Mehr über Haskell, Teil 3

...Nachträge und Spezialitäten über Listen

Stichwörter:

- Listenkomprehension
- (Polymorphe) Funktionen auf Listen
- Vordefinierte Funktionen auf Listen
- Polymorphie vs. Überladen
- Listen und Muster

Listenkomprehension (1)

...ein für funktionale Prog.sprachen charakteristisches Ausdrucksmittel.

Anwendungsbeispiele: ...wobei `lst = [1,2,4,7,8,11,12,42]`

a) `[ square n | n <- lst ]` $\Rightarrow^*$ `[1,4,16,49,64,121,144,1764]`

b) `[ n | n <- lst, isPowOfTwo n ]` $\Rightarrow^*$ `[1,2,4,8]`

c) `[ n | n <- lst, isPowOfTwo n, n>=5 ]` $\Rightarrow^*$ `[8]`

d) `[ isPrime n | n <- lst ]` $\Rightarrow^*$
`[False,True,False,True,False,True,False,False]`

Listenkomprehension (2)

Weitere Beispiele...

e)
```
addCoordinates :: [Point] -> [Float]
addCoordinates pLst = [ x+y | (x,y)<-pLst, (x>0||y>0) ]

addCoordinates [(0.0,0.5),(3.14,17.4),(-1.5,-2.3)] ⇒*
                                            [0.5,20.54]
```

f)
```
allOdd :: [Integer] -> Bool
allOdd xs = ( [ x | x<-xs, isOdd x ] == xs )

allEven :: [Integer] -> Bool
allEven xs = ( [ x | x<-xs, isOdd x ] == [] )
```


Listenkomprehension (3)

g)
```
grabCapVowels :: String -> String
grabCapVowels s = [ c | c<-s, isCapVowel c ]

isCapVowel :: Char -> Bool
isCaPVowel 'A' = True
isCapVowel 'E' = True
isCapVowel 'I' = True
isCapVowel 'O' = True
isCapVowel 'U' = True
isCapVowel  c  = False
```

Listenkomprehension "at work" (1)

...am Beispiel von "Quicksort".

Aufgabe: Sortiere eine Liste L ganzer Zahlen aufsteigend.

Lösung mittels Quicksort:

- *Teile*: Wähle ein Element l aus L und partitioniere L in zwei (möglicherweise leere) Teillisten L_1 und L_2 so, dass alle Elemente von L_1 (L_2) kleiner oder gleich (größer) dem Element l sind.
- *Herrsche*: Sortiere L_1 und L_2 mit Hilfe rekursiver Aufrufe von Quicksort.
- *Zusammenführen der Teilergebnisse*: Trivial, die Gesamtliste entsteht durch Konkatenation der sortierten Teillisten.

Listenkomprehension "at work" (2)

Quicksort: Eine typische imperative Realisierung...

```
quickSort (L,low,high)
  if low < high
    then splitInd = partition(L,low,high)
         quickSort(L,low,splitInd-1)
         quickSort(L,splitInd+1,high) fi

partition (L,low,high)
  l = L[low]
  left = low
  for i=low+1 to high do
    if L[i] <= l then left = left+1
                      swap(L[i],L[left]) fi od
  swap(L[low],L[left])
  return left
```

...und dem initialen Aufruf `quickSort(L,1,length(L))`.

Listenkomprehension "at work" (3)

QuickSort: Eine typische Realisierung in Haskell...

```
quickSort :: [Int] -> [Int]

quickSort []     = []
quickSort (x:xs) =
    quickSort [ y | y<-xs, y<=x ] ++
                  [x] ++ quickSort [ y | y<-xs, y>x ]
```

Beachte: Funktionsanwendung bindet stärker als Listenkonstruktion. Deshalb Klammerung des Musters `x:xs` in `quickSort (x:xs) = ...`

Rückblick auf Teil 1 der Vorlesung...

Ein Vergleich - imperative vs. funktionale Programmierung

Gegeben ein Problem P.

Imperativ: Typischer Lösungsablauf besteht aus folgenden Schritten

1. Beschreibe eine(n) Lösung(sweg) L für P
2. Gieße L in die Form einer Menge von Anweisungen für den Rechner und organisiere dabei die Speicherverwaltung

Funktional:

- ...das "was" statt des "wie" in den Vordergrund stellen
- $\leadsto$ etwas von der Eleganz der Mathematik in die Programmierung bringen!

Quicksort: Ein eindrucksvolles Beispiel? Urteilen Sie selbst...

Generische Funktionen, Polymorphie, Typvariablen (1)

- Listen von Basistypelementen
  `[2,4,23,2,53,4] :: [Int]`
- Listen von Listen
  `[[2,4,23,2,5],[3,4],[],[56,7,6,]] :: [ [Int] ]`
- Listen von Paaren
  `[(3.14,42.0),(56.1,51.3),(1.12,2.22)] :: [ Point ]`
- Listen von Funktionen
  `[fact, fib, fibFix] :: [ Integer -> Integer ]`
- ...

