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

WS 2014/15
24.10.2014
Abgabe: 31.10.2014

Einführung in die Informatik 2
3. Übung

Aufgabe G3.1 Einfache Listenfunktionen

Implementieren Sie per Rekursion über Listen die folgenden Funktionen:

1. Die Funktion `snoc :: [a] -> a -> [a]` nimmt eine Liste $[x_1, \ldots, x_n]$ und ein Element y und gibt die Liste $[x_1, \ldots, x_n, y]$ zurück. Implementieren Sie diese Funktion ohne ++ zu verwenden.
2. Die Funktion `member :: Eq a => a -> [a] -> Bool` nimmt ein Element t und eine Liste $[x_1, \ldots, x_n]$ und liefert genau dann `True` zurück, wenn es ein $1 \leq i \leq n$ gibt mit $x_i = t$.

   (Diese Funktion ist übrigens Teil vono `Data.List` unter dem Namen `elem`.)
3. Die Funktion `butlast :: [a] -> [a]` nimmt eine Liste $[x_1, \ldots, x_{n-1}, x_n]$ und gibt die Liste $[x_1, \ldots, x_{n-1}]$ zurück. Ist $n = 0$, so wird die leere Liste erwartet.

   Eine ähnliche Funktion ist übrigens Teil der Bibliothek `Data.List` unter dem Namen `init`.

Aufgabe G3.2 Entfernen wiederholter Elemente

1. Schreiben Sie eine Funktion `uniq :: Eq a => [a] -> [a]`, die aufeinanderfolgende gleiche Elemente entfernt. Zum Beispiel sollen die folgenden Gleichheiten gelten:

```
uniq [] == []             uniq [1,2,2,1]   == [1,2,1]
uniq [1,2,3] == [1,2,3]   uniq [1,1,4,1,1] == [1,4,1]
```

   Eine Hilfsfunktion vom Typ `Eq a => a -> [a] -> [a]` kann hilfreich sein.
2. Schreiben Sie eine Funktion `uniqCount :: Eq a => [a] -> [(a, Integer)]`, die zusätzlich zählt, wie viele gleiche Elemente in der Eingabe aufeinander folgen. Zum Beispiel sollen die folgenden Gleichheiten gelten:

```
uniqCount []           == []
uniqCount [1,2,3]      == [(1,1),(2,1),(3,1)]
uniqCount [1,2,2,1]    == [(1,1),(2,2),(1,1)]
uniqCount [1,1,4,1,1] == [(1,2),(4,1),(1,2)]
```

   Eine Hilfsfunktion vom Typ `Eq a => (a, Integer) -> [a] -> [(a, Integer)]` kann hilfreich sein.

Aufgabe G3.3 Trenner

Implementieren Sie die folgenden Funktionen.

1. Die Funktion `intersep :: a -> [a] -> [a]` nimmt ein Element t (den *Trenner*) und eine Liste $[x_1, x_2, \ldots, x_n]$ und liefert die Liste $[x_1, t, x_2, t, \ldots, t, x_n]$ zurück, mit $n-1$ Trennern. Für $n = 0$ wird die leere Liste erwartet. Zum Beispiel:

   ```
   intersep ',' "" == ""
   intersep ',' "a" == "a"
   intersep ',' "ab" == "a,b"
   intersep ',' "abcdef" == "a,b,c,d,e,f"
   intersep 0 [3, 2, 1] == [3, 0, 2, 0, 1]
   ```

   (Diese Funktion ist übrigens Teil von `Data.List` unter dem Namen `intersperse`.)

2. Die Funktion `andList :: [[Char]] -> [Char]` nimmt eine Liste von Wörtern $[w_1, w_2, \ldots, w_{n-1}, w_n]$ und liefert im Allgemeinen den Text „w_1, w_2, …, w_{n-1}, and w_n“ zurück, mit einem Komma vor dem „and“.[1] Es gibt aber Ausnahmen für $n < 3$:

   ```
   andList [] == ""
   andList ["Ayize"] == "Ayize"
   andList ["Bhekizizwe", "Gugu"] == "Bhekizizwe and Gugu"
   andList ["Jabu", "Lwazi", "Nolwazi"] ==
     "Jabu, Lwazi, and Nolwazi"
   andList ["Phila", "Sihle", "Sizwe", "Zama"] ==
     "Phila, Sihle, Sizwe, and Zama"
   ```

Aufgabe G3.4 Dreieck (optional)

