Sven Eric Panitz

Lehrbriefe Funktionale Programmierung

2 Listen

Listen

Sven Eric Panitz

24. März 2025

Inhaltsverzeichnis

1 Listen

Mehrere Dinge gleicher Art zusammenzufassen, ist eine zentrale Aufgabe in der Programmierung. Imperative Programmiersprache bieten hierfür zunächst Arrays an, die einen Abschnitt im Hauptspeicher repräsentieren. Arrays haben zunächst den Nachteil, dass sie nicht dynamisch wachsen können. Hierzu werden dann meist Datenstrukturen entwickelt, die auf Arrays basieren und zur Not den Arrays in einen größeren Array kopieren.

In funktionalen Sprachen ist das Mittel der Wahl eine einfach verkettete Liste. Dieses war für die Programmiersprache Lisp so zentral, dass dort nicht nur die Daten sondern auch die Programme selbst in der Listenstruktur notiert waren.

Auch in Haskell spielen Listen eine zentrale Rolle. Sie haben teilweise eine eigene Syntax und umfangreiche Funktionen zur Arbeit mit Listen sind bereits im Prelude definiert.

In dieser Aufgabe soll das Arbeiten mit Listen eingeübt werden.

Listen.lhs

```
> module Listen where
```

Um auf alle wichtigen Funktionen zuzugreifen, die für Listen implementiert sind, importieren wir das Standardmodul zu Listen:

Listen.lhs

```
> import Data.List
```

Wir werden auch Aufgaben haben, die mit rationalen Zahlen arbeiten, und importieren auch das entsprechende Modul.

Listen.lhs

```
> import Data.Ratio
```

1.1 Listenkonstruktoren und Listenliterale

Betrachten wir zunächst, wie Listen direkt im Quelltext durch ihre Elemente erzeugt werden können.

1.1.1 Konstruktoren

Streng genommen sind die Standardlisten in Haskell ein algebraischer Datentyp, der polymorph (generisch) über den Elementtypen ist und zwei Konstruktoren hat. Einen Konstruktor für die leere Liste und einen Konstruktor, um aus einer tail-Liste und einem head-Element eine neue Liste zu konstruieren. Der zweite Konstruktor ist ein Infixkonstruktor.

Vergessen wir einmal kurz, dass es bereits Standardlisten gibt und definieren einen eigenen Listendatentyp:

Listen.lhs

```
> data Li a = E | a :> Li a deriving (Show,Eq)
```

Operatoren, die mit einem Doppelpunkt beginnen sind in Haskell Infixkonstruktoren. Bezeichner, die mit einem Großbuchstaben beginnen, sind Konstruktoren.

Der Typ `Li` ist polymorph.[1] Der Typ der Elemente der Liste ist variabel gehalten. Hierfür haben wir die Typvariable `a` verwendet.

Es wurden zwei Konstruktoren definiert. der Konstruktor `E` für eine leere Liste und der Infixkonstruktor, der ein Kopfelement vorn an eine Restliste hinzufügt.

Den Infixkonstruktor `:>` definieren wir als rechtsassoziativ.

Listen.lhs

```
> infixr 5 :>
```

Jetzt können mit den beiden Konstruktoren Listen unserer eigenen Listenimplementierung erzeugen. Die Liste aus den Zahlen 1 bis 5 lässt sich schreiben als:

[1] In der objektorientierten Programmierung würde man sagen, er ist generisch.

```
1:>2:>3:>4:>5:>E
```

```
1 :> (2 :> (3 :> (4 :> (5 :> E))))
```

Die Standardlisten in Haskell unterscheiden sich kaum von unserem Datentyp `Li`.

Unser Infixkonstruktor `:>` ist in den Standardlisten der Infixkonstruktor `:`. Unser Konstruktor `E` zum Konstruieren einer leeren Liste ist bei den Standardlisten ein leeres eckiges Klammerpaar: `[]`.

Auch wenn die Standardlisten mehr oder weniger ein ganz normaler algebraischer Datentyp sind, so sind sie doch in einigen Aspekten gleicher als gleich, zu anderen strukturierten Datentypen.

