Tag 3

Funktionen

Heute werden wir den wichtigsten Typ (oder die wichtigste Klasse von Typen) in Haskell überhaupt genau unter die Lupe nehmen: Funktionen, die wir ansatzweise schon am letzten Tag kennengelernt haben.
Am Ende der vorigen Lektion haben wir die Typen von Operatoren angesehen und festgestellt, daß die Tatsache, daß binäre Operatoren zwei Argumente haben, auf eine auf den ersten Blick etwas merkwürdige Weise dargestellt wird:

```
Prelude> :t (+)
forall a. (Num a) => a -> a -> a
```

Zur Erinnerung: Dies ist zu lesen wie eine logische Formel: Für alle (Typen) `a`, die das Prädikat `Num` erfüllen, haben wir . . . ja, was nun eigentlich? Bei

```
Prelude> :t abs
forall a. (Num a) => a -> a
```

war es ein `a -> a`, was wir als eine Funktion von `a` nach `a` interpretiert hatten. Der Funktionspfeil `->` ist ein sogenannter Typkonstruktor. Das ist eine Art Operator auf Typen, wiederum in Infix-Schreibweise mit zwei Argumenten, und er ist rechtsassoziativ. Das heißt, der Typausdruck `a -> a -> a` ist zu lesen als `a -> (a -> a)`. Wir haben also eine Funktion von `a` nach `a -> a`. Das wiederum bedeutet für `(+)`, daß es zunächst ein Argument von einem Typ, der `Num` erfüllt, erwartet, also zum Beispiel `Integer`. Das Ergebnis dieser ersten Applikation, im Beispiel `(+) 1` ist wieder eine Funktion, nämlich vom Typ `Integer -> Integer`. Diese erwartet dann wiederum einen Integer als Argument, so daß `(+) 1 2` dann vom Typ `Integer` ist (und `3` als Ergebnis hat). In der Tat ist `(+) 1`, wie wir bereits gesehen haben, ein gültiger Haskell-Ausdruck.
Diese Schachtelung von Funktionen ineinander ist eine gängige Methode in Haskell, um Funktionen mit mehreren Argumenten darzustellen. Sie hat sogar einen Namen: „Currying“ (nach Haskell B. Curry, einer Person, deren Vorname sogar für den Namen dieser Programmiersprache verantwortlich ist). Der Vorteil dieser Methode ist schon offenbar geworden: Es ist dadurch möglich, Funktionen nur partiell anzuwenden. So ist es machbar, zur Lösung eines Problems eine sehr generelle Funktion mit vielen Parametern zu entwickeln, und dann für Spezialfälle auf einfache Weise Funktionen abzuleiten, bei denen einige Parameter bereits appliziert sind.
Damit haben wir bereits eine Möglichkeit kennengelernt, „eigene“ Funktionen in Haskell zu definieren, nämlich durch partielle Applikation einer komplizierten Funktion. Das Gegenstück dazu ist die sogenannte „Lambda-Abstraktion“. Sie wird benutzt, um Funktionen aus dem Nichts heraus zu erschaffen:

```
Prelude> :t (\x -> 'a')
forall t. t -> Char
```

Der Backslash steht für ein Lambda. Das hat theoretische Hintergründe, die wir später wahrscheinlich noch immer mal wieder kurz beleuchten werden. Dahinter wird eine Variable eingeführt, das formale Argument der zu definierenden Funktion, in diesem Falle `x`. Man muß sich das so vorstellen, daß, wenn die Funktion appliziert wird, das `x` an das konkrete Argument der Funktion in diesem speziellen Aufruf gebunden wird. Die niedergeschriebene Funktion im Beispiel ist also zu lesen als „die Funktion, die `x` auf `'a'` abbildet". (In diesem Fall ist das nicht besonders eindrucksvoll, weil das `x` gar nicht verwendet wrd. Diese Funktion liefert immer ein `'a'`, egal für welches `x`.)
Wir sehen, daß der Typ, der für die Funktion abgeleitet ist, wieder eine quantifizierte Variable enthält. Er bedeutet: Für alle (wirklich alle) Typen `t` haben wir eine Funktion von `t` nach `Char`. Das ist, wie bereits erwähnt, nicht wirklich verwunderlich. Hinter dem `->` in dem Ausdruck oben kommt kein `x` mehr vor, also wird mit dem `x` nichts gemacht. Es wird einfach vergessen. Daher ist es auch egal, welchen Typ es hat. Probieren wir aus:

```
Prelude> (\x -> 'a') 1
'a'

Prelude> (\x -> 'a') 'b'
'a'

Prelude> (\x -> 'a') "Hallo"
'a'

Prelude> (\x -> 'a') (+)
'a'

Prelude> (\x -> 'a') (\x -> 'a')
'a'
```

