Tag 4

Listen

Im Vergleich zu den relativ komplizierten Funktionen wird es heute zunächst wieder einfacher. Es geht um Listen, eine allgegenwärtige Datenstruktur in Haskell.
Natürlich werden uns auch weiterhin Funktionen begegnen, aber da wir mit Hilfe von Listen eine schier unerschöpfliche Menge an Möglichkeiten hinzugewinnen, können wir viel mehr machen und hoffentlich auch an neuen Beispielen noch einiges der vergangenen Tage auffrischen und aufklären.
Listen in Haskell dienen zur Sammlung von mehreren Objekten, die alle denselben Typ haben. Anders formuliert: Zu *jedem* Typ `t` in Haskell gibt es auch einen Typ „Liste von `t`“, dieser wird von Haskell mit `[t]` bezeichnet. Daher nennt man den Listentyp auch einen „Typkonstruktor“: Er erschafft aus Datentypen neue Datentypen. In dieser Hinsicht ist er ähnlich dem Funktionspfeil `->`, der ebenfalls aus zwei schon bestehenden Datentypen den Typ der Funktionen zwischen diesen beiden Typen bildet.
Eine Liste selbst ist induktiv aufgebaut. Entweder ist sie leer, oder sie besteht aus einem Kopfelement und einer Restliste. Die leere Liste wird in Haskell mit `[]` bezeichnet:

```
Prelude> :t []
forall t. [t]
```

Wieder einmal sehen wir, daß Haskell versucht, keine unnötigen Spezialisierungen des Typs vorzunehmen. Da die leere Liste kein Element enthält, ist auch durch sie alleine noch nicht klar, welchen Typ ihre Elemente haben. Jeder beliebige Elementtyp ist möglich, darum wird über diesen quantifiziert. Das ändert sich, sobald wir ein Element vorne an die Liste anfügen. Das geschieht mit dem „cons“-Operator `:`:

```
Prelude> :t (:)
forall t. t -> [t] -> [t]

Prelude> :t ('x' : [])
[Char]
```

Auch der „cons“-Operator hat einen sogenannten „polymorphen“ Typ (der allquantifizierte Variablen enthält), denn auch er wirkt auf für alle Listen, nicht nur für die eines bestimmten Elementtyps. Er nimmt zwei Argumente, nämlich ein neues Kopfelement vom Elementtyp `t` sowie eine bereits bestehende Liste vom Typ `[t]`, und bildet daraus eine neue Liste, indem eben das neue Element vorne angehängt wird.
„Cons“ ist ein Operator, er kann infix geschrieben werden, aber (wie alle Operatoren) auch normal als Funktion in Präfix-Schreibweise verwendet werden, wenn man ihn in Klammern einschließt:

```
Prelude> (:) 2 (1 : [])
[2,1]
it :: [Integer]
```

Normalerweise verwenden wir die deutlichere Infix-Schreibweise. Noch ein Beispiel:

```
Prelude> 'x' : []
"x"
it :: [Char]
```

Wenn wir obigen Ausdruck auswerten, so erhalten wir als Ergebnis `"x"` in der String-Schreibweise. Dies verrät uns, daß in Haskell der Typ `String` ein Synonym ist für `[Char]`. In der Tat gibt es drei gleichbedeutende Schreibweisen für einen String:

```
Prelude> "Hallo"
"Hallo"
it :: [Char]

Prelude> ['H','a','l','l','o']
"Hallo"
it :: [Char]

Prelude> 'H' : 'a' : 'l' : 'l' : 'o' : []
"Hallo"
it :: Char
```

Die dritte Variante repräsentiert explizit den tatsächlichen, rekursiven Aufbau von Listen. An dieser Variante kann man sehen, daß „cons“ rechts-assoziativ ist. Der Ausdruck ist daher gleichwertig zu

```
Prelude> 'H' : ('a' : ('l' : ('l' : ('o' : []))))
```