Eine der Besonderheiten der Standardlisten sind die eckige Klammern. Einen Konstruktor, der aus Klammerpaaren gebildet wird, kann sonst nicht definiert werden.

Ebenso wie unsere Liste mit den beiden Konstruktoren lässt sich auch eine Standardliste erzeugen.

```
1:2:3:4:5:[]
```

```
[1,2,3,4,5]
```

Man sieht lediglich, dass die Funktion zur Anzeige der Standardlisten diese nicht mit den Konstruktoren anzeigt, sondern die Elemente in eckigen Klammern umschlossen aufgelistet.

Eine weitere Besonderheit der Standardlisten ist der Typname. Während unsere Listendatenstruktur den Typ `Li a` definiert, ist der Datentyp der Standardlisten als `[a]` notiert.

Der Typ unserer Liste:

```
:t 1:>2:>3:>4:>5:>E
```

```
1:>2:>3:>4:>5:>E :: Num a => Li a
```

Der Typ der Standardlisten:

```
:t 1:2:3:4:5:[]
```

```
1:2:3:4:5:[] :: Num a => [a]
```

1.1.2 Listliterale

Die Standardlisten haben eine spezielle Syntax, um Listen direkt hinzuschreiben. Hierzu werden die Elemente in eckigen Klammern eingeschlossen

```
[17,4,42]
```

```
[17,4,42]
```

1.1.3 Strings

Es gibt eine besondere Art von Listen, für die auch eine eigene Syntax existiert. Es sind die Listen, deren Elemente vom Typ `Char` sind, also `[Char]`. Für diesen Typ gibt es ein Typsynonym. Sie werden auch als `String` bezeichnet.

```
['h','a','l','l','o']
```

```
"hallo"
```

Sie haben eine eigene Syntax für Literale: statt `['h','a','l','l','o']` lässt sich auch `"hallo"` schreiben.

```
"hallo"
```

```
"hallo"
```

Wir hätten natürlich auch ganz umständlich einen String mit den expliziten Listenkonstruktor definieren können.

```
'h':'a':'l':'l':'o':[]
```

Das ist umständlich, aber im Kopf zu behalten, wenn man in Pattern einen String in einzelne Bestandteile matchen will.

1.2 Listpattern

Für die Fallunterscheidung auf die unterschiedlichen Fälle eines algebraischen Typs bietet sich in Haskell das Pattern Matching als Mittel der Wahl an. Hierbei werden mehrere Funktionsgleichungen aufgestellt, die die unterschiedlichen Fälle für die Argumente als Pattern abbilden. Für unsere eigene Listenimplementierung können wir so zum Beispiel eine Funktion für die Längenberechnung schreiben als:

Listen.lhs

```
> laengeLi E = 0
> laengeLi (_:>xs) = 1 + laengeLi xs
```

Hier gibt es zwei Gleichungen. Die erste ist für Listen, die mit dem Konstruktor `E` für leere Listen erzeugt wurde. Deren Länge ist 0. Die zweite Gleichung nutzt das Pattern des Infixkonstruktors `:>`, der zwei Argumente hat. Sein linkes Argument interessiert nicht für die Längenberechnung, daher wird der Unterstrich als Pattern verwendet. Sein rechtes Argument, die Teilliste ohne das erste Element, wird an die Variable `xs` gebunden.

Die entsprechende Funktion für die eingebauten Standardlisten, lässt sich definieren als:

Listen.lhs

```
> laenge [] = 0
> laenge (_:xs) = 1 + laenge xs
```

Argumentpattern in den Funktionsgleichungen können auch verschachtelt und beliebig komplex sein. So lässt sich das letzte Element einer Liste, durch folgende Definition beschreiben:

Listen.lhs

```
> letzteLi (x:>E) = x
> letzteLi (_:>xs) = letzteLi xs
```

Für eingebaute Listen gibt es für Listen mit einer fest vorgegebenen Anzahl von Elemente ein eigenes Pattern. Statt der folgenden Gleichungen:

