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

WS 2012/13
30. 10. 2012
Abgabe: 6. 11. 2012, 15:30

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 der Bibliothek `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 `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 `(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 der `List`-Bibliothek 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

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:

$$\begin{bmatrix} (a_1,a_2), & (a_1,a_3), & (a_1,a_4), & \ldots & (a_1,a_{n-1}), & (a_1,a_n), \\ & (a_2,a_3), & (a_2,a_4), & \ldots & (a_2,a_{n-1}), & (a_2,a_n), \\ & & (a_3,a_4), & \ldots & (a_3,a_{n-1}), & (a_3,a_n), \\ & & & \ddots & \vdots & \vdots \\ & & & & (a_{n-2},a_{n-1}), & (a_{n-2},a_n), \\ & & & & & (a_{n-1},a_n) \end{bmatrix}$$

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 Leersymbole normalisieren (4 Punkte)

Wir definieren den Begriff eines Leersymbols.

Leersymbol Ein Zeichen, für das die Funktion `isSpace` (aus der `Data.Char`-Bibliothek) den Wert `True` zurück gibt. Unter anderem Leerzeichen (`' '`) und Zeilenumbruch (`'\n'`).

Schreiben Sie eine Funktion `simplifySpaces :: [Char] -> [Char]`, die in der Eingabe aufeinanderfolgende Leersymbole durch ein einziges Leerzeichen ersetzt, sowie führende und nachgestellte Leersymbole entfernt. Alle verbleibenden Leersymbole sollen durch Leerzeichen ersetzt werden.

Beispiel (␣ ist ein Leerzeichen, `\n` ein Zeilenumbruch und `\t` ein Tabulator):

```
simplifySpaces
  "␣␣\nA\t␣quick␣␣␣brown␣\n\tfox\njumps␣over␣the\rlazy␣dog\r\n" ==
  "A␣quick␣brown␣fox␣jumps␣over␣the␣lazy␣dog"
```

Aufgabe H3.2 Zeilenumbrüche (5 Punkte)

Gegeben sei eine Funktion `wrap :: [Char] -> [Char]`, die eine Zeichenkette nimmt und diesen String möglichst so umbricht, dass keine Zeile länger als 40 Zeichen ist. Die Ausgabe ist die umformatierte Zeichenkette. Wir definieren Wörter und Zeilen. Für die Definition von Leersymbolen vergleiche H3.1.

Wort Eine nicht-leere Zeichenkette, die keine Leersymbole enthält. Die Wörter einer Zeichenkette sind durch Leersymbole getrennt und können mit `words :: [Char] -> [[Char]]` (aus `Prelude`) berechnet werden.

Zeile Eine nicht-leere Zeichenkette, die keinen Zeilenumbruch (`'\n'`) enthält. Die Zeilen einer Zeichenkette sind durch Zeilenumbrüche getrennt und können mit der Haskell-Funktion `lines :: [Char] -> [[Char]]` (aus `Prelude`) berechnet werden.

Beispiele: `"Tag"` ist ein Wort, `"a b "` und `""` nicht. Die Wörter von `"Tag"` sind `["Tag"]`, die von `"a b "` sind `["a","b"]` und die von `""` sind `[]`. Zeilen sind analog zu Wörtern definiert.

Ihre Aufgabe ist es jetzt, QuickCheck-Tests zu schreiben, die kontrollieren, dass die Ausgabe der `wrap`-Funktion die folgenden Regeln erfüllt:

1. Die Liste der Wörter von Eingabe und Ausgabe sind gleich.
2. Als Leersymbole sind nur Leerzeichen (`' '`) und Zeilenumbruch (`'\n'`) erlaubt.
3. Leersymbole am Anfang und am Ende der Zeichenkette sind nicht erlaubt.
4. Mehrere aufeinanderfolgende Leersymbole sind nicht erlaubt.

5. Eine Zeile enthält mindestens ein Wort. Besteht sie aus mehreren Wörtern, so ist sie höchstens 40 Zeichen lang.

Ihre Tests sollen das gewünschte Verhalten der Funktion `wrap` *vollständig* spezifizieren; das heißt, wenn `wrap` nicht korrekt implementiert wurde, muss es eine Eingabe für einen Ihrer Tests geben, so dass der Test fehlschlägt. (Ob diese Eingabe von QuickChecks Zufallsstrategie tatsächlich gefunden wird, sei dahingestellt.)

Sie können bis zu zehn Tests `prop_wrap1`, ..., `prop_wrap10` schreiben, es reichen aber deutlich weniger aus. Jede der Testfunktionen muss die folgende Form haben:

```
prop_wrapN :: WrapFun -> [Char] -> Bool
prop_wrapN wrap ... = ...
```

Zum Beispiel:

```
prop_wrap3 :: WrapFun -> [Char] -> Bool
prop_wrap3 wrap xs = wrap xs == wrap (xs + "  ")
```

Sie sehen hier, dass die Funktion `wrap` sowohl links als auch rechts vom Gleichheitszeichen steht. Ignorieren Sie das vorerst. Der Typ `WrapFun` ist in der offiziellen Übungsschablone[2] durch

```
type WrapFun = [Char] -> [Char]
```

definiert. Sie können Ihre `prop_wrap`-Funktionen testen, indem Sie eine Funktion mit der richtigen Typsignatur als erstes Argument angeben. Zum Beispiel:

```
prop_wrap3 simplifySpaces "Hallo\nWelt!\n"
```

(Hier ist `simplifySpaces` nur ein Beispiel. Vollständige Tests sollen offenbaren, dass sie keine korrekte `wrap`-Funktion ist, obwohl sie die Regeln 1 bis 4 erfüllt.)

