

# Erste Funktionen

Sven Eric Panitz

16. März 2021

## Inhaltsverzeichnis

<b>1 Funktionen</b>	<b>1</b>
1.1 Einstellige Funktionen . . . . .	2
1.2 Nullstellige Funktionen . . . . .	5
1.3 Mehrstellige Funktionen . . . . .	5
1.4 Nicht strikte Auswertung . . . . .	7
1.5 Funktionen höherer Ordnung . . . . .	9
1.6 $\lambda$ -Ausdrücke . . . . .	11
1.7 Operatoren . . . . .	11
1.8 Fallunterscheidungen in Funktionen . . . . .	13
1.8.1 Guards . . . . .	14
1.8.2 Fallunterscheidungen durch if-Ausdrücke . . . . .	14
1.8.3 Fallunterscheidungen mit Pattern Matching . . . . .	15
1.9 Lokale Funktionsdefinitionen . . . . .	16
1.9.1 Mit der where-Klausel . . . . .	16
1.9.2 Mit let-Ausdrücken . . . . .	16
1.10 Tupeltypen . . . . .	17
<b>2 Aufgaben</b>	<b>18</b>
<b>3 Lernzuwachs</b>	<b>24</b>

## 1 Funktionen

In dieser Aufgabe werden wir erste Schritte in Haskell machen und einüben, wie in Haskell Funktionen definiert und aufgerufen werden.

Es wird empfohlen diese Aufgabe mit dem Interpreter `ghci` durchzulesen und dort eigene Aufrufe der vorgestellten Funktionen zu machen.

Das Haskellmodul dieser Aufgabe ist im *literate programming* Stil geschrieben, d.h. es ist eigentlich ein  $\text{\LaTeX}$ -Dokument, in dem Haskell Code markiert ist. Die Zeilen, in denen Haskell steht beginnen mit einem `>` Zeichen. Alle anderen Zeilen sind für den Haskell-Compiler nur Kommentar. Man nennt diese Art auch umgekehrte Kommentierung. Nicht Kommentare sind markiert sondern der Programmtext. Dieses spiegelt auch die Philosophie der Haskell-Gemeinschaft wider. Es ist wichtiger über Programme und Algorithmen nachzudenken und die Programme sind nur kleine Teile, wie die Formeln in einem mathematischen Lehrbuch.

Während Haskell-Quelltext mit der Dateiendung `hs` gespeichert wird, haben Quelltextdateien in Literate-Haskell die Endung `lhs`.

## 1.1 Einstellige Funktionen

Wir definieren für unsere ersten Gehversuche in Haskell ein eigenes kleines Modul:

Haskell: FirstFunctions

```
> module FirstFunctions where
```

Wir werden in einigen Aufgabe mit rationalen Zahlen arbeiten und importieren daher schon einmal das entsprechende Modul.

Haskell: FirstFunctions

```
> import Data.Ratio
```

Eine Funktion wird in Haskell in Form einer Gleichung definiert. Nach dem Funktionsbezeichner folgen die Argumente. Dann kommt ein Gleichheitszeichen und die Funktionsdefinition, die aus genau einem Ausdruck besteht. Beispielsweise kann die Funktion zum Quadrieren einer Zahl wie folgt definiert werden.

Haskell: FirstFunctions

```
> square x = x * x
```

Wir können das Modul in dem Interpreter des `ghc` öffnen und ein Beispielaufrufe machen.

Funktionen werden aufgerufen, indem nach dem Funktionsnamen das Argument schreibt. Es braucht keine Klammern.

## Shell

```
00001SimpleFunctions$ ghci solution/FirstFunction.lhs
GHCi, version 8.6.5: http://www.haskell.org/ghc/ :? for help
[1 of 1] Compiling FirstFunctions ( solution/FirstFunction.lhs, interpreted )
Ok, one module loaded.
*FirstFunctions> square 5
25
```

Eine kleine Schwierigkeit sind negative Zahlenlitterale. Diese sind zu klammern

## Shell

```
*FirstFunctions> square (-3)
9
```

Wer mutig ist, wird feststellen, dass die Funktion offensichtlich auch für beliebig große Zahlen funktioniert:

## Shell

```
*FirstFunctions> square 1243344332345324332132132245856756756544376
1545905128775240326235593100283361177394055131829160954169949825892679167904857229376
```

