# Haskell mit Hugs98 - Ein Tutorial

Clemens Adolphs

7. April 2005

**Zusammenfassung**

Die meisten angehenden Informatiker werden, so wie ich am Anfang des ersten Semesters, keine Erfahrung mit funktionalen Programmiersprachen haben sondern eher mit Java, C, C++ und anderen imperativen so wie objektorientierten Sprachen. Aus diesem Grund habe ich mich dazu entschlossen, in diesem Tutorial nicht nur die Sprachkonzepte und die Syntax, sondern auch die Denkweise hinter dem funktionalen Programmieren zu erläutern. Als Sprache benutze ich hierzu Haskell und den Hugs-Interpreter, einfach weil dies die Sprache ist, die man an der RWTH-Aachen im ersten Semester lernt... Es handelt sich hier erst einmal um eine Art Rohfassung. Wer weitere Anregungen zum Inhalt hat oder ein bestimmtes Thema lieber noch ausführlicher hätte kann mir das mitteilen, genau so wie Fehler oder sonstiges Feedback:
clemens.adolphs@t-online.de

# Inhaltsverzeichnis



# 1 Einleitung

## 1.1 Wozu das alles!?

Ja echt. Reichen C++ und so nicht aus? Warum gleich ein völlig neues Konzept lernen? Dazu von mir nur zwei Kommentare:

- Angeblich hat die Firma ERICSSON jede Menge Zeit und Geld gespart, seit deren Programmierer auf funktionales Programmieren umgestiegen sind...

- Funktionales Programmieren ist elegant. Quicksort ist in Haskell ein Dreizeiler!

### 1.2 Das Grundkonzept

Die Grundidee imperativer Programme ist das Abarbeiten - Zeile für Zeile - von einem gegebenen Programm, eventuell gesteuert durch Schleifen und Verweigungen.
Bei funktionalen Programmiersprachen ist die Grundidee das schrittweise Auswerten eines Ausdruckes. Ein Ausdruck ist so etwas wie $3 + 4 - 2 * 1$, der zu 5 ausgewertet wird.
Das Auswerten erledigt ein sogenannter Interpreter, wie der oben genannte Hugs-Interpreter. Es gibt verschiedene Auswertungsstrategien, mehr dazu später.
Ein Programm in Haskell besteht im Prinzip nur aus Funktionen [1]. Diese kann man dann später in seinen Ausdrücken benutzen.

### 1.3 Ein erstes Beispiel

Hier also mal ein „Programm“ in Haskell:

```
square :: Int -> Int
square x = x * x
```

Die erste Zeile ist nicht zwingend notwendig, aber sie erklärt dem System, dass `square` eine Funktion ist, die Integer auf Integer abbildet. Die zweite Zeile sagt dann, was `square` mit einem $x$ tut. Wir nennen dabei `x` auch Parameter oder Argument der Funktion.
Der Interpreter würde nun also den Ausdruck `square 3` durch `3* 3` ersetzen und diesen Ausdruck weiter zu 9 auswerten.

## 2 Die Grundkonzepte

### 2.1 Auswertungsstrategien

Was passiert nun, wenn man einen etwas komplizierten Ausdruck hat, wie z.B. `square`(3 + 2) ? Nun gibt es mehrere Möglichkeiten:

- Der Ausdruck wird zu `square 5` ausgewertet und dann weiter wie im Beispiel oben zu `5 * 5` und weiter zu 25
- Der Ausdruck wird zu `(3 + 2) * (3 + 2)` ausgewertet, dann zu `5 * (3 +` 2) und schliesslich zu `5 * 5` und 25

Die erste Variante nennt man *strikte Auswertung*, die zweite entsprechend *nichtstrikte Auswertung*.
Klar: Das *Ergebnis* darf nicht von der Auswertungsstrategie abhängen. Ob strikt oder nicht, man kommt zum selben Ergebnis.
Interessant wird es, wenn es um das Terminieren geht. Dazu basteln wir uns eine Endlosschleife:

[1] Und eigenen Datentypen, mehr dazu später

```
endlos :: Int -> Int
endlos x = endlos (endlos x)
```

Man sieht leicht, dass ein Ausdruck wie `endlos 3` zu einer Art Endlosschleife führt und der Interpreter irgendwann einen Stack-Overflow meldet. Schauen wir uns jetzt mal diese Funktion an:

```
drei :: Int -> Int
drei x = 3
```

