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

WS 2012/13
23. 10. 2012
Abgabe: 30. 10. 2012, 15:30

Einführung in die Informatik 2
2. Übung

Aufgabe G2.1 Listenkomprehensionen

Benutzen Sie Listenkomprehensionen um die folgenden Funktionen zu implementieren:

1. Schreiben Sie eine Funktion `all_sums :: [Integer] -> [Integer] -> [Integer]`. Das Ergebnis von `all_sums xs ys` soll eine Liste sein, die für jedes x in `xs` und jedes y in `ys` den Wert $x + y$ enthält.
2. Schreiben Sie eine Funktion `evens :: [Integer] -> [Integer]`, die alle ungerade Zahlen aus einer Liste entfernt. Sie können die Funktion `mod` verwenden.
3. Schreiben Sie eine Funktion `n_lists :: [Integer] -> [[Integer]]`, die eine Liste von Zahlen so auf eine Liste von Listen abbildet, dass n auf die Liste von 1 bis n abgebildet wird.
4. Schreiben Sie mit nur einer Listenkomprehension die Funktion

   ```
   all_even_sum_lists :: [Integer] -> [Integer] -> [[Integer]]
   ```

   die für Eingaben `xs` und `ys` die Ausgabe `n_lists (evens (all_sums xs ys))` berechnet.

Aufgabe G2.2 Mengenoperationen

Wir verwenden Listen um Mengen darzustellen. Schreiben Sie Funktionen für die üblichen Mengenoperationen:

1. Vereinigung: `union :: [Integer] -> [Integer] -> [Integer]`
2. Schnitt: `intersection :: [Integer] -> [Integer] -> [Integer]`
3. Differenz: `diff :: [Integer] -> [Integer] -> [Integer]`.

   Sie können die Funktion `elem :: Integer -> [Integer] -> Bool` verwenden, die bestimmt ob ein Wert Element einer Liste ist.
4. Wie lässt sich `elem` mit Listenkomprehensionen schreiben?
5. Modifizieren Sie gegebenenfalls Ihre Funktionen so, dass keine Duplikate erzeugt werden (d.h., wenn beide Eingabelisten duplikatfrei sind, so ist auch die Ausgabeliste duplikatfrei).

Aufgabe G2.3 Brüche

Wir können Brüche als Tupel darstellen, d.h. das Tupel `(a,b)` repräsentiert den Bruch a/b. Dabei sollen zwei Tupel als gleich behandelt werden, wenn sie den gleichen Bruch repräsentieren, d.h. `(1,2)` und `(3,6)` repräsentieren den gleichen Wert.

1. Schreiben Sie eine Funktion

   ```
   eq_frac :: (Integer,Integer) -> (Integer,Integer) -> Bool
   ```

   die entscheidet, ob zwei Brüche gleich sind.

2. Implementieren Sie die Funktion

   ```
   intersection_frac :: [(Integer,Integer)]
       -> [(Integer,Integer)] -> [(Integer,Integer)]
   ```

   die den Schnitt zweier Mengen von Brüchen berechnet.

Aufgabe G2.4 Potenzen von 2

Die Funktion

```
pow2 :: Integer -> Integer
pow2 0 = 1
pow2 n | n > 0 = 2 * pow2 (n - 1)
```

implementiert $n \mapsto 2^n$ für $n \geq 0$. Für einen gewissen n benötigt die Berechnung n Schritte:

$$2^{100} = 2 \cdot 2^{99} = 2 \cdot 2 \cdot 2^{98} = 2 \cdot 2 \cdot 2 \cdot 2^{97} = \cdots = \overbrace{2 \cdot 2 \cdot \ldots \cdot 2}^{100 \text{ mal}} \cdot 1$$