Generische Funktionen, Polymorphie, Typvariablen (2)

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Sprechweisen:

- `a` in `length :: [a] -> Int` heißt *Typvariable*. Typvariablen werden gewöhnlich mit Kleinbuchstaben vom Anfang des Alphabets bezeichnet: `a`, `b`, `c`,...
- Typen wie...

  ```
  length :: [Bool] -> Int
  length :: [ [Int] ] -> Int
  length :: [ Integer -> Integer ] -> Int
  ...
  ```

  heißen *Instanzen* des Typs `[a] -> Int`. Letzterer heißt *allgemeinster Typ* der Funktion `length`.

Weitere Beispiele polymorpher Funktionen

```
id :: a -> a
id x = x

ignoreSecond :: Num a => (Bool,b) -> a
ignoreSecond (x,y) = if x then 1 else -1

zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _           = []

unzip :: [(a,b)] -> ([a],[b])
unzip []  = ([],[])
unzip ((x,y):ps) = (x:xs,y:ys)
                   where
                   (xs,ys) = unzip ps
```


In Haskel vordefinierte (polymorphe) Funktionen auf Listen...

`:`	`a->[a]->[a]`	Listenkonstruktor (rechtsassoziativ)
`!!`	`[a]->Int->a`	Proj. auf i-te Komp., Postfixop.
`length`	`[a]->Int`	Länge der Liste
`++`	`[a]->[a]->[a]`	Konkatenation zweier Listen
`concat`	`[[a]]->[a]`	Konkatenation mehrerer Listen
`head`	`[a]->a`	Listenkopf
`last`	`[a]->a`	Listen "schlusselement"
`tail`	`[a]->[a]`	Liste ohne Listenkopf
`init`	`[a]->[a]`	Liste ohne Listenschlusselement
`splitAt`	`Int->[a]->[[a],[a]]`	Aufspalten einer Liste an Stelle i
`reverse`	`[a]->[a]`	Umdrehen einer Liste
...		

Randbemerkungen... (1)

Listenkonstruktor vs. *Listenoperator*

: ist Listenkonstruktor, ++ ist Listenoperator...

Abgrenzung: Konstruktoren führen zu eindeutigen Darstellungen, Nichtkonstruktoren i.a. nicht.

Veranschaulichung:

`42:17:4:[] == (42:(17:(4:[])))`, aber...

`[42,17] ++ [] ++ [4] == [42,17,4] == [42] ++ [17,4] ++ []`

Bemerkung: `(42:(17:(4:[])))` deutet an, dass eine Liste *ein* Objekt ist, erzwungen durch die Typstruktur. Anders in imperativen/objektorientierten Sprachen: Listen nur indirekt existent, nämlich bei "geeigneter" Verbindung von Elementen durch Zeiger.

Randbemerkungen... (2)

"Verstecken" von in `Prelude.hs` vordefinierten Funktionen...

Ergänze...

```
import Prelude hiding (reverse,tail,zip)
```

...am Anfang des Haskell-Skripts im Anschluss an die Modul-Anweisung (so vorhanden), um `reverse`, `tail` und `zip` zu verbergen.

(vgl. Aufgabenblatt 2 vom 04.11.2003: `reverse`)

Polymorphie vs. Überladen

- Polymorphie ("Vielgestaltigkeit)
- Überladen (engl. Overloading) oder ad-hoc Polymorphie

Vordergründig...

...ein Funktionsname kann auf Argumente unterschiedlichen Typs angewendet werden.

Bsp.:

```
Polymorphie                                 Ueberladen
-----------                                 -----------
length :: [a] -> Int                        gleich :: a -> a -> Bool
length []     = 0                  "naiv":  gleich x y = x == y
length (_:xs) = 1 + length xs
```

Beachte: Anders als für `length` ist für `gleich` keine typunabhängige einheitliche Implementierung möglich (denke an Gleichheit auf Basistypen, Paaren, Listen, Bäumen, etc.)

Listen, Muster und Funktionen

Weitere Beispiele...

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs

head :: [a] -> a
head (x:_)  =  x

tail :: [a] -> [a]
tail (_:xs) = xs

null :: [a] -> Bool
null []    = True
null (_:_) = False
```

Fortsetzung: Muster können sein...

Bislang:

- *Werte* (z.B. 0, 'c', True)
  ...ein Argument "passt" auf das Muster, wenn es vom entsprechenden Wert ist.
- *Variablen* (z.B. n)
  ...jedes Argument passt.
- *Wild card* "_"
  ...jedes Argument passt.

Neu:

- *Konstruktormuster* (hier über Listen; z.B. [], (p:ps))
  - Eine Liste passt auf das Muster [], wenn sie leer ist.
  - Eine Liste passt auf (p:ps), wenn sie nicht leer ist und der Listenkopf auf p und der Listenrest auf ps passt.

Hinweis: Im Fall von (p:ps) reicht es, dass die Argumentliste nicht leer ist.

Beachte...

Musterbasierte Funktionsdefinitionen sind (i.a.)...

- elegant und
- führen zu knappen, gut lesbaren Spezifikationen.

Aber: Musterbasierte Funktionsdefinitionen...

- können zu subtilen Fehlern führen und
- erschweren (oft) Programmänderungen/-weiterentwicklungen (bis hin zur Tortur)