An den letzten beiden Beispielen wird hoffentlich deutlich, daß die Sache mit „für alle Typen" ernst gemeint ist. Funktionen sind nichts besonderes in Haskell, sie sind wie alles andere auch.
Okay, nächstes Beispiel, zugegebenermaßen wiederum ziemlich trivial:

```
Prelude> :t (\x -> x)
forall t. t -> t
```

Hier haben wir die Identitätsfunktion geschrieben, die es auch vordefiniert unter dem Namen `id` gibt

```
Prelude> :t id
forall a. a -> a
```

Egal welchen Typ man hineinsteckt, man bekommt einen Wert desselben Typs wieder heraus. In Beispielen:

```
Prelude> (\x -> x) 1
1
it :: Integer

Prelude> (\x -> x) 'x'
'x'
it :: Char

Prelude> :t (\x -> x) (\x -> 'a')
forall t. t -> Char
```

Im letzten Fall müssen wir wieder nach dem Typ fragen; die Auswertung resultiert in einer Fehlermeldung (weil, wie wir schon festgestellt haben, funktionswertige Ergebnisse nicht ausgegeben werden können).
Als drittes Beispiel werden wir eine Funktion schreiben, die es ebenfalls vordefiniert unter dem Namen `const` gibt.

```
Prelude> :t const
forall a b. a -> b -> a
```

Wir sehen wieder: Eine Funktion mit zwei Argumenten, eins vom Typ `a` (egal welcher), eins vom Typ `b` (auch egal welcher). Das Resultat ist wieder vom ersten Typ. Ausprobieren:

```
Prelude> const 2 'a'
2

Prelude> const 'a' 2
'a'
```

Offenbar „vergißt“ die Funktion ihr zweites Argument und liefert das erste. Es ist also, nach der Spezifikation des ersten Arguments, die konstante Funktion (daher der Name), die immer dieses erste Argument zurückliefert.

```
Prelude> const (\x -> x) (\x -> 'a') 2
2
```

Was war das? Drei Argumente? Nun, Funktionsapplikation wird von links nach rechts ausgewertet, bis zum zweiten Argument wird `const` bedient. Dieses vergißt sein zweites Argument und liefert das erste, welches die Identitätsfunktion ist. Diese konsumiert die 2, macht aber nichts mit dieser.
Wie implementieren wir nun diese Funktion `const` selbst? Wir haben ja bereits eine Funktion, die konstant ist, geschrieben, nämlich `(\x -> 'a')`. Diese ist äquivalent zu der Funktion `const 'a'`. Um zu der gesuchten Funktion zu kommen, muß `'a'` durch ein weiteres Argument ersetzt werden:

```
Prelude> :t (\y -> (\x -> y))
forall t t1. t1 -> t -> t1
```

Haskell bietet eine kürzere Schreibweise dafür:

```
Prelude> :t (\y x -> y)
forall t t1. t1 -> t -> t1
```

Aufgabe 1

Verifiziere an Beispielen, daß wir tatsächlich die Funktion `const` reimplementiert haben.

Aufgabe 2

Schreibe eine Funktion, die von drei Argumenten das mittlere zurückliefert. Sie soll den Typ

```
forall t t1 t2. t -> t2 -> t1 -> t2
```

(oder einen äquivalenten Typ) haben.

Ein kleiner Hinweis ohne viel Theorie: Im GHCi (nicht im Hugs, leider) kann man selbst interaktiv Namen für Ausdrücke vergeben, so daß wir schreiben können:

```
Prelude> let const' = (\y x -> y)
const' :: forall t t1. t1 -> t -> t1
```

oder

```
Prelude> let one = 1
one :: Integer
```

Von der Definition an kann die Funktion unter diesem Namen verwendet werden.

Aufgabe 3

Schreibe eine Funktion `twice`, die eine gegebene Funktion zweimal hintereinander auf ihr Argument anwendet. Beispiel der Verwendung:

```
Prelude> :t twice
forall t. (t -> t) -> t -> t

Prelude> twice (\x -> x) 2
2

Prelude> twice (\x -> x + 1) 2
4

Prelude> twice (twice (\x -> x + 1)) 2
6
```

Richtig nützlich werden Funktionen in der Zusammenarbeit mit Listen und mit Rekursion. Mit diesen Themen beschäftigen wir uns daher in der nahen Zukunft.

Lernziele Tag 3

- ☐ „Currying“ ist keine Zauberei: Mehrargumentige Funktionen werden in Haskell meistens durch Funktionen repräsentiert, die ihrerseits Funktionen als Wertebereich haben.
- ☐ Partielle Applikation ist durch „Currying“ einfach möglich und eine gängige Methode.
- ☐ Funktionen können mittels Lambda-Abstraktion kreiert werden. Einer (oder mehrere) formale Parameter werden zwischen `\` und `->` eingeführt und können rechts vom `->` verwendet werden.
- ☐ Nicht nur beim Currying, sondern ganz generell können Funktionen in Haskell Argumente und Resultate von anderen Funktionen sein. Das ist ein wesentlicher Grund dafür, daß Haskell eine „funktionale“ Programmiersprache ist.