Gesucht ist eine effizientere Implementierung, die maximal $\lceil 2 \log_2 n \rceil$ Schritte benötigt. Die Gleichungen $2^{2n} = (2^n)^2$ und $2^{2n+1} = 2 \cdot 2^{2n}$ können dabei hilfreich sein. Zum Beispiel:

$$\begin{aligned} 2^{100} &= (2^{50})^2 = ((2^{25})^2)^2 = ((2 \cdot 2^{24})^2)^2 = ((2 \cdot (2^{12})^2)^2)^2 = ((2 \cdot ((2^6)^2)^2)^2)^2 \\ &= ((2 \cdot (((2^3)^2)^2)^2)^2)^2 = ((2 \cdot (((2 \cdot 2^2)^2)^2)^2)^2)^2 \end{aligned}$$

Aufgabe G2.5 n-schrittige Erreichbarkeit in Graphen

Wir modellieren einen (gerichteten) Graphen G als eine Liste von Kanten. Dabei werden Kanten als Paare von Knoten und Knoten als `Integer`-Werte dargestellt. Die Knotenmenge ist nur implizit durch die in der Kantenmenge verwendete Knoten gegeben.

Gesucht ist eine Funktion `reachable :: [(Integer, Integer)] -> Integer -> [(Integer, Integer)]`, die alle Paare von Knoten (v_0, v_n) berechnet, so dass v_n von v_0 aus über n Kanten zu erreichen ist (d.h. es gibt Knoten $v_1, \ldots, v_{n-1}$ und Kanten (v_i, v_{i+1}) für alle $i = 0, \ldots, n-1$ in G). Zum Beispiel:

```
reachable [(1,2), (2,3), (2,5), (4,5), (5,4)] 0 ==
    [(1,1),(2,2),(3,3),(5,5),(4,4)]
reachable [(1,2), (2,3), (2,5), (4,5), (5,4)] 1 ==
    [(1,2),(2,3),(2,5),(4,5),(5,4)]
```

```
reachable [(1,2), (2,3), (2,5), (4,5), (5,4)] 2 ==
    [(1,3),(1,5),(2,4),(4,4),(5,5)]
reachable [(1,2), (2,3), (2,5), (4,5), (5,4)] 3 ==
    [(1,4),(2,5),(4,5),(5,4)]
reachable [(1,2), (2,3), (2,5), (4,5), (5,4)] 4 ==
    [(1,5),(2,4),(4,4),(5,5)]
reachable [(1,2), (2,3), (2,5), (4,5), (5,4)] 5 ==
    [(1,4),(2,5),(4,5),(5,4)]
```

Aufgabe H2.1 Filtern und Zählen (6 + 1 Punkte)

Implementieren Sie folgende Funktionen.

1. Eine Funktion `factorials :: [Integer] -> [Integer]`, die $[k_1, \ldots, k_n] \mapsto [k_1!, \ldots, k_n!]$ rechnet, wobei ! die Fakultät darstellt. Zum Beispiel:

   ```
   factorials [5] == [120]
   factorials [2, 4, 0, 3, 3] == [2, 24, 1, 6, 6]
   ```

   Die Funktion soll zusätzlich robust sein und negative Elemente ignorieren:

   ```
   factorials [-1, 5, -2, 3, 0] == [120, 6, 1]
   ```

   Schreiben Sie weiterhin mindestens zwei QuickCheck-Tests, die interessante Eigenschaften von `factorials` testen. Es gibt einen Bonuspunkt, wenn diese Eigenschaften die Funktion vollständig beschreiben.

2. Eine Funktion `count :: [Char] -> Char -> Integer`, die zählt, wie oft ein Buchstabe in einem Wort vorkommt. Zum Beispiel:

   ```
   count "Abrakadabra" 'a' == 4
   count "Abrakadabra" 'x' == 0
   ```

   Sie dürfen hierbei Basisfunktionen der `List`-Standardbibliothek verwenden, sowie `sum` und `genericLength`.[1]

Aufgabe H2.2 Tabellenlookup (4 Punkte)

Wir stellen eine Tabelle als Liste von Schlüssel-Wert Paaren dar. Dabei hat ein Schlüssel den Typ `Integer` und ein Wert den Typ `[Char]`. Eine Tabelle darf zum gleichen Schlüssel unterschiedliche Werte assoziieren.