Aufgabe H3.3 Rotation im Uhrzeigersinn (5 Punkte)

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:

```
Exercise_3> printPicture pic
 ## 
 # # 
 ### 
####
```

[2] `http://www21.in.tum.de/teaching/info2/WS1213/blaetter/Exercise_3.hs`

Sie sollen eine Funktion `rotateClockwise :: Picture -> Picture` definieren, die ein Bild um 90° im Uhrzeigersinn dreht. Zum Beispiel:

```
Exercise_3> printPicture (rotateClockwise pic)
#␣␣␣
####
##␣#
###␣
```

Beachten Sie dass die Eingabebilder nicht unbedingt „rechteckig“ sein müssen (die Längen der inneren Listen bzw. Zeilen dürfen unterschiedlich sein). Dabei kann es passieren, dass Sie nach der Rotation manche Zeilen mit dem Leerzeichen `'␣'` auffüllen müssen, wie die Beispiele a_1, b_1, c_1 und a_2, b_2, c_2 zeigen. a_i ist dabei jeweils das Eingabebild angezeigt mittels `printPicture` (die Eingabe im Beispiel a_1 ist `["##","␣"]` und im Beispiel a_2 `["␣##","␣#","#"]`). b_i ist das erwartete Ergebnis nach der Rotation, welches jedoch nicht mit unserer Representation der Bilder kompatibel ist. Die Funktion `rotateClockwise` soll stattdessen die in c_i dargestellten Bilder zurückgeben.

a_1)
```
##
␣
```
b_1)
```
␣#
 #
```
c_1)
```
␣#
␣#
```

a_2)
```
␣##
␣#
#
```
b_2)
```
#␣␣
##
 #
```
c_2)
```
#␣␣
␣##
␣␣#
```

Aufgabe H3.4 Teillisten und Teilsequenzen (6 Punkte, Wettbewerbsaufgabe)

Der Master of Competition sucht diese Woche zwei Funktionen:

```
sublist :: Eq a => [a] -> [a] -> Bool   -- Teilliste
subseq :: Eq a => [a] -> [a] -> Bool    -- Teilsequenz
```

- Der Aufruf `sublist` xs ys soll genau dann `True` zurückgeben, wenn ys von der Form as ++ xs ++ bs für gewisse as und bs ist. Das heißt, die Liste xs muss irgendwo in ys vorkommen (wie z.B. „rko“ im Wort „vorkommen“).
- Der Aufruf `subseq` $[x_1, x_2, \ldots, x_n]$ ys soll genau dann `True` zurückgeben, wenn ys sich in die Form as_0 ++ $[x_1]$ ++ as_1 ++ $[x_2]$ ++ as_2 ++ $\cdots$ ++ as_{n-1} ++ $[x_n]$ ++ as_n für gewisse $as_0, \ldots, as_n$ bringen lässt. Das heißt, alle Elemente in xs müssen in ys vorkommen und in der gleichen Reihenfolge, wobei ys zusätzlich andere Elemente beinhalten kann, auch mittendrin (z.B. „ttdr“ im Wort „mittendrin“).

Hier sind Beispiele, die alle `True` sind (`w` ist ein beliebiges Wort):

```
sublist "" w
sublist w w
sublist "ab" "cab"
not (sublist "cb" "cab")
```

```
subseq "" w
subseq w w
subseq "ab" "cab"
subseq "cb" "cab"
```

```
not (sublist "aa" "xaxa")       subseq "aa" "xaxa"
not (sublist "aa" "xax")        not (subseq "aa" "xax")
not (sublist "ab" "ba")         not (subseq "ab" "ba")
```

Standardbibliothekfunktionen auf Listen sind erlaubt, `List.isInfixOf` aber nicht!

Für den Wettbewerb zählt die Tokenanzahl (je kleiner, desto besser[3]). Um teilzunehmen müssen Sie die vollständige Lösung (außer `import`-Direktiven) innerhalb von Kommentaren `{-WETT-}` und `{-TTEW-}` eingeben. Zum Beispiel:

```
import Data.List

{-WETT-}
sublist :: Eq a => [a] -> [a] -> Bool
sublist xs ys = ...

subseq :: Eq a => [a] -> [a] -> Bool
subseq xs ys = ...
{-TTEW-}
```

Für seine Lösung hat der MC 48 Tokens gebraucht (exklusive Typsignaturen). Versuchen Sie, ihn zu schlagen!

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.

[3] `http://www21.in.tum.de/teaching/info2/WS1213/wettbewerb.html`