Die mittlere Variante ist eine abkürzende Schreibweise, die für alle Listentypen zur Verfügung steht. Die erste Variante ist eine noch bequemere Schreibweise für Strings, also Listen mit Elementtyp `Char`.
Tatsächlich ist die mittlere Variante die bevorzugte Schreibweise für alle anderen Formen von Listen, wie man an der Ausgabe

```
Prelude> 1 : 2 : 3 : []
[1,2,3]
it :: [Integer]
```

sehen kann.
Wir haben jetzt auch schon genug Informationen, um grob, aber hoffentlich einleuchtend, erklären zu können, warum die direkte Eingabe von

```
Prelude> []
```

zu einer Fehlermeldung führt. (Zur Erinnerung: Oben haben wir nach dem Typ der leeren Liste gefragt, sie aber nicht ausgewertet.) Wieder einmal geht nicht das Auswerten, sondern das anschließende Ausgeben schief. Wir wissen, daß die leere Liste einen beliebigen Listentyp haben kann. Nun würde aber der leere String als `""` repräsentiert, während jede andere leere Liste als `[]`

dargestellt würde. Daher kann, ohne den Elementtyp zu kennen, die leere Liste nicht ausgegeben werden. Haskell erlaubt es aber, Typinformation explizit zu machen (auch wenn dies nur selten notwendig ist), so daß wir schreiben können

```
Prelude> [] :: [Char]
""
it :: [Char]

Prelude> [] :: [Integer]
[]
it :: [Integer]
```

Funktionen, die auf Listen arbeiten, gibt es in Hülle und Fülle bereits in der Prelude.

Aufgabe 1

Versuche, durch Erfragen des Typs und durch Ausprobieren die Wirkungsweise der folgenden Funktionen herauszufinden:

`sum`
`product`
`maximum`
`length`
`reverse`
`(++)` (ein Operator, der infix verwendet werden kann)
`head`
`tail`
`take`
`drop`

Wichtig ist, daß Listen zwar für jeden beliebigen Typ (also insbesondere auch Listen- und Funktionstypen) definiert sind, aber nie Elemente von unterschiedlichem Typ enthalten können. Die Eingabe

```
Prelude> ['c',id]
```

etwa (`id` war die Identitätsfunktion `(\x -> x)`) liefert einen Typfehler, der im GHCi so aussieht:

```
<interactive>:1:
    Couldn't match 'Char' against 'a -> a'
        Expected type: Char
        Inferred type: a -> a
    In the list element: id
    In the definition of 'it': ['c',id]
```

Wegen des ersten Elements wird angenommen, daß es sich um eine Liste von Elementen des Typs `Char` handelt. Das zweite Element wird aber als Funktion `a -> a` inferiert (die Quantifizierung

wird hier weggelassen, ist aber implizit hinzuzudenken). Diese beiden Typen können nicht in Einklang gebracht werden.
Eine sehr mächtige Funktion im Zusammenhang mit Listen ist die Funktion `map`:

```
Prelude> :t map
forall a b. (a -> b) -> [a] -> [b]
```

Diese Funktion hat zwei Argumente: Das erste ist eine Funktion von `a` nach `b`. Das zweite eine Liste von `a`. Das Resultat ist die Liste mit Elementen vom Typ `b`, die dadurch entsteht, daß die Funktion auf jedes Element der ursprünglichen Liste angewendet wird.

```
Prelude> map (\x -> x + 1) [1,2,3]
[2,3,4]
it :: [Integer]
```

Aufgabe 2

Reimplementiere die Funktion `length` unter Verwendung von `map`, `const` und `sum` (und Lambda-Abstraktion).

Lernziele Tag 4

- ☐ Listen sind Sammlungen von Elementen gleichen Typs. Listen sind entweder leer oder bestehen aus Kopfelement und Restliste.
- ☐ Strings sind nur Listen von einzelnen Zeichen.
- ☐ Mit `[]` und `(:)` lassen sich Listen schrittweise aufbauen.
- ☐ Es gibt mächtige Funktionen wie zum Beispiel `map`, um auf Listen zu arbeiten.