Gesucht ist eine polymorphe Funktion `triangle :: [a] -> [(a, a)]`, die eine Liste von Werten $[a_1, \ldots, a_n]$ nimmt und die folgende $n(n-1)/2$-elementige Liste von Paaren zurückgibt:

$$\big[(a_1,a_2),\ (a_1,a_3),\ (a_1,a_4),\ \ldots\ (a_1,a_{n-1}),\ (a_1,a_n),$$
$$(a_2,a_3),\ (a_2,a_4),\ \ldots\ (a_2,a_{n-1}),\ (a_2,a_n),$$
$$(a_3,a_4),\ \ldots\ (a_3,a_{n-1}),\ (a_3,a_n),$$
$$\ddots\ \vdots\ \vdots$$
$$(a_{n-2},a_{n-1}),\ (a_{n-2},a_n),$$
$$(a_{n-1},a_n)\big]$$

Zum Beispiel:

```
triangle [] == []
triangle [1111] == []
triangle [1, 2] == [(1, 2)]
triangle [222, 11] == [(222, 11)]
triangle ["AA", "AA"] == [("AA", "AA")]
triangle [1, 2, 3] == [(1, 2), (1, 3), (2, 3)]
triangle [3, 5, 7, 11] == [(3, 5), (3, 7), (3, 11),
                           (5, 7), (5, 11), (7, 11)]
```

[1] Dieses Komma heißt „serial comma“ oder „Oxford comma“ und ist im Prinzip wahlfrei, wobei es bei amerikanischen Autoren sehr beliebt ist.

Die Funktion lässt sich sowohl rekursiv als auch mit Listenkomprehensionen schreiben.

Schreiben Sie QuickCheck-Tests für Ihre Implementierung.

Aufgabe H3.1 Einmal auffüllen, bitte!

Wie Sie von der Vorlesungswebseite wissen, wird Haskell unter anderem gerne für Finanzsoftware eingesetzt. Für ein größeres Finanzsoftwareprojekt benötigen Sie zur Anzeige von Geldbeträgen unter anderem eine Funktion, die eine Liste von Zahlen von vorne mit Leerzeichen auffüllt, sodass alle Zahlen die gleiche Länge haben.

```
padNumbers :: [Integer] -> [String]
```

Die Funktion muss dabei folgende Voraussetzungen erfüllen:

- Die Ergebnisliste hat die gleiche Länge wie die Eingabeliste
- Alle Strings in der Ergebnisliste haben die gleiche Länge
- Der i-te Eintrag der Ergebnisliste ist, nach Entfernen des Paddings, die String-Repräsentation des i-ten Eintrags der Eingabeliste.[2] Das Padding darf hierbei nur am Anfang des Strings (bei nichtnegativen Zahlen) bzw. zwischen Vorzeichen und der Zahl selbst (bei negativen Zahlen) stattfinden.

Beispiel: Für `padNumbers [23, -42, 1337]` wären z. B. folgende Ausgaben erlaubt:[3]

```
["␣␣23", "-␣42", "1337"]
["␣␣␣23", "-␣␣42", "␣1337"]
```

Nicht erlaubt wären unter anderem:

```
["␣23", "-␣42", "1337"]
["␣␣23", "␣-42", "1337"]
["+␣23", "-␣42", "1337"]
["␣023", "-␣42", "1337"]
["␣␣23␣", "-␣42␣", "1337␣"]
["-␣42", "␣␣23", "1337"]
["␣␣23", "1337"]
```

Mit einer gewissen Skepsis haben Sie vernommen, dass die Implementierung dieser Funktion einem notorisch unzuverlässigen neuen Mitarbeiter zugeteilt wurde, der bis vor kurzem bei einer jetzt insolventen Finanzfirma gearbeitet hat. Zur Sicherheit wollen Sie daher QuickCheck-Tests schreiben, die sicher stellen, dass der Code keine Fehler enthält.

Stellen Sie dabei, dass Ihr Test *korrekt* und *vollständig* ist. *Korrekt* bedeutet, dass der Test für keine Implementierung, die die obigen Voraussetzungen erfüllt, fehlschlägt. *Vollständig* bedeutet,

[2] Mit „String-Repräsentation“ ist hierbei der String gemeint, den `show` ausgibt – also ohne führende Nullen, mit `'-'` als Vorzeichen für negative Zahlen und ohne Vorzeichen für nichtnegative Zahlen.

[3] Ein `'␣'` steht hierbei für ein Leerzeichen.

dass der Test (theoretisch) jede falsche Implementierung erkennt, also dass es für jede falsche Implementierung Testparameter gibt, sodass der Test fehlschlägt.