Man sieht: Die Funktion `drei` spuckt immer eine 3 aus, also `drei` $4 = 3$, `drei` $(3 * 4 + 2 - 1) = 3$ und so weiter.
Jetzt wird es interessant: Wir schauen uns `drei (endlos 1)` an. Bei der strikten Auswertung versucht das System zunächst, `endlos 1` auszuwerten. Das führt aber zu einem Stack-Overflow, also terminiert die Anfrage an das Haskell-System nicht!
Bei der nicht-strikten Auswertung wird zunächst die Funktion `drei` ausgewertet, und da kommt für jedes beliebige x 3 raus.
Wir sehen: Die Art der Auswertung bestimmt das Terminierungsverhalten.
Es gilt:

| Wenn irgend eine Art der Auswertung für einen Ausdruck terminiert, terminiert auch die nicht-strikte Auswertung |
|---|

Bei Haskell wird die *nicht-strikte* Auswertung benutzt. Eine kleine Besonderheit ist hierbei noch ein Konzept, dass sich *Lazy-Evaluation* nennt: Wird die selbe Variable in einem Ausdruck mehrmals benutzt, so merkt der Interpreter dies und wertet dazu gehörende Ausdrücke nur einmal aus:

```
square (3 + 2)
(3 + 2) * (3 + 2)
5 * 5
25
```

Das System rechnet dann nur *einmal* $3 + 2$ aus.

## 2.2 Typen

Hier mal ein paar gebräuchliche Typen, und wie man damit umgeht. Eine größere Übersicht bzw. eine Übersicht der speziell in der Vorlesung benutzen Typen findet man am besten bei den Vorlesungsunterlagen. Hier also ein grober Umriss:

### 2.2.1 Ganzzahlen

Da gibt es `Int` und `Integer`. Dabei hat `Integer` *infinite-precision*, also theoretisch unendliche Genauigkeit. Für ganz normale Fälle reicht also `Int`.

### 2.2.2 Gleitkommazahlen

Diese werden mit `Float` deklariert.

### 2.2.3 Char

Der Typ `Char` steht für einzelne Zeichen. Man nimmt den einfachen Akzent, um einen Char zu kennzeichnen: `'a'`

### 2.2.4 Bool

Vom Typ `Bool` sind die Wahrheitswerte `True` und `False`, die in keiner Programmiersprache fehlen dürfen.

### 2.2.5 Listen

Setzt man einen Typ zwischen zwei eckige Klammern erhält man einen Typ „Liste vom Typ“. Beispiel: `[Int]` ist eine Liste von ganzen Zahlen und `[Char]` ist eine Liste von Zeichen. In Haskell schreibt man eine konkrete Liste so: `[1,2,3]` oder ähnlich. Eine andere Möglichkeit, Listen zu erzeugen ist der *Listenkonstruktor*. Mit dem : fügt man ein Element vorne in eine Liste ein:
`1`: [2,3] wird zu `[1,2,3]`. Dabei muss das einzufügende Element den selben Typ haben, wie die Elemente, die schon in der Liste sind!
`'a'`: [2,3] führt zu zeinem Fehler.
Man kann übrigens `String` statt `[Char]` schreiben und dann die Schreibweise `"Hallo"` benutzen.

### 2.2.6 Tupel

Tupel schreibt man in der Form $(Wert_1, Wert_2, \cdots, Wert_n)$, z.B. `(Int,`Char, [Int]) . Dies bezeichnet ein Tupel mit einer ganzen Zahl an erster Stelle, einem Zeichen an zweiter Stelle und einer Liste von ganzen Zahlen an dritter Stelle. Ein Tupel dieses Typs könnte so aussehen:
`(3,`'x', [10,9,8])

### 2.2.7 Funktionen

In Haskell sind auch Funktionen Objekte, mit denen man umgehen kann wie mit jedem anderen Typ auch: Funktionen können Argument oder Rückgabewert einer anderen Funktion haben. Mehr dazu später. Den Funktionstyp deklariert man mit `->`, wie man schon im ersten Beispiel sieht.