Listen.lhs

```
> letzte1 (x:[]) = x
> letzte1 (_:xs) = letzte1 xs
```

Lässt sich das letzte Element auch durch folgende äquivalente Definition schreiben:

Listen.lhs

```
> letzte2 [x] = x
> letzte2 (_:xs) = letzte2 xs
```

Patterngleichungen können natürlich auch Guards enthalten. So lässt sich eine Funktion, die testet, ob eine Liste sortiert ist, formulieren als:

Listen.lhs

```
> istSortiertLi (x1:>x2:>xs)
>   |x1<=x2 = istSortiertLi (x2:>xs)
>   |otherwise = False
> istSortiertLi _ = True
```

Oder für die Standardliste:

Listen.lhs

```
> istSortiert (x1:x2:xs)
>   |x1<=x2 = istSortiert (x2:xs)
>   |otherwise = False
> istSortiert _ = True
```

Bei dieser Funktion fällt auf, dass das Teilpattern `(x2:xs)` für die rechte Seite der Funktion wieder benötigt wird. Daher ist es mögich, Teilpattern mit einer Variablen zu bezeichnen. Diese wird dem Pattern mit dem Symbol `@` getrennt voran gestellt.

So wäre eine alternative Formulierung:

Listen.lhs

```
> istSortiert2 (x1:ys@(x2:xs))
>   |x1<=x2 = istSortiert2 ys
>   |otherwise = False
> istSortiert2 _ = True
```

1.3 Unendliche Listen

Wir können Ausdrücke definieren, die zu einer unendlichen Liste auswerten. Eine einfache Art, einer solchen Liste lässt sich durch folgende Funktion erzeugen:

Listen.lhs

```
> wiederholeLi x =  x:>wiederholeLi x
```

Oder entsprechend für die Standardlisten als:

Listen.lhs

```
> wiederhole x =  x:wiederhole x
```

Der Ausdruck `wiederhole 42` wertet dann zu einer unendlichen Liste aus. Wenn wir aber nur einen Teil dieser Liste benötigen, dann können wir diese unendliche Liste verarbeiten. So berechnet die Standardfunktion `take n` die Teilliste der ersten n Elemente einer Liste. Diese ist wieder endlich:

```
take 10 $ wiederhole 42
```

```
[42,42,42,42,42,42,42,42,42,42]
```

Grund hierfür ist die lazy Auswertung von Haskellprogrammen. Als Java-Programmierer kann man die Listen in Haskell eher mit einem Iterator oder einem Stream vergleichen. Erst nach Bedarf wird dort mit `next` das nächste Element der Iteration generiert.

1.4 Aufzählungen

Es gibt noch eine einfache und praktische Art, um Listen zu erzeugen, indem man für aufzählbare Werte mit zwei Punkten den Aufzählungsbereich beschreibt. Die natürlichen Zahlen von 1 bis n lassen sich in einer Liste durch `[1,2..n]` beschreiben.

So gelangen wir mit der Standardfunktion `product`, die das Produkt aller Listenelemente errechnet, zu folgender Definition der Fakultät.

Listen.lhs

```
> factorial1 n = product [1,2..n]
```

In dieser Schreibweise geben die ersten beiden Elemente die Schrittweite zwischen den Elementen an. Die ungeraden Zahlen kleiner 10 lassen sich definieren als:

```
[1,3..10]
```

```
[1,3,5,7,9]
```

In dieser Aufzählungsschreibweise kann das Endelement auch fehlen, so dass wieder eine unendliche Liste erzeugt wird. So ließe sich auch die Fakultät wie folgt definieren:

Listen.lhs

```
> factorial2 n = product$take n [1,2..]
```

1.5 Mengenschreibweise

Ein besonderer syntaktischer Zucker für Listen in Haskell ist die Mengenschreibweise, die als *list comprehension* bezeichnet werden. In der Mathematik lassen sich Mengen zum Beispiel beschreiben als:

$$\{(x, x^y) | x \in \{1, 2 \dots\}, y \in \{2, 3, 5\}, x\%2 == 1\}$$

Ebenso lassen sich in Haskell Listen erzeugen:

```
take 10 [(x,x^y)|x<-[1,2..],y<-[2,3,5],x`mod`2==1]
```

```
[(1,1),(1,1),(1,1),(3,9),(3,27),(3,243),(5,25),(5,125),(5,3125),(7,49)]
```