Auf was für einen Zahlentyp wird hier gerechnet? Es kann offensichtlich nicht mehr ein 64-Bit Int sein. Wir haben bei unserer Funktionsdefinition keine Typdeklaration mitgegeben. Trotzdem ist Haskell statisch getypt, also hat jede Funktion eine vom Compiler überprüfte Typisierung. Im Interpreter können wir mit dem Meta-Befehl `:t` für einen Ausdruck den Typ anzeigen lassen. Der Compiler hat eine Typinferenz, errechnet also vollständig eigenständig die Typisierung einer Funktion, ohne dass diese im Programmtext notiert sein muss. Man nennt dieses *Typinferenz*.

## Shell

```
*FirstFunctions> :t square
square :: Num a => a -> a
*FirstFunctions>
```

Der Typ der Funktion `square` wird angezeigt. Dieser ist gar nicht so einfach zu lesen, weil hier schon eine Besonderheit des Haskell-Typsysteams zutage tritt, die Typklassen. Die Zeile `square :: Num a => a -> a` ist zu lesen als:

Der Typ der Funktion `square` ist polymorph für alle Typen, die zur Typklasse `Num` gehören. Für einen variabel gehaltenen Typ `a`, der zur Typklasse `Num` gehört, wird für ein Argument vom Typ `a` ein Ergebnistyp `a` errechnet. Das Symbol `->` zeigt an, dass es sich um eine Funktion handelt. Vor dem Pfeil steht der Argumenttyp, nach dem Pfeil der Ergebnistyp der Funktion.

Die Typklasse `Num` repräsentiert im wesentlichen Typen für ganze Zahlen. In Haskell gibt es dazu den Typ `Int`, der einen Maximal- und einen Minimalwert hat und der Typ `Integer`, der beliebig große Zahlenbeträge zulässt.

Um ein wenig mehr ein Gefühl für die Typisierung zu geben, sei eine zweite Funktion definiert,

#### Haskell: FirstFunctions

```
> i x = x
```

Die Funktion `i` ist die Identitätsfunktion, die das Argument direkt wieder zurück gibt. Betrachten wir einmal den Typen für diese Funktion im Interpreter.

#### Shell

```
*FirstFunctions> :t i  
i :: p -> p
```

Auch diese Funktion ist polymorph. Sie kann für einen beliebig aber festen Typ `p` so typisiert werden, dass für ein Argument des Typ `p` das Ergebnis auch wieder vom Typ `p` ist.

Somit lässt sich die Funktion `i` auf Argumente unterschiedlicher Typen anwenden.

Einmal auf eine Zahl:

#### Shell

```
*FirstFunctions> i 42  
42
```

Aber ebenso auf ein String.

#### Shell

```
*FirstFunctions> i "hallo"  
"hallo"
```

Und die Ergebnistypen sind dann auch korrekt inferiert. Bei der Anwendung auf eine Zahl, ist das Ergebnis ein Zahlentyp:

#### Shell

```
*FirstFunctions> :t i 42  
i 42 :: Num p => p
```

Bei der Anwendung auf einen String ist auch das Ergebnis ein String, der in Haskell als Liste von Zeichen notiert wird: `[Char]`.

#### Shell

```
*FirstFunctions> :t i "hallo"  
i "hallo" :: [Char]
```

## 1.2 Nullstellige Funktionen

Bisher haben wir Funktionen mit einem Argument betrachtet. Man kann auch Funktionen ohne Argumente definieren. Solche haben einen konstanten Wert als Ergebnis.

Haskell: FirstFunctions

```
> x42 = (17 + 4) * 2
```

Diesen Wert braucht Haskell nur einmal auszuwerten und kann ihn dann bei weiteren Aufruf der konstanten Funktion direkt zurück geben. Das liegt daran, dass es keine Seiteneffekte gibt, die bei der Auswertung der Funktionsdefinition ausgeführt werden.

Man spricht bei konstanten Funktionen auch von einer *constant applicative form*.

Man sollte die konstanten Funktion nicht mit Variablen in einer imperativen Programmiersprache gleichsetzen. Im Gegensatz dazu ist es eine Funktionsdefinition, die erst ausgeführt wird, wenn in einer Berechnung ihr Ergebnis benötigt wird.