## 2.3 Einfache Funktionen

Im ersten Beispiel hat man schon einmal gesehen, wie man eine Funktion grundsätzlich deklariert. Zunächst den Typen, dann die Funktion selbst. Schauen wir uns noch einmal die Funktion an:

```
square :: Int -> Int
square x = x * x
```

Mit dem :: zeigt man dem System, dass es sich nun um eine Typdeklaration handelt. Man kann den :: also lesen als „hat den Typ:“. Das $->$ steht für den sogenannten Funktionsraumkonstruktor. Er sagt: Ein Argument mit dem Typ, der links steht wird auf etwas abgebildet, das den rechten Typ hat. `square`

hat also den Typ „Funktion, die ganze Zahlen auf ganze Zahlen abbildet“. Die Schreibweise erinnert stark an die mathematische Schreibweise für Abbildungen:

$$\text{square} : \mathbf{N} \to \mathbf{N} : x \mapsto x \cdot x$$

Die nächste Zeile definiert nun, was die Funktion so macht. Dabei werde ich den Teil links vom Gleichheitszeichen als Funktions*kopf* bezeichnen, den Teil danach als Funktions*rumpf*.
Man kann auch mehr als nur eine Definition für die Funktion benutzen. Schauen wir uns das Beispiel an:

```
bla :: Int -> Bool
bla 0 = False
bla 1 = True
```

Dies erinnert an stückweise definierte mathematische Formeln und funktioniert auch genau so! Ruft man nun `bla 1` auf sucht sich der Interpreter die passende Zeile des Programms dazu aus. Dies bezeichnet man auch als *Pattern Matching*.

## 2.4 Pattern Matching

### 2.4.1 Grundidee

Mit Pattern Matching[2] bezeichnet man das Verfahren, mit welchem der Interpreter bei einem Funktionsaufruf feststellt, welchen Funktionsrumpf er benutzen soll. Um also einen Funktionsaufruf auszuwerten geht der Interpreter die Funktionsköpfe von oben nach unten durch, bis er einen findet, der von der Struktur her zum Argument des Aufrufes passt. Hier hilft wieder ein Beispiel:
Die berühmte Fibonaccifolge ist wie folgt definiert:

$$\text{fibo(n)} = \begin{cases} 1 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ \text{fibo}(n-1) + \text{fibo}(n-2) & \text{sonst} \end{cases}$$

In Haskell schreibt sich das:

```
fibo :: Int -> Int
fibo 0 = 1
fibo 1 = 1
fibo n = fibo (n - 1) + fibo (n - 2)
```

Für die Aufrufe `fibo 0` und `fibo 1` wird sofort 1 zurückgegeben, für jeden anderen Aufruf wird der Parameter für `n` eingesetzt und damit der Funktionsrumpf ausgewertet, also:

```
fibo 3
fibo 2 + fibo 1
fibo 1 + fibo 0 + fibo 1
1 + 1 + 1
3
```

[2] Man übersetzt es am besten mit „Mustererkennung“ oder „Musterabgleich“

**W**ichtig: Da der Interpreter von oben nach unten einen passenden Funktionskopf sucht und auch *nur* die erste passende Variante wählt muss man bei der Funktionsdeklaration auf die Reihenfolge achten:

```
fibo n = fibo (n - 1) + fibo (n - 2)
fibo 0 = 1
fibo 1 = 1
```

Dieses Beispiel würde immer zu einem Stack-Overflow führen, da der Interpreter niemals zu den Zeilen `fibo 0` oder `fibo 1` kommt!
Passen Funktionsaufruf und Pattern zusammen sagt man auch, der Ausdruck *matcht* das Pattern. In unserem Beispiel matcht die 4 also das Pattern `n`.

### 2.4.2 Die verschiedenen Pattern

Zwei Möglichkeiten für ein Pattern haben wir schon gesehen: Einmal einen konstanten Wert wie 0 oder 1, einmal eine einzelne Variable wie $x$ oder $n$. Hier eine kleine Übersicht, was es noch so gibt:

**Konstante** Alle Patterns wie 1 oder `'a'` fasst man unter dem Begriff der konstanten Patterns zusammen. Wann ein Ausdruck diese Patterns matcht ist logisch: Wenn er den selben Wert wie die Konstante des Patterns hat.