Ein wenig ersetzt diese Schreibweise eine foreach-Schleife. In der Sprache Scala könnte man die folgende Schleife schreiben:

```
for (x<-List(1,3,5) if x%2==1;y<-List(2,3,5)) yield (x,scala.math.pow(x,y))
```

```
res: List[(Int, Double)] = List((1,1.0), (1,1.0), (1,1.0), (3,9.0), (3,27.0),
(3,243.0), (5,25.0), (5,125.0), (5,3125.0))
```

Die Mengenschreibweise hat also:

- einen Ausdruck, der die Konstruktion der Elemente der Ergebnisliste beschreibt, im Beispiel: `(x,x^y)`.
- Generatoren für Laufvariablen, im Beispiel `x<-[1,2..]` und `y<-[2,3,5]`.
- bool'sche Ausdrücke, die über die Generatoren filtern, im Beispiel: `x`mod`2==1`.

2 Aufgaben

So, und nun wird es Zeit, die ersten eigenen Haskell-Funktionen für Listen zu schreiben und auszuprobieren.

Aufgabe 1 Machen Sie sich mit den Listen-Funktionen des Standard-Prelude vertraut. Gehen Sie das API durch und machen im Interpreter `ghci` eigene Testaufrufe.

Insbesondere die folgenden Funktionen sollten Sie gut kennen: `length`, `(++)`, `map`, `filter`, `concat`, `null`, `foldl`, `foldr`, `iterate`, `repeat`, `take`, `drop`, `span`, `elem`, `lookup`, `zip`, `sort`, `nub`.

Aber alle anderen auch…

Aufgabe 2 Wie Sie gesehen haben, ist es schwer, weitere Aufgaben zu Listen zu entwerfen, da fast alles bereits im Prelude definiert ist. Wir versuchen trotzdem ein paar weitere Listenfunktionen als Aufgaben zum Üben zu definieren.

a) Schreiben Sie eine Funktion `isPalindrome`, die testet, ob eine Liste rückwärts wie vorwärts gelesen die gleiche Elementreihenfolge ergibt.

Listen.lhs

```
> isPalindrome :: Eq a => [a] -> Bool
> isPalindrome xs = False
```

b) Schreiben Sie eine Funktion `everyNth`, die für ein $n > 0$ die Teilliste einer Liste $[x_0, x_1, x_2, \dots]$ der jeweils n-ten Elemente
$[x_0, x_n, x_{2*n}, x_{3*n}, \dots]$
erzeugt.

Zwei Beispielaufrufe:

```
everyNth 2 [1,2..10]

[1,3,5,7,9]
```

Oder für jedes dritte Element, beginnend mit dem ersten Element.

```
everyNth 3 [1,2..10]

[1,4,7,10]
```

Listen.lhs

```
> everyNth :: Integral a => a -> [b] -> [b]
> everyNth _ _ = []
```

c) Schreiben Sie eine Funktion `swapNeighbours`, die in einer Liste alle aufeinanderfolgende Elemente vertauscht also aus $[x_0, x_1, x_2, x_3, \dots]$ die Liste $[x_1, x_0, x_3, x_2, \dots]$ erzeugt.

Listen.lhs

```
> swapNeighbours :: [a] -> [a]
> swapNeighbours xs = xs
```

Ein Beispielaufruf:

```
swapNeighbours [0,1..10]

[1,0,3,2,5,4,7,6,9,8,10]
```

d) Schreiben Sie eine Funktion, die die maximale Länge einer Teilliste mit gleichen aufeinander folgenden Elemente berechnet

Listen.lhs

```
> maxRep :: Eq a => [a] -> Int
> maxRep _ = 0
```

Ein Beispielaufruf:

```
maxRep "aabbbbccddeeeehhhhhhhhgjgfzvkgjlkj"
```

```
8
```

Aufgabe 3 In dieser Aufgabe wollen wir uns mit Reihen beschäftigen, die zur Kreiszahl π konvergieren.

a) Die Leibnizreihe konvergiert nach $\frac{\pi}{4}$.