Den Unterschied zu Variablen in imperativen Sprachen kann man gut an folgender Funktion sehen:

Wir können nämlich auch Funktionen definieren, deren Auswertung nicht terminieren. Eine simple solche konstante Funktion ist die folgende Funktion, die direkt in ihrer Definition sich selbst wieder aufruft

Haskell: FirstFunctions

```
> bot = bot
```

Eine entsprechende Definition in Java wäre die Folgende:

Java: Bot

```
class Bot{
    static <A> A bot(){return bot();}
}
```

## 1.3 Mehrstellige Funktionen

Auf naheliegende Weise lassen sich auch Funktionen mit mehr als einem Argument definieren. Zum Beispiel eine Funktion, die zwei Zahlenargumente addiert und das Ergebnis verdoppelt:

Haskell: FirstFunctions

```
> addTwice x y = 2 * (x + y)
```

Eine solche Funktion kann aufgerufen werden, indem die beiden Argumente direkt nach dem Funktionsnamen folgen. Es braucht hierfür keine Klammern oder Trennungen der Argumente mit einem Komma.

Shell

```
*FirstFunctions> addTwice 17 4  
42
```

Der inferierte Typ dieser Funktion ist interessant. Es überrascht nicht, dass sie wieder für numerische Typen definiert ist:

Shell

```
*FirstFunctions> :t addTwice  
addTwice :: Num a => a -> a -> a
```

Der eigentliche Typ der Funktion ist für einen numerischen Typen  $a$ :  $a \rightarrow a \rightarrow a$ . Das Pfeilsymbol  $\rightarrow$  drückt einen Funktionstypen aus. Es tauchen in der Typisierung zwei Funktionstypen auf. Die Funktionstypen sind als rechtsassoziativ zu lesen. Der obige Typ ist also ausgeschrieben mit Klammerung:  $a \rightarrow (a \rightarrow a)$ . Für die erste Funktion ist das Argument vom Typ  $a$  und das Ergebnis vom Typ  $a \rightarrow a$ , also von einer weiteren Funktion.

Das wird noch einmal deutlich, wenn man die Typinferenz nach den Typ fragt, den man erhält, wenn man die Funktion `addTwice` nur auf ein Argument anwendet:

Shell

```
*FirstFunctions> :t addTwice 17  
addTwice 17 :: Num a => a -> a
```

Der Typ von `addTwice` ist eine Funktion, die quasi das zweite Argument erwartet, um dann ein Endergebnis zu berechnen.

In Haskell gibt es streng genommen keine mehrstelligen Funktionen, sondern nur einstellige Funktionen. Eine mehrstellige Funktion wird als einstellige Funktion angesehen, deren Ergebnis wieder eine Funktion ist, die dann das zweite Argument erwartet und so weiter.

Man kann dieses Prinzip auch durch den Aufruf der Funktion mit einer expliziten Klammerung deutlich machen. Komplett geklammert sieht der Ausdruck wie folgt aus:

Shell

```
*FirstFunctions> ((addTwice 17) 4)  
42
```

Dieses Prinzip wird als *Currying* bezeichnet, nach dem Mathematiker Haskell B. Curry (\* 12. September 1900 in Millis, Massachusetts, USA; † 1. September 1982 in State College, Pennsylvania, USA).

Und nun weiß der Leser auch, woher die Programmiersprache Haskell ihren Namen hat. Dabei hätte man dieses Prinzip korrekter Weise wohl besser als *Schönfinkeling* bezeichnet, denn es findet sich bereits in den Arbeiten von Moses Isajewitsch Schönfinkel (russisch Моисей Исаевич Шейнфинкель, wiss. Transliteration Moisej Isaewič Šejnfinkel'; \* 9. September 1888 in Jekaterinoslaw; † 1942 in Moskau) beschrieben.

Tatsächlich bilden die Arbeiten von Curry und Schönfinkel zur kombinatorischen Logik als Berechenbarkeitsmodell eine der Grundlagen der funktionalen Programmierung.

In ihren Arbeiten spielt auch folgende Funktion eine wichtige Rolle.

#### Haskell: FirstFunctions

```
> k x _ = x
```

Diese Funktion hat zwei Argumente. Sie ignoriert das zweite Argument und hat das erste Argument als Ergebnis. Man sagt auch, es ist eine Projektion auf das erste Argument. Da das zweite Argument in der Funktionsdefinition gar nicht verwendet wird, verzichten wir in Haskell darauf, dafür einen Namen zu erfinden und bedienen uns des reservierten Unterstrichs `_`, der anzeigt, dass hier kein eigener Bezeichner notwendig ist.

Werfen wir auch einen Blick auf den Typ der Funktion `k`:

#### Shell

```
*FirstFunctions> :t k  
k :: p1 -> p2 -> p1
```

Es ist nun also streng genommen der Typ: angewendet auf ein Argument des Typs `p1` erhält man als Ergebnis eine Funktion, die ein Argument eines beliebigen Typs `p2` hat und als Ergebnis ein `p1` liefert.

## 1.4 Nicht strikte Auswertung

Der wahrscheinlich größte Unterschied in Bezug auf Funktionen in Haskell zu fast allen anderen Programmiersprachen<sup>1</sup> ist das Auswertungsmodell, das weitgreifende Konsequenzen hat.

Betrachten wir hierzu die folgenden zwei Funktionsdefinitionen:

#### Haskell: FirstFunctions

```
> doppel x = x + x  
> quadrat x = x * x
```

---

<sup>1</sup>Fast alle schließt die Sprachen Clean, Miranda und Natural Expert Language aus.

Wie wird der Ausdruck `quadrat (doppel 5)` zu einem Ergebnis ausgewertet? Wir können das schrittweise machen, durch Einsetzen der Funktionsdefinitionen. In herkömmlichen Programmiersprachen werden erst die Argumente ausgewertet und dann das Ergebnis der Argumente der aufrufenden Funktion übergeben. Für unser Beispiel kommt es zu folgenden Auswertungsschritten:

```
quadrat (doppel 5)
→ quadrat (5+5)
→ quadrat 10
→ 10 * 10
→ 100
```

Haskell geht aber anders vor. Statt von innen, erst die Argumente und dann nach außen die aufrufende Funktion durch die Definition zu ersetzen, wird erst der Aufruf durch die Funktionsdefinition ersetzt. Es kommt in unserem Beispiel zu folgenden Auswertungsschritten:

```
quadrat (doppel 5)
→ (doppel 5) * (doppel 5)
→ (5 + 5) * (doppel 5)
→ 10 * (doppel 5)
→ 10 * (5 + 5)
→ 10 * 10
→ 100
```

Das Ergebnis beider Auswertung ist dasselbe. Die herkömmliche Auswertung, in der erst die Argumente zu ihren Ergebnis ausgewertet werden wird als *applikative Auswertungsreihenfolge* bezeichnet. Man spricht auch von einer *strikten* Auswertung.

Die Auswertungsreihenfolge, in der erst die äußere Funktionsdefinition ersetzt wird, nennt man *normale Auswertungsreihenfolge*. Hier spricht man von *nicht-strikter* Auswertung.

Ein Ergebnis aus der Theorie garantiert bei reinen funktionalen Sprachen, dass beide Auswertungsreihenfolgen, wenn sie zu einem Ergebnis führen, das gleiche Ergebnis errechnen.

Im obigen Beispiel sieht man auch, dass die nicht-strikte Auswertung ein Problem hat. Sie braucht mitunter mehr Auswertungsschritte, weil sie durch das Einsetzen der Definition der Funktion `quadrat` das Argument zweimal in der Definition hat. Da das Argument noch nicht ausgewertet ist, wird damit die Arbeit es auszuwerten verdoppelt.

Dieses lässt sich verhindern, indem man nicht Terme auswertet sondern Graphen, in denen es also Sharing gibt. So wird nicht das Argument doppelt notiert, sondern nur einmal und zweimal darauf referenziert.

Eine nicht-strikte Auswertungsreihenfolge mit Sharing wird als *lazy evaluation* bezeichnet. Haskell hat das Prinzip der *lazy evaluation*.



Die Auswertung eines Haskellausdrucks besteht aus dem Ersetzen einer Graphstruktur entsprechend der Funktionsdefinitionen. Daher heißt die abstrakte Maschine zum Auswerten von Haskellprogrammen auch *g-machine*, wobei das ›g‹ für *graph* steht.

Was hat man durch die *lazy evaluation* gewonnen? Hierzu betrachte man den folgenden simplen Ausdruck:

```
k 42 bot.
```

Wenn wir diesen versuchen strikt auszuwerten, terminiert die Auswertung nicht:

```
k 42 bot
→ k 42 bot
→ k 42 bot
→ k 42 bot
→ k 42 bot
→ ...
```

Es wird immer versucht erst das Argument `bot` auszuwerten. Die Auswertung terminiert aber nicht, und so kann auch nie der Gesamtausdruck `k 42 bot` zu einem Ergebnis ausgewertet werden.

Anders sieht es in der normal Auswertungsreihenfolge aus:

```
k 42 bot
→ 42
```

Hier wird in einem Schritt als erstes die Definition der Funktion `k` eingesetzt. Dadurch verschwindet das zweite Argument komplett. Es bleibt das erste Argument stehen und wir erhalten ein Ergebnis.

Jetzt erkennt man auch, warum diese Auswertung als *faul* bezeichnet wird. Es werden die Argumente erst ausgewertet, wenn es anders gar nicht mehr geht. Es wird nur das nötigste ausgewertet.

Dieses eröffnet in Haskell ein Reihe von geschickten Programmierungen. Man kann mit unendlichen Datenstrukturen arbeiten und man erhält bestimmte Programmierpattern wie Backtracking oder das Producer-Consumer-Pattern automatisch.

In strikten Sprachen wie C oder Java gibt es drei eingebaute Operatoren, die auch nicht-strikt ausgewertet werden. Diese sind die logischen Operatoren `&&` und `||` sowie der Bedingungsoperator mit `?:`.

In strikten Sprachen kann man nicht sinnvoll eigene Funktionen oder Operatoren zur Fallunterscheidung schreiben. Es werden dann ja beim Aufruf der Funktion trotzdem alle Alternativen der Fallunterscheidung, die als Argumente übergeben werden ausgewertet. In Haskell ist das möglich, wie wir in Kürze zeigen.

## 1.5 Funktionen höherer Ordnung

Funktionen, deren Argumente Funktionen sind, werden als Funktionen höherer Ordnung bezeichnet.

Eine einfache solche Funktion höherer Ordnung ist die klassische Funktionsnacheinanderausführung.

Haskell: FirstFunctions

```
> nach f g x = f (g x)
```

Betrachten wir für diese Funktion einmal die Typisierung.

Shell

```
*FirstFunctions> :t nach
nach :: (t1 -> t2) -> (t3 -> t1) -> t3 -> t2
```

Das Argument `g` sei eine Funktion, die ein Argument vom Typ `t3` hat und als Ergebnis den Typ `t1`. Das erste Argument ist eine Funktion mit Argumenttyp `t1` und Ergebnistyp `t2`.

Wir können jetzt eine Funktionskomposition von zwei Funktionen definieren. Zum Beispiel die im letzten Abschnitt bereits betrachtete Komposition, in der erst das Argument verdoppelt und anschließend die Quadrierung aufgerufen wird.

Haskell: FirstFunctions

```
> qd = nach quadrat doppel
```

Die so neu definierte Funktion `qd` lässt sich nun auf ein Argument anwenden.

Shell

```
*FirstFunctions> qd 5
100
```

Interessant ist, dass in der Definition von `qd` das eigentliche Argument komplett ignoriert wird. Es sieht so aus, als würde die Funktion `nach` nicht mit genügend Argumenten aufgerufen. Man hätte die Funktion auch mit sichtbaren Argument definieren können:

Haskell: FirstFunctions

```
> qd2 x = nach quadrat doppel x
```

Diese Definition mag sichtbar machen, dass `qd2` eine Funktion ist, die ein Argument erwartet, aber es ist in Haskell unnötig, hier dieses Argument sichtbar zu machen. Grund dafür ist das Currying. Das Ergebnis der Funktion `nach` auf zwei Argumente angewendet ist eine Funktion, die ein weiteres Argument erwartet.

In der Kombinatorlogik gibt es neben den Funktionen `i` und `k` noch eine weitere fundamentale Funktion, die Funktion `s`, die in Haskell wie folgt definiert werden kann:

#### Haskell: FirstFunctions

```
> s x y z = x z (y z)
```

Der Typ dieser Funktion ist schon recht komplex, denn hier sind auch die Argumente `x` und `z` beide wieder von einem Funktionstypen.

#### Shell

```
*FirstFunctions> :t s  
s :: (t1 -> t2 -> t3) -> (t1 -> t2) -> t1 -> t3
```

## 1.6 $\lambda$ -Ausdrücke

Eine Funktion kann überall direkt definiert werden, indem man einen  $\lambda$ -Ausdruck verwendet. Neben der Kombinatorlogik von Schönfinkel und Curry sind der  $\lambda$ -Kalkül von Alonzo Church (\* 14. Juni 1903 in Washington, D.C.; † 11. August 1995 in Hudson, Ohio) die zweite große theoretische Grundlage der funktionalen Programmierung. Aus diesem leitet sich die Notation her, mit der in Haskell anonyme, also namenlose Funktionen überall im Programm geschrieben werden können.

Statt des Buchstabens  $\lambda$  wird dabei das Zeichen `\` verwendet. Die Funktionsdefinition wird von der Argumenten durch einen stilisierten Pfeil `->` getrennt. Letzteres kennt man ja auch seit Java 8 in Java.

Statt auf die beiden globalen Funktionen `quadrat` und `doppel` zurück zu greifen, können diese auch direkt als  $\lambda$ -Ausdruck notiert werden.

#### Haskell: FirstFunctions

```
> qd3 = nach (\x->x*x) (\x->x+x)
```

Man kann auch soweit gehen, alle globalen Funktionen als nullstellige Funktionen zu definieren, deren Definition dann ein  $\lambda$ -Ausdruck ist.

#### Haskell: FirstFunctions

```
> qd4 = \x -> nach quadrat doppel x
```

## 1.7 Operatoren

Bisher waren alle unsere Funktionen, die wir definiert haben in Präfixnotation zu verwenden. Wenn eine Funktion zwei Argumente hat, kann man diese in Haskell auch in Infixnotation verwenden. Hierzu ist der Funktionsname in rückwertigen einfachen Anführungszeichen ``` zu klammern. So lässt die die Funktion `nach` auch infix anwenden.

#### Haskell: FirstFunctions

```
> qd5 = quadrat `nach` doppel
```

Wir haben als Infixoperatoren schon die üblichen arithmetischen Operatoren kennen gelernt. In Haskell können wir aber auch beliebige eigene Operatoren definieren.

Statt eine Funktion `nach` können wir auch einen Infixoperator `-<-` definieren.

#### Haskell: FirstFunctions

```
> f -<- g = \x -> f (g x)
```

Dann wäre eine weitere Definition der Funktion, die nach der Verdopplung quadriert, wie folgt zu notieren.

#### Haskell: FirstFunctions

```
> qd6 = quadrat-<-doppel
```

Unser eigener Operator `-<-` ist die Funktionskomposition zweier Funktionen, die in der Mathematik mit einem Kringel geschrieben wird:  $f \circ g$ .

In der Haskellstandardbibliothek (das sogenannte *Prelude*) ist in Anlehnung an diese Notation unsere Funktion `nach` als der Operator mit dem Symbol `.` definiert:

#### Haskell: Prelude

```
(.)    :: (b -> c) -> (a -> b) -> a -> c  
(.) f g = \x -> f (g x)
```

Mit diesem Operator lassen sich Funktionskompositionen wie in der Mathematik notieren:

#### Haskell: FirstFunctions

```
> qd7 = quadrat.doppel
```

Ein weiterer Operator im Haskell-Prelude erlaubt es Klammern zu sparen. Es ist der Operator `$`.

Er hat die folgende Definition:

#### Haskell: Prelude

```
f $ x = f x
```

Dieses ist ein echter Taschenspielertrick. Es ermöglicht nämlich bei einem Ausdruck der Form: `f (g (h x))` auf die Klammern zu verzichten, die bei weiteren Funktions-

anwendungen und verketteten Funktionsanwendungen sehr umfangreich werden können. Stattdessen lässt sich schreiben: `f$g$h x`.

Oder für unsere kleine Beispielfunktion:

Haskell: FirstFunctions

```
> qd8 x = quadrat$doppel x
```

Der Operator `$` der Funktionsanwendung wird gerne von Haskellprogrammierern verwendet