**Einfache Variable** Ein Pattern der Form `x`. Ein Ausdruck matcht auf dieses Pattern, wenn er den selben Typ hat, wie in der Funktionsdeklaration angegeben. Der Interpreter setzt dann für die Variable das Argument ein.

**Wildcard** Auf den Unterstrich `_` matcht alles. Dabei wird allerdings nichts zugewiesen, der Ausdruck verfällt. Beispiel:

```
blub :: Int -> Bool
blub 1 = True
blub _ = False
```

Für 1 erhält man also `True`, für alles andere `False`. Man hätte auch `blub x = False` schreiben können. Aber mit dem Wildcard macht man deutlicher, dass der konkrete Wert egal ist!

**Tupel** Auch ein Tupel kann man als Pattern benutzen, wobei jeder einzelne Eintrag dieses Tupels wieder ein Pattern ist. Beispiel: Wir stellen uns die komplexen Zahlen als Tupel von zwei Zahlen vor und schauen uns diese Funktion an:

```
realteil :: (Float, Float) -> Float
realteil (x, _) = x
```

Hier benutzen wir also einmal das Variablen- und einmal das Wildcardtupel.

**Listen** Auch eine Liste kann als Pattern herhalten, das funktioniert dann genau wie bei den Tupeln, z.B.

```
switch :: [Int] -> [Int]
switch [x, y] = [y,x]
```

Bei Listen gibt es aber noch eine andere Möglichkeit, wenn man mit dem : als Listenkonstruktor arbeitet. Dazu ein Beispiel:

```
laenge :: [Int] -> Int
laenge [] = 0
laenge (x : xs) = 1 + laenge xs
```

Auf das erste Pattern matcht nur die leere Liste. Jede andere Liste kann man aber in der Form x : xs schreiben, z.B. [1,2,3] = 1 : [2,3] . Der Interpreter weist also dem x das erste Element der Liste zu und dem xs die Restliste.

**Rechnendes Pattern** Die interne Darstellung positiver Ganzzahlen als $1+1+\cdots+1$ erlaubt es, Patterns wie x + 1 zu benutzen:

```
vorgaenger :: Int -> Int
vorgaenger (x + 1) = x
```

Dies ist zwar theoretisch möglich, streng genommen aber überflüssig, da man das rechnen weg vom Pattern in den Funktionsrumpf packen kann und auch sollte!

**Konstruktoren** Später, wenn wir eigene Datentypen einführen, werden wir sehen, dass jeder beliebige Datenkonstruktor für ein Pattern gut sein kann. Hier steht es jetzt nur wegen der Vollständigkeit.

## 2.5 Polymorphie

Schauen wir uns noch einmal die laenge Funktion an, wie sie bisher definiert war:

```
laenge :: [Int] -> Int
laenge [] = 0
laenge (x : xs) = 1 + laenge xs
```

Nicht so schön, denn die Länge einer Liste von char oder die Länge einer Liste von Tupeln oder überhaupt die Länge jeder x-beliebigen Liste berechnet man auf die gleiche Weise.
Unsere Funktion funktioniert aber nur mit Listen von ganzen Zahlen. Hier hilft uns das Konzept der *parametrisierten* Polymorphie. Schauen wir uns folgendes Listing an:

```
laenge :: [a] -> Int
laenge [] = 0
laenge (x : xs) = 1 + laenge xs
```

Der Kleinbuchstabe in der Typdeklaration sagt: „Hier könnte jeder beliebige Typ stehen“. Es funktioniert also für Int-Listen, Bool-Listen, String-Listen, Listen von Funktionen und allem was man sich denken kann.

**Wichtig:** Gleiche Buchstaben bedeutet gleiche Typen:

```
erstes :: [a] -> a
erstes (x : xs) = x
```

Das `a` kommt doppelt vor, also weiß der Interpreter: Ich bilde eine Liste von Elementen des Typs `a` auf ein Element des Typs `a` ab, z.B. eine Int-Liste auf einen Int.