Hinweis: Es ist nicht gefordert und auch nicht notwendig für diese Aufgabe, die `padNumbers`-Funktion selbst zu implementieren. Sie müssen *nur* die Tests schreiben. In der Übungsschablone ist hierfür die Property `prop_padNumbers` vorgesehen, mit folgender Definition:

```
prop_padNumbers padNumbers xs =
  undefined :: Bool
```

Eine beispielhafte (aber unnütze) Implementation:

```
prop_padNumbers padNumbers xs =
  0 <= length (padNumbers xs)
```

Sie sehen hier, dass die Funktion `padNumbers` sowohl links als auch rechts vom Gleichheitszeichen steht. Das können Sie getrost ignorieren – verwenden Sie `padNumbers` einfach wie eine gewöhnliche Funktion.

Aufgabe H3.2 Zeilen im Umbruch

Implementieren Sie eine Funktion `wrapText :: Integer -> [String] -> String`. Der Aufruf `wrapText` *n ws* soll die Liste von Wörtern *ws* zu einem Text formatieren. Ein *Wort* besteht dabei aus beliebigen Zeichen, die kein *Leersymbol* sind (vgl. `isSpace :: Char -> Bool` aus `Data.Char`).

Dabei sollen die folgenden Regeln beachtet werden:

- Die Reihenfolge der Wörter wird beibehalten.
- Wörter werden nicht getrennt.
- Zwischen zwei Wörtern steht genau ein Leerzeichen(`` ` ` ``) oder ein Zeilenumbruch (`` `\n` ``).
- Besteht eine Zeile aus zwei oder mehr Wörtern, so besteht sie aus höchstens n Zeichen (ohne den abschließenden Zeilenumbruch).
- Jede Zeile ist so lang wie möglich.
- Entfernt man den abschließenden Zeilenumbruch, so hat eine Zeile keine Leersymbole am Anfang oder Ende.

Aufgabe H3.3 Stretch-Bilder

Wie aus der Vorlesung bekannt, werden Bilder (Typ `Picture`) durch Listen von Strings dargestellt. In der Übungsschablone finden Sie die Definition eines Beispielbildes

```
pic = [".##.", ".#.#", ".###", "####"]
```

sowie der Funktion `printPicture :: Picture -> IO ()`, die Sie als Black-Box für die zweidimensionale Ausgabe von Bildern **im Interpreter** nutzen können. Zum Beispiel:

```
> printPicture pic
.##.
.#.#
.###
####
```

Sie sollen eine Funktion `stretch :: Picture -> Picture` definieren, die ein Bild auf das zweifache vergrößert. Zum Beispiel:

```
> printPicture (stretch pic)
..####..
..####..
..##..##
..##..##
..######
..######
########
########
```

Wie im Beispiel zu sehen ist, sollen die existierenden Pixel vervierfacht werden.

Sie dürfen annehmen, dass die Eingabebilder „rechteckig“ sind, d.h. jede Zeile hat die gleiche Länge. Die Ausgabe muss ebenso rechteckig sein.

Aufgabe H3.4 Grüppchenbildung (Wettbewerbsaufgabe)

Der MC hat diese Woche schon wieder Probleme bei der Punkteberechnung. Wie letzte Woche bereits erwähnt wurde, speichert er die Punkte für den Wettbewerb als Assoziationslisten vom Typ `[(String, Integer)]`. Beim Berechnen der Gesamtpunktzahl aus den Punkten für die einzelnen Blätter hat er nun eine Liste der Form

```
[("A",23), ("A",42), ("B",13), ("C",36), ("C",32), ("C",20)]
```

Er will nun jeweils alle adjazenten Elemente mit gleichem Schlüssel zusammenfügen und dabei die Punkte addieren, also:

```
[("A",65), ("B",13), ("C",88)]
```

Schreiben Sie hierfür eine Funktion

```
coalesce :: [(String, Integer)] -> [(String, Integer)]
```

die dies leistet. Beachten Sie, dass *nur adjazente* Elemente zusammengefügt werden sollen; eine Liste wie `[("A",1), ("B",2), ("A",3)]` würde von `coalesce` nicht verändert werden.

Für den Wettbewerb zählt die Tokenanzahl (je kleiner, desto besser; siehe Aufgabenblatt 1). Lösungen, die gleichwertig bezüglich der Tokenanzahl sind, werden im Hinblick auf ihre Effizienz

verglichen. Die vollständige Lösung (außer `import`-Direktiven) muss innerhalb von Kommentaren {`-WETT-`} und {`-TTEW-`} stehen.

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.