Es soll nun eine Funktion `lookupTab :: [Integer] -> [(Integer, [Char])] -> [[[Char]]]` definiert werden, die als Parameter eine Liste von Schlüsseln und eine Tabelle erhält. Für jeden Schlüssel in der Eingabeliste soll die Liste der in der Tabelle assoziierten Werte in der Ausgabeliste stehen. Zum Beispiel:

[1] Ärgerlicherweise liefert die in den Folien erwähnte `length`-Funktion einen 32-Bit-`Int` und keinen unbegrentzten `Integer` zurück.

```
lookupTab [1,5,-3,2,2]
  [(1, "Hallo"),
   (-3, "I"),
   (1, "Welt"),
   (4, "42"),
   (-3, "love"),
   (-3, "Haskell")] ==
  [["Hallo","Welt"], [], ["I","love","Haskell"],[],[]]
```

Verwenden Sie lediglich Listenkomprehensionen, um über die Listen zu iterieren.

Aufgabe H2.3 Strings der Länge n (5 Punkte)

Gesucht ist eine Funktion `wordsOfLength :: [Char] -> Integer -> [[Char]]`, die als Parameter eine Liste von Buchstaben (das Alphabet) und eine nicht-negative Zahl n erhält und alle Strings der Länge n, die aus Buchstaben des Alphabets bestehen, in einer Liste zurückgibt. Ihre Ausgabeliste soll keine Duplikate enthalten. Dabei dürfen Sie davon ausgehen, dass auch das Alphabet keine Duplikate enthält. Zum Beispiel:

```
wordsOfLength "abc" 2 ==
    ["aa","ab","ac","ba","bb","bc","ca","cb","cc"]
wordsOfLength "ab" 3 ==
    ["aaa","aab","aba","abb","baa","bab","bba","bbb"]
wordsOfLength "ab" 4 ==
    ["aaaa","aaab","aaba","aabb","abaa","abab","abba","abbb",
     "baaa","baab","baba","babb","bbaa","bbab","bbba","bbbb"]
```

Aufgabe H2.4 Permutationen (5 Punkte, Wettbewerbsaufgabe)

Gesucht ist eine Funktion `perms :: [Char] -> [[Char]]`, die alle Permutationen eines Wortes in umgedrehter alphabetischen (oder "ASCII-betischen") Reihenfolge zurückliefert. Die Liste der Permutationen soll keine Duplikate enthalten. Zum Beispiel:

```
perms "" == [""]
perms "a" == ["a"]
perms "ab" == ["ba","ab"]
perms "ba" == ["ba","ab"]
perms "aba" == ["baa","aba","aab"]
perms "abc" == ["cba","cab","bca","bac","acb","abc"]
perms "yhwh" ==
    ["ywhh","yhwh","yhhw","wyhh","whyh","whhy",
     "hywh","hyhw","hwyh","hwhy","hhyw","hhwy"]
perms "xxxxxxxxxxxxxxx" == ["xxxxxxxxxxxxxxx"]
```

Sie dürfen dafür die folgenden Funktionen aus der `List` Standardbibliothek benutzen, wo `a` einen beliebigen Typ darstellt (z.B. `Integer`, `Char` oder `[Char]`):

```
delete :: a -> [a] -> [a]   -- delete 3 [3,1,3] == [1,3]
nub :: [a] -> [a]           -- nub [1,3,1,2,2] == [1,3,2]
reverse :: [a] -> [a]       -- reverse [1,3,2] == [2,3,1]
sort :: [a] -> [a]          -- sort [1,3,2] == [1,2,3]
```

Basisfunktionen wie `++` sind erlaubt, `List.permutations` aber nicht! Für den Wettbewerb zählt die Tokenanzahl (je kleiner, desto besser; siehe Aufgabenblatt 1). Lösungen, die gleichwertig bzgl. der Tokenanzahl sind, werden bzgl. ihrer Effizienz verglichen. Die vollständige Lösung (außer `import`-Direktiven) muss innerhalb von Kommentaren {`-WETT-`} und {`-TTEW-`} stehen. Zum Beispiel:

```
import Data.List

{-WETT-}
perms :: [Char] -> [[Char]]
perms [] = ...
perms cs = ...
{-TTEW-}
```