## 2.6 Currying

Bisher hatten unsere Funktionen nur einen Parameter, z.B. `square x`. Bei Funktionen mit mehr als einem Parameter liegt es nahe, Tupel zu benutzen, wie hier:

```
plus :: (Int, Int) -> Int
plus (x, y) = x + y
```

Aufruf z.B. über `plus (3, 4)`. Das hat den Nachteil, dass man sehr viele Klammern setzen muss und es - bei mehreren Parametern, von denen manche ja durchaus wieder Tupel oder Listen sein können - sehr unübersichtlich wird. Eine Alternative sieht so aus:

```
plus :: Int -> Int -> Int
plus x y = x + y
```

Formal gesehen reiht man also einfach alle Parameter hintereinander. Der Aufruf lautet dann `plus 3 4`.
Was bedeutet diese Schreibweise? Die erste Zeile sagt uns: `plus` ist eine Funktion, die einen `Int` auf eine Funktion abbildet, die vom Typ `Int-> Int` ist. Man beachte hierbei: Der `->`-Operator assoziiert nach rechts, die Zeile kann also auch als `plus :: Int -> (Int -> Int)` gelesen werden.
Neben dem Einsparen von Klammern bringt uns das noch mehr. Wir können die Funktion stückweise aufrufen: `plus 1` liefert uns diejenige Funktion, die eine ganzzahl um `1` erhöht. Wir könnten z.B. schreiben:

```
inc :: Int -> Int
inc x = plus 1 x
```

oder noch kürzer

```
inc :: Int -> Int
inc = plus 1
```

Das ist einerseits sehr elegant, andererseits wird es später noch sehr nützlich, wenn man mit anonymen Funktionen arbeitet. Mehr dazu später.
Man kann mit der vordefinierten Funktion `curry` eine bereits vorhandene Funktion von der Tupelschreibweise in die Curryform bringen und andererseits eine bereits vorhandene Funktion mit dem Befehl `uncurry` von der Curry- in die Tupelschreibweise überführen:

```
plus1 :: (Int, Int) -> Int
plus1 (x, y) = x + y

plus2 :: Int -> Int -> Int
plus2 = curry plus1
```

plus2 hat dann die Form wie das plus im Currybeispiel oben.
Wen es interessiert: Intern könnte man curry so realisieren:

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x,y)
```

Und uncurry entsprechend

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

## 2.7 Rekursion

Bei imperativen Sprachen ist sie einfach nur schick und elegant, bei funktionalen Sprachen absolut notwendig: Rekursion. Einfache Beispiele hatten wir schon, z.B. bei den Fibonacci-Zahlen oder der Längenberechnung einer Liste.
Platt gesagt bezeichnet man eine Funktion als rekursiv, wenn sie sich selbst wieder aufruft.
Damit das ganze auch irgendwann auffhört, müssen zwei Voraussetzungen gelten:

1. Es gibt eine Abbruchbedingung, wie z.B. `laenge` [] = 0. Es muss also mindestens einen Funktionsrumpf geben, der nicht rekursiv ist.

2. Dieser nichtrekursive Rumpf muss auch irgendwann erreichbar sein:

   ```
   foo :: Int -> Int
   foo 0 = 1
   foo x = foo (foo x)
   ```

   Dies führt für alle Werte außer 0 zu einer Endlosrekursion, da die Abbruchbedingung nie erreicht wird.

Als Anfänger tut man sich oft schwer damit, Probleme rekursiv zu formulieren. Die Frage, die man sich stellen muss, lautet immer: „Was hat mein Problem mit dem nächstkleineren Problem zu tun?“ Machen wir uns das an einem Beispiel klar:
Gesucht ist eine Funktion, die das letzte Element einer Liste zurück gibt, z.B. `letztes` [1,2,3,4,5] = 5. Wie gehen wir das an?

- Als Abbruchbedingung eignet sich immer der allereinfachste Fall: Das letzte Element einer einelementigen Liste ist... das einzige Listenelement:

  ```
  letztes :: [a] -> a
  letztes [x] = x
  ```

- Nun fragen wir uns: Mit dem Listenkonstruktor kann ich die Liste aufspalten in ihr erstes Element und die Restliste. Was hat nun das letzte Element der Liste mit dem letzten Element der Restliste zu tun? Sie sind gleich! Das letzte Element von `[1,2,3]` ist das selbe wie das letzte Element von `[2,3]` . Also ist klar:

  ```
  letztes (x : xs) = letztes xs
  ```

# 3 Weiterführende Techniken

## 3.1 Spezielle Ausdrücke

### 3.1.1 Bedingte Ausdrücke

Schauen wir uns folgende mathematische Funktion an:

$$\text{signum}(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

Hier kommt man mit bloßem Pattern Matching nicht weiter! In Haskell schreibt man das so:

```
signum :: Float -> Int
signum x | x  > 0  =  1
         | x == 0    =  0
         | x  < 0    = -1
```

Man kann nach dem `|` auch `otherwise` schreiben, z.B.