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

Schreiben Sie eine nullstellige Funktion, die die unendliche Liste der Terme der Leibnizreihe für den Typ `Rational` erzeugt:

Listen.lhs

```
> leibniz:: [Rational]
> leibniz = [k  {-- ToDo --}|k<-[0,1..]]
```

Mit Ihrer Lösung sollten Sie in der Lage sein `pi` annähernd berechnen zu können, wenn auch nicht sehr effizient.

Die ersten 10 Elemente dieser Liste:

```
take 10 leibniz
```

```
[1%1,(-1)%3,1%5,(-1)%7,1%9,(-1)%11,1%13,(-1)%15,1%17,(-1)%19]
```

Mit den ersten 1000 Elementen können wir π annähern:

```
fromRational  (4* (sum.take 10000) leibniz)
```

```
3.1414926535900434
```

Das entspricht fast genau der Konstante π:

```
pi
```

```
3.141592653589793
```

b) Mit der Eulersche Reihentransformation kann man aus der Leibniz-Reihe die Fatio-Reihe bekommen.

$$\frac{\pi}{2} = 1 + \frac{1}{1 \cdot 3} + \frac{1 \cdot 2}{1 \cdot 3 \cdot 5} + ... + \frac{1 \cdot 2...n}{1 \cdot 3 \cdot 5...(2n+1)} + ...$$

Schreiben Sie eine nullstellige Funktion, die die unendliche Liste der Terme der Fatio-Reihe für den Typ `Rational` erzeugt:

Listen.lhs

```
> fatio:: [Rational]
> fatio = [k  {-- ToDo --}|k<-[0,1..]]
```

Jetzt sollten Sie in der Lage sein, effizienter die Kreiszahl π anzunähern:

```
take 10 fatio
```

```
[1%1,1%3,2%15,2%35,8%315,8%693,16%3003,16%6435,128%109395,128%230945]
```

Jetzt reichen die ersten 55 Elemente zu Annäherung an π.

```
fromRational(2*(sum$take 55 fatio))
```

```
3.141592653589793
```

Aufgabe 4 Der Verschiebungssatz in der Statistik ist eine Rechenregel für die Ermittlung der Summe der Abweichungsquadrate SQ_x für n Zahlen $x_1, \ldots, x_n$ und deren arithmetisches Mittel $\overline{x}$ nach folgender Gleichung.

$$SQ_x = \sum_{i=1}^{n} (x_i - \overline{x})^2 = \left(\sum_{i=1}^{n} x_i^2\right) - n\overline{x}^2 = \left(\sum_{i=1}^{n} x_i^2\right) - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2$$

Eine Funktion zur Berechnung der Summe einer Sammlung von Elementen existiert schon im Prelude:

```
:t sum
```

```
sum :: (Foldable t, Num a) => t a -> a
```

Sie existiert sogar nicht nur für Listen numerischer Werte, sondern für alle Typen, in denen mit einer Faltung verknüpfbare Elemente sind. Listen sind ein solcher Typ.

a) Schreiben Sie eine Funktion zur Berechnung der Summe der Quadrate.

   $\sum_{i=1}^{n} x_i^2$ der Elemente einer Liste.

   Listen.lhs

```
> squaresum :: (Foldable t, Num a, Functor t) => t a -> a
> squaresum xs = 0
```

b) Schreiben Sie jetzt unter Verwendung von `sum` und `squaresum` die Funktion, die die Summe der Abweichungsquadrate SQ_x berechnet.

   Um einen möglichst allgemeinen Typ zu erhalten, verwenden Sie für die Listenlänge die Funktionskomposition `(fromIntegral . length)`.

