Tag 2

Mehr Typen

Zunächst wieder

```
Prelude> :set +t
```

um Typinformationen bei der Auswertung von Ausdrücken zu bekommen.
Am ersten Tag haben wir hauptsächlich mit ganzzahligen Werten (`Integer`) und Fließkommazahlen (`Double`) gerechnet. Wir haben auch schon Beispiele für Zeichen und Zeichenketten gesehen, zur Erinnerung:

```
Prelude> 'c'
Prelude> "Hallo"
```

Es gibt aber viel mehr Datentypen: In Haskell hat jeder Ausdruck einen Typ, und jeder Teilausdruck ebenfalls. Wir haben bereits mit einer ganzen Reihe von Funktionen und Operatoren gearbeitet, wie etwa `sin` oder `+`. Was haben sie für einen Typ? Probieren wir einmal

```
Prelude> sin
```

Wir bekommen eine Fehlermeldung. GHCi sagt uns

```
No instance for (Show (a -> a))
```

und noch viel mehr, während Hugs uns mit

```
ERROR - Cannot find "show" function for:
*** Expression : sin
*** Of type    : Double -> Double
```

beglückt. Um den Fehler zu erklären, werden wir versuchen zu erkennen, warum ein Fehler auftritt.
Wenn man einen Haskell-Ausdruck am Interpreter-Prompt eingibt, dann wird der Ausdruck ausgewertet und das Ergebnis angezeigt. Bei dieser Ausgabe auf dem Bildschirm bedient sich der Interpreter einer speziellen Funktion `show`, über die wir später noch mehr lernen werden.
Wir haben eine Funktion, nämlich `sin` eingegeben. Da die Funktion kein Argument bekommen hat, kann sie nicht weiter ausgewertet werden, daher ist die Auswertung der Funktion die Funktion selbst. An der Auswertung scheitert es hier also *nicht*! Das Problem tritt bei der Ausgabe auf! In Haskell gibt es keine vordefinierte Möglichkeit, eine Funktion auszugeben. Genau diesen Umstand versucht die kryptische Fehlermeldung von oben in Worte zu fassen.
Glücklicherweise gibt es aber die Möglichkeit, sich vom Interpreter den Typ von Ausdrücken anzeigen zu lassen, ohne die Auswertung und Ausgabe des Ausdruckes zu veranlassen. Dies

geschieht mittels `:t` (zur Erinnerung: diese Anweisungen, die mit einem Doppelpunkt beginnen, sind allesamt Kommandos an den Interpreter und haben mit der Sprache Haskell eigentlich nichts zu tun), also

```
Prelude> :t 'c'
Char
Prelude> :t sin
forall a. (Floating a) => a -> a
```

(Beim Hugs fehlt das `forall a.` in der Ausgabe.) Das scheint ziemlich kompliziert, ist aber zu lesen wie eine logische Formel, wobei `Floating` ein Prädikat oder eine einstellige Relation ist. Also: Für alle (Typen) `a`, für die `Floating a` gilt, haben wir eine Funktion `a -> a`. Insbesondere gilt `Floating Double`, daher läßt sich der Type vereinfachen zu `Double -> Double` und wir können die Funktion auf Werte vom Typ `Double` anwenden:

```
Prelude> 0.5
0.5
it :: Double
Prelude> sin 0.5
0.479425538604203
it :: Double
```

Ein ähnliches Bild liefert die Funktion für den Absolutbetrag `abs`:

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

Bis auf das Prädikat gibt es keinen Unterschied. `Num` ist ein Prädikat, das von mehr Typen erfüllt wird als `Floating`, unter anderem sowohl von `Double` als auch von `Integer`. Daher

```
Prelude> abs (-2)
2
it :: Integer
Prelude> abs 0.5
0.5
it :: Double
```

Aber

```
Prelude> abs 'c'
```

produziert eine Fehlermeldung, im GHCi wie folgt:

```
<interactive>:1:
    No instance for (Num Char)
    arising from use of ‘abs’ at <interactive>:1
    In the definition of ‘it’: abs ‘c’
```

Die relevante der vier Zeilen ist die zweite: `No instance for (Num Char)` bedeutet nichts weiter, als daß `Char` das Prädikat "Num" nicht erfüllt. Wir werden viel später genauer beleuchten, was es mit Prädikaten auf sich hat und welche Typen welche Prädikate erfüllen. Bislang genügt es, sich zu merken:

`Num` (numerische Werte) wird von `Integer` und `Double` erfüllt
`Floating` (Fließkommazahlen) wird von `Double` erfüllt

Schon an Tag `??` haben wir eine sehr ähnliche Fehlermeldung erzeugt, nämlich mit

```
Prelude> 2+'c'
```

Auch `+` ist eine Haskell-Funktion und verhält sich fast genauso wie `abs` und `sin`, kann aber infix verwendet werden, also zwischen zwei Argumenten. Die Infix-Schreibweise ist in Haskell nur sogenannter syntaktischer Zucker, also eine Erleichterung, um die Programme schöner aussehen zu lassen. Jeder Operator ist eine ganz normale Funktion, wenn man ihn in runde Klammern einschließt:

```
Prelude> (+) 2 3
```

ist gleichbedeutend mit

```
Prelude> 2 + 3
```

Aufgabe 1

Offenbar werden auch bei Funktionen mit mehreren Argumenten keine Klammern zur Auszeichnung und Begrenzung der Argumente verwendet. Die Schreibweise

```
Prelude> (+) (2,3)
```

etwa führt zu einer sehr komplizierten Fehlermeldung. (Es handelt sich dabei aber, wie wir später sehen werden, durchaus um einen syntaktisch gültigen Haskell-Ausdruck.) Was für Vorteile könnte die erste Schreibweise haben? Probiere die folgenden Eingaben aus:

```
Prelude> :t (+)
Prelude> :t (+) 2
Prelude> :t ((+) 2) 3
Prelude> ((+) 2) 3
Prelude> :t (+2)
Prelude> (+2) 3
Prelude> :t (/)
Prelude> :t (/2.5)
Prelude> :t (2.5/)
Prelude> (/2.5) 7
Prelude> (2.5/) 7
```

Lernziele Tag 2

- ☐ Typinformation ohne Auswertung und Ausgabe erhält man mit `:t`.
- ☐ Funktionstypen enthalten das Symbol `->`.
- ☐ Typen haben Ähnlichkeit mit logischen Formeln: Sie können Prädikate, Quantifizierungen und Variablen enthalten.