```
maximum :: (Int, Int) -> Int
maximumx (x, y) | x < y      = x
                | otherwise  = y
```

### 3.1.2 Lokale Deklarationen

Um einen Funktionsaufruf übersichtlicher zu machen kann man Zwischenrechnungen in lokalen Variablen unterbringen.
Vorher:

```
nullstellen :: (Float, Float) -> (Float, Float)
nullstellen (p,q) =
  (-p/2 + sqrt((p/2)*(p/2) - q), -p/2 - sqrt((p/2)*(p/s) - q))
```

Besser:

```
nullstellen :: (Float, Float) -> (Float, Float)
nullstellen (p,q) = (a + b, a - b)
            where a = -p/2
                  b = sqrt(a*a -q)
```

### 3.1.3 Lambdaausdrücke

An manchen Stellen braucht man spontan eine Funktion, für die es sich aber nicht lohnt, eine extra Definition zu geben. Wir werden später Beispiele dafür sehen. So eine Funktion deklariert man nach dem Muster

```
\x -> x * x
\x y -> x + y
\(a,b) -> (b,a)
```

Ein kleines Beispiel:

```
inc :: Int -> Int
inc = \x -> x + 1
```

Die Zeile besagt: `inc` ist die Funktion, die `x` auf `x + 1` abbildet. Man könnte auch direkt in den Interpreter eingeben

```
(\x -> x + 1) 2
```

Das Ergebnis wäre natürlich 3.
Die Dinger nennt man übrigens Lambdaausdrücke, weil das \ ein stilisiertes $\lambda$ sein soll. Mit dem sogenannten Lambdakalkül kann man in der theoretischen Informatik eine ganze Menge anfangen. Der Beweis für die umrahmte Aussage zur Terminierung von Auswertungsstrategien z.B. lässt sich gut mit dem Lambdakalkül lösen.

## 3.2 Funktionen höherer Ordnung

Ganz toll an Haskell ist, dass Funktionen auch Objekte sind, die als Rückgabewert oder als Parameter fungieren können. Jede Funktion, die wieder mit Funktionen arbeitet nennt man Funktion höherer Ordnung.
Die vordefinierte Funktion `curry` ist so eine Funktion. Hier ein paar wichtige Beispiele:

### 3.2.1 Map

Gegeben ist eine Liste von ganzen Zahlen. Wir wollen eine Funktion schreiben, die jedes Element der Liste um 1 erhöht:

```
listeInc :: [Int] -> [Int]
listeInc [] = []
listeInc (x : xs) = (x + 1) : (listeInc xs)
```

Nun wollen wir eine Funktion, die jedes Element verdoppelt:

```
listeDbl :: [Int] -> [Int]
listeDbL [] = []
listeDbl (x : xs) = (2 * x) : (listeDbl xs)
```

Man sieht, dass die Struktur im Prinzip gleich ist:

```
listeBla :: [Int] -> [Int]
listeBla [] = []
listeBla (x : xs) = (bla x) : (listeBla xs)
```

Immer sieht es so aus, dass man sich das erste Element der Liste greift, darauf eine Funktion anwendet und dieses dann vorne in die rekursiv bearbeitete Liste einfügt und so die ganze Liste durchläuft.
Schön wäre, wenn man die Funktion gleich als Parameter mitliefern könnte. Das sieht dann so aus:

```
listeMap :: [a] -> (a -> b) -> [b]
listeMap [] _ = []
listeMap (x : xs) f = (f x) : (listeMap xs f)
```

Erklärung: Als Parameter haben wir eine Liste mit Elementen vom Typ `a`, dann eine Funktion, die Elemente des Typs `a` auf Elemente des Typs `b` abbildet und als Rückgabewert letztendlich eine Liste vom Typ `b`.
Bei der leeren Liste kommt immer die leere Liste raus. Bei jeder anderen Liste wenden wir unsere Funktion auf das erste Element an, danach auf die Restliste, und fügen das ganze wieder zusammen.
Um nun eine Liste von ganzen Zahlen zu verdoppeln können wir aufrufen