   Listen.lhs

```
> sqx :: (Fractional a, Foldable t, Functor t) => t a -> a
> sqx xs = 0 -- Todo
>   where
>    l = fromIntegral . length
```

Aufgabe 5 Für eine Folge von Werten $\{x_1, x_2, \cdots, x_n\}$, lassen sich die Mittelwerte durch folgende Rekursionsformel berechnen:

$$m_0 = 0 \tag{1}$$

$$m_n = \frac{(n-1) * m_{n-1}}{n} + \frac{x_n}{n} \tag{2}$$

Dabei gilt:

$$m_n = \frac{1}{n} \sum_{i=0}^{n} x_i$$

Implementieren Sie eine Funktion, die für eine Liste von Zahlenwerten die Liste der Mittelwerte erzeugt. Nutzen Sie hierfür die Funktion `scanl` aus dem Prelude. Die Argumentliste können Sie dabei geschickt mit einem entsprechenden Aufruf der Funktion `zip` durchnummerieren.

Listen.lhs

```
> mittelwerte :: (Fractional a, Enum a) => [a] -> [a]
> mittelwerte xs = [] --ToDo
```

Aufgabe 6 Schreiben Sie eine Funktion, die einen String aus Nullen und Einsen als natürliche Zahl einliest.

Listen.lhs

```
> readBinary:: String -> Integer
> readBinary xs = aux 0 xs
>   where
>     aux result xs = result
```

Aufgabe 7 Schreiben Sie eine Funktion, die eine Zahl als String darstellt, der diese Zahl im Oktalsystem repräsentiert.

Listen.lhs

```
> toOctalString :: (Show a, Integral a) => a -> String
> toOctalString x = ""
```

Aufgabe 8 Im Prelude gibt es für Listen die Funktion `permutations`, die eine Liste aller Permutationen (Vertauschungen von Elementen) der Argumentliste berechnet:

```
:t permutations
```

```
permutations :: [a] -> [[a]]
```

Hier ein Beispielaufruf.

```
permutations "abc"
```

```
["abc","bac","cba","bca","cab","acb"]
```

der Ergebnisliste hat eine Länge von 6:

```
length $ permutations "abc"
```

```
6
```

Diese Funktion betrachtet dabei keine Gleichheit auf den Elementen. So hat die Ergebnisliste immer `factorial.length` Elemente.

```
permutations "aaa"
```

```
["aaa","aaa","aaa","aaa","aaa","aaa"]
```

Zum Beispiel für die Bildung aller Anagramme eines Wortes sind keine doppelten Elemente in der Ergebnisliste erwünscht.

Möchte man eine Funktion `anagram` haben, die alle Permutationen einer Liste ohne doppelte Elemente berechnet, so lässt sich diese schnell definieren als:
`nub.permutations`

Die Preludefunktion `nub` steht für *no duplicates* und erzeugt eine Liste ohne weitere doppelte Auftreten von Elementen.

```
(nub.permutations) "aaa"
```

```
["aaa"]
```

Diese Umsetzung ist aber hoffnungslos ineffizient. Schon recht einfache Ausdrücke benötigen mehrere Sekunden zum Auswerten:

```
length $nub$permutations  "aaabbbbbbb"
```

```
120
```

Schreiben Sie in dieser Aufgabe eine Funktion, die alle unterschiedlichen Anagramme einer Argumentliste generiert.

Gehen Sie dabei zweistufig vor:

a) Schreiben Sie eine Funktion `occurrences`, die für eine Liste eine Liste von Paaren aus den Listenelemente und ihre Häufigkeit in der Argumentliste erzeugt:

```
occurrences "aabab"
```

```
[(2,'b'),(3,'a')]
```

Listen.lhs

```
> occurrences :: (Num a, Eq t) => [t] -> [(a, t)]
> occurrences _ = [] --ToDo
```

b) Nutzen Sie die Funktion `occurrences`, um alle Anagramme einer Liste zu bilden:

Listen.lhs

```
> anagram :: Eq a => [a] -> [[a]]
> anagram xs = [] --ToDo
```

3 Lernzuwachs

- Lazy Listen.
- Listenpattern
- Mengenschreibweise.
- Funktionen höherer Ordnung auf Listen.

Listenkonstruktoren
Listenliterale
Strings
Listpattern
Unendliche Listen
Aufzählungen
Mengenschreibweise