welche der folgenden Listenpaare gleich sind. Begründen Sie Ihre Antwort.

a) `x : []` und `[x]`

b) `xs : []` und `xs`

c) `[] : xs` und `xs`

d) `x : y` und `[x,y]`

e) `xs : ys` und `xs ++ ys`

f) `(x : (y : xs)) ++ ys` und `[x] ++ ([y] ++ (xs ++ ys))`

Hinweis: Der Operator `++` verkettet zwei Listen, d.h. für zwei Listen `xs` = `[`$x_1, \ldots, x_n$`]` und `ys` = `[`$y_1, \ldots, y_m$`]` liefert `xs ++ ys` = `[`$x_1, \ldots, x_n, y_1, \ldots, y_m$`]`.

Aufgabe 2 (1 + 1.5 + 1.5 = 4 Punkte)

Analysieren Sie das folgende Haskell-Programm

```
magic :: Int -> Int -> [Int]
magic 0 _ = []                    -- m1
magic m n = m : (magic n (m+n))   -- m2

getIt :: [Int] -> Int -> Int
```

```
getIt [] _ = 0                    -- g1
getIt (x:xs) 1 = x                -- g2
getIt (x:xs) n = getIt xs (n-1)   -- g3
```

Geben Sie alle Zwischenschritte bei der Auswertung der folgenden Ausdrücke an. Geben Sie zudem bei jedem Funktionsaufruf die verwendete Deklaration an: `m1`, `m2` sowie `s1`, `s2` oder `s3`. Sollte die Berechnung eines Ausdrucks nicht terminieren, dann geben Sie dies an und brechen Sie die Auswertung nach dem 4. Schritt ab.

a) `getIt (magic 1 1) 0`

b) `getIt (magic (-3) 2) 0`

c) `getIt (magic 1 1) 3`

Hinweis: Beachten Sie, dass Haskell Ausdrücke nicht-strikt (nach der Leftmost-Outermost-Strategie) auswertet. Hierbei wird jeweils der am weitesten links außen stehende Teilausdruck ausgewertet. Vergleichen Sie hierzu Folie 4 im Kapitel III.2 Bei Pattern Matching werden die Argumente nur soweit ausgewertet, dass klar wird, welches Pattern zutrifft. Auch hier werden (soweit nötig) die Argumente schrittweise von links nach rechts betrachtet.

Aufgabe 3 (1 + 1 + 1 + 2 = 5 Punkte)

Implementieren Sie folgende Listen-Funktionen in Haskell.

a) `letztes :: [a] -> a`, welche das letzte Element der Eingabeliste zurückliefert.
Beispiel: `letztes [5,4,3]` → `3`

b) `verknuepfe :: [a] -> [b] -> [(a,b)]`, welche komponentenweise die Elemente zweier Eingabelisten zu einem Zweiertupel verknüpft und diese in einer Liste kombiniert. Falls eine Liste länger als die andere ist, werden die überschüssigen Elemente weggelassen.
Beispiel: `verknuepfe [5,4,3,2,1] ['a','b']` → `[(5,'a'),(4,'b')]`

c) `wiederhole :: Int -> a -> [a]`, welche ein Liste mit der angegebenen Anzahl von Kopien des zweiten Arguments erzeugt.
Beispiel: `wiederhole 5 'x'` → `"xxxxx"`

d) `wende :: [a] -> [a]`, kehrt die Reihenfolge der Elemente in einer Liste um.
Beispiel: `wende [5,4,3]` → `[3,4,5]`.

Verwenden Sie bei der Implementierung der obigen Funktionen bis auf den Operator `+` und `-` zur Addition bzw. Subtraktion natürlicher Zahlen keine von Haskell vordefinierten Funktionen und auch nicht den Operator `++`. Sie können evtl. benötigte Funktionalität in separaten Hilfsfunktionen implementieren.

Aufgabe 4 (5 + 3 + 2 + 1 = 11 Punkte)

In dieser Aufgabe sollen mehrere Funktionen zur Verarbeitung und Formatierung von Texten implementiert werden. Ein Text ist eine Zeichenkette vom Typ `String` und besteht aus einzelnen Wörtern, welche durch sog. Whitespace-Zeichen (dazu gehören Leerzeichen, Tabulatoren, Zeilenumbrüche, etc.) getrennt werden. Ihr Programm soll in der Lage sein, einzelne Wörter zu extrahieren und sie in Zeilen vorgegebener Länge anordnen. So wird der Text

```
In dieser            Aufgabe
 sollen  mehrere   Funktionen
zur   Verarbeitung und          Formatierung
   von            Texten  implementiert
werden.
```

für eine Zeilenlänge von 40 wie folgt verarbeitet

```
In dieser Aufgabe sollen mehrere Funktionen
zur Verarbeitung und Formatierung von Texten
implementiert werden.
```

In den nachfolgenden Beispielen ist zur Verdeutlichung das normale Leerzeichen als ␣ dargestellt.

a) Implementieren Sie die folgenden Funktionen, welche aus einem Eingabetext eine Wortliste berechnen:

- `holeWort :: String -> String`, welche einen Text übergeben bekommt und das erste Wort zurückliefert. Falls der Text mit einem Whitespace beginnt, dann ist das Resultat die leere Liste.
  Beispiel: `holeWort "Dies␣␣␣ist"` → `"Dies"` und `holeWort "\n\nLeer␣"` → `[]`
- `loescheWort :: String -> String`, welche das erste Wort in dem übergebenen Text löscht und den gekürzten Text zurückliefert. Falls der Text mit einem Whitespace beginnt, dann wird der unveränderte Eingabetext zurückgegeben.
  Beispiel: `loescheWort "Dies␣␣␣ist"` → `"␣␣␣ist"`
- `loescheWhitespace :: String -> String`, welche alle Whitespace-Zeichen am Anfang des übergebenen Textes bis zum ersten Wort löscht und den bearbeiteten Text zurückliefert. Falls der Text mit keinem Whitespace beginnt, dann wird der Text unverändert zurückgegeben.
  Beispiel: `loescheWhitespace (loescheWort "Dies␣␣␣ist")` → `"ist"`
- `erzeugeWortListe :: String -> [String]`, welche aus einem übergebenen Text eine Liste aus Wörtern berechnet. Nutzen Sie hierbei die bisher implementierten Funktionen.
  Beispiel: `erzeugeWortListe "Dies␣␣␣ist\n␣ein␣␣"` → `["Dies","ist","ein"]`

b) Implementieren Sie weiterhin die folgenden Funktionen, welche aus einer Worliste (vom Typ `[String]`) eine Liste aus Zeilen berechnen. Eine Zeile ist hierbei eine Liste aus Wörtern, also vom Typ `[[String]]`:

- `holeZeile :: Int -> [String] -> [String]`, welche aus einer Liste von Wörtern eine Zeile gegebener Länge extrahiert. Dazu werden Wörter aus der Eingabeliste so lange zu der Ausgabeliste hinzugefügt, bis das erste Mal die angegebene Länge überschritten wird. Hierbei soll ebenfalls das Trennzeichen (ein Leerzeichen) bei der Berechnung der Länge mit berücksichtigt werden.
  Beispiel: `holeZeile 6 ["Dies","ist","ein"]` → `["Dies","ist"]`
- `loescheZeile :: Int -> [String] -> [String]`, löscht analog zu `loescheWort` eine Zeile vorgegebener Länge aus der Eingabeliste und gibt die veränderte Eingabeliste zurück.
- `erzeugeZeilen :: Int -> [String] -> [[String]]`, verwendet die in diesem Aufgabenteil implementierten Funktionen und erzeugt eine Liste von Zeilen vorgegebener Länge.
  Beispiel: `erzeugeZeilen 6 ["Dies","ist","ein"]` → `[["Dies","ist"],["ein"]]`

c) Implementieren Sie schliesslich Funktionen, welche die Wort- und Zeilen-Listen zu einem String zusammenfügen:

- Die Funktion `verbindeWoerter :: [String] -> String`, bildet aus einer Wortliste eine Zeichenkette. Die einzelnen Wörter werden mit Leerzeichen getrennt. Die Zeichenkette soll mit einem Zeilenumbruch (dem Zeichen `'\n'`) abgeschlossen werden.
  Beispiel: `verbindeWoerter ["Dies","ist","ein"]` → `"Dies␣ist␣ein␣\n"`
- Schliesslich sollen Sie die Funktion `verbindeZeilen :: [[String]] -> String` implementieren, welche mit Hilfe von `verbindeWoerter` aus einer Liste von Zeilen den finalen Text zusammensetzt. Verwenden Sie hierzu die bisher implementierten Funktionen.
  Beispiel: `verbindeZeilen [["Dies","ist"],["ein"]]` → `"Dies␣ist␣\nein␣\n"`

d) Testen Sie Ihre Funktionen, indem Sie das Ergebnis der Funktion

```
putStr (verbindeZeilen (erzeugeZeilen 40 (erzeugeWortListe testText)))
```

berechnen und angeben. `putStr` ist in Hugs vordefiniert und wird zur Ausgabe von Strings verwendet. Den Test-String `testText` finden Sie in der Datei `textverarbeitung.hs` auf der Webseite der Vorlesung.

Hinweis: Verwenden Sie zur Überprüfung, ob ein Zeichen `x` ein Whitespace ist, die Liste `leerzeichen = ['\n','\t',' ']` und die Funktion `elem x leerzeichen`, welche wahr ist, wenn `x` ein Leerzeichen ist. Die Liste `leerzeichen` ist ebenfalls in der Datei `textverarbeitung.hs` deklariert, welche Sie auf der Webseite der Vorlesung finden.