```
listeMap [1,2,3,4,5] (\x -> 2 * x)
```

Man sieht hier auch schön die Verwendung des Lambdaausdrücke.
Nun gibt es noch jede Menge andere Möglichkeiten, mit einer Liste umzugehen, wo sich Funktionen höherer Ordnung anbieten:

### 3.2.2 Fold

Sollen alle Elemente einer Liste zusammen addiert werden? Oder Multipliziert? Dann brauchen wir eine Funktion, die als Eingabe die Liste enthält sowie zwei Funktionen: Eine, die auf den Elementen der Liste arbeitet und eine, die dieses Element mit dem Rest verknüpft:

```
listeFold :: [a] -> (a -> b) -> (b -> b -> c) -> [c]
listeFold [] _ _ = []
listeFold (x : xs) f g = g (f x) (listeFold xs f g)
```

Um also alle Elemente einer Liste miteinander zu addieren schreiben wir

```
listeFold [1,2,3,4,5] (\x -> x) (\x y -> x + y)
```

### 3.2.3 Zip

Zwei Listen sollen über eine Funktion miteinander verknüpft werden:

```
listeZip :: [a] -> [b] -> (a -> b -> c) -> [c]
listeZip [] ys _ = ys
listeZip xs [] _ = xs
listeZip (x : xs) (y : ys) f = (f x y) : (listeZip xs ys f)
```

### 3.2.4 Filter

Eine Liste soll gefiltert werden. Wir brauchen dazu eine Funktion, die diese Bedingung ausdrückt. Diese Funktion wird vom Typ `Bool` sein:

```
listeFilter :: [a] -> (a -> Bool) -> [a]
listeFilter [] _ = []
listeFilter (x : xs) f | f x      = x : (listeFilter xs)
                       | otherwise = listeFilter xs
```

So ergibt z.B. der Aufruf

```
listeFilter [1,210,2,23,25,20] (\x -> x < 24) = [1,2,23,20]
```

## 3.3 Eigene Datentypen

### 3.3.1 Abkürzungen

Mit dem Schlüsselwort `type` kann man einen neuen Namen für einen bereits existierenden Typ einführen, z.B.

```
type Complex = (Float, Float)
type Vector3D = (Float, Float, Float)
```

Diese Definitionen darf man dann in seinen Funktionen genau so wie die Standardtypen verwenden:

```
skalarProdukt :: Vector3D -> Vector3D -> Float
skalarProdukt (x,y,z) (u,v,w) = x*u + y*v + z*w
```

Auch hier kann man die parametrisierte Polymorphie benutzen, also statt einem konkreten Typen einen Kleinbuchstaben benutzen.

```
type Paar a = (a, a)
```

Bei der Verwendung kann man dann den gewünschten Typ mit angeben, entweder in einer neuen Typdeklaration oder einem Funktionskopf, wie die zwei folgenden Beispiele zeigen:

```
type Punkt2D = Paar Float


blub :: Paar Bool -> Int
blub (True, True) = 1
blub _ = 0
```

Auch mehrstellige Parameter sind möglich:

```
type Paar a b = (a, b)
```

Dabei darf ein und der selbe Buchstabe nicht mehrmals auf der linken Seite auftauchen.
Außerdem darf so eine Deklaration nicht rekursiv sein: Der Typ, der links steht, darf nicht rechts irgenwo auftauchen.

### 3.3.2 Neue Datentypen

Mit dem Schlüsselwort `data` lassen sich gänzlich neue Datentypen einführen, z.B. durch eine Aufzählung von sogenannten Datenkonstruktoren:

```
data Antworten = Ja | Nein | Vielleicht
```

Die Datenkonstruktoren `Ja`, `Nein` und `Vielleicht` kann man sofort in seinen Patterns benutzen:

```
blub :: Antworten -> Float
blub Ja = 1
blub Nein = 0
blub Vielleicht = 0.5
```

Hierbei darf man auch Rekursion verwenden. Bestes Beispiel: Die natürlichen Zahlen[3]. Eine natürliche Zahl ist entweder die Null, oder sie ist Nachfolger einer natürlichen Zahl:

```
data Nat = Null | Nachfolger Nat
```

In der Schreibweise wäre 2 z.B. `Nachfolger(Nachfolger Zero)` . Auch hiermit kann man ganz normal Pattern Matching betreiben:

```
plus :: Nat -> Nat
plus Zero x = x
plus (Nachfolger x) y = Nachfolger (plus x y)
```

Auch hier muss man sich immer die Frage stellen: Was hat das reduzierte Problem mit dem eigentlichen Problem zu tun? Dann fällt es einem leicht, die richtige Rekursion zu finden.
Hier lautete die Frage: Was hat die Summe zweier Zahlen $x, y$ mit der Summe $x - 1, y$ zu tun? Klar: Sie ist deren Nachfolger.
Zusätzlich muss man wieder auf die Abbruchbedingung achten: Hätten wir in der zweiten Zeile

```
plus x Zero = x
```

geschrieben würde eine Anfrage nie terminieren, da in der dritten Zeile beim rekursiven Aufruf der *erste* Parameter geändert wurde (von `Nachfolger x` auf `x`. Man sollte ruhig noch ein bisschen mit dieser Definition der natürlichen Zahlen herumspielen und sich selber Beispiele ausdenken. Wie würde man z.B. multiplizieren?
Auch bei der `data`-Deklaration kann man parametrisierte Typen benutzen. Das beste Beispiel hierfür ist eine rekursiv definierte Liste. Eine Liste ist entweder leer, oder sie besteht aus einem Element, gefolgt von einer Liste:

```
Liste a = Leer | Element a (Liste a)
```

Die Liste `[1,2, 3]` wäre dann durch

```
Element 1 (Element 2 (Element 3 Leer))
```

gegeben.
Mit der so definierten Liste geht man im Prinzip genau so um, wie mit den vordefinierten Listen in Haskell, z.B.

```
laenge :: Liste a -> Nat
laenge Leer = Null
laenge (Element x (Liste xs)) = Nachfolger (laenge xs)
```

Natürlich sieht das nicht so schön aus und man braucht Unmengen an Klammern. Das ganze hat also mehr theoretisch-didaktischen Wert, und man kann viele schöne Aufgaben damit stellen...
Als letztes Beispiel eines solchen Datentypes möchte ich den Binärbaum nennen, und zwar einen, der nur Elemente in den Blättern, nicht aber den Knoten hat. So ein Baum ist entweder *leer*, oder ein Blatt, oder ein Knoten mit zwei Teilbäumen, also:

[3] Hier zählen wir mal die 0 hinzu, auch wenn die Mathematiker das nicht gerne sehen!

```
data Baum a = Leer | Blatt a | Knoten (Baum a) (Baum a)
```

Hier könnte man jetzt sehr viel zu sagen, wie man z.B. ähnliche Operationen, wie wir sie bei den Listen schon hatten, auf einen Baum anwendet (z.B. alle Elemente summieren, eine Funktion auf jedes Blatt anwenden usw. )
Hier ein Beispiel dafür, wie man die Funktion *map* (Eine Funktion jeweils auf die Elemente anwenden) für einen Baum realisieren könnte. Dazu die schrittweisen Überlegungen:

1. Fangen wir mit dem Typ an. Wir erwarten einen Baum und eine Funktion, die die Elemente in den Blättern auf neue Elemente abbildet. Wir erhalten also einen neuen Baum. Der Typ sieht dann so aus:

   ```
   mapBaum :: Baum a -> (a -> b) -> Baum b
   ```

2. Dann machen wir mit dem einfachsten Fall weiter: Der leere Baum gibt wieder einen leeren Baum

   ```
   mapBaum Leer _ = Leer
   ```

3. Der nächste Datenkonstruktor, den wir verarbeiten ist das Blatt. Bei einem einzelnen Blatt müssen wir einfach die Funktion auf das Element anwenden und erhalten dann ein neues Blatt:

   ```
   mapBaum (Blatt x) f = Blatt (f x)
   ```

4. Und zu guter letzt müssen wir die Funktion für einen Knoten definieren. Hier kommt die Rekursion ins Spiel: Die Funktion auf den ganzen Baum anzuwenden heißt, sie auf den linken und den rechten Teilbaum anzuwenden.

   ```
   mapBaum (Knoten links rechts) f =
          Knoten (mapBaum links f) (mapBaum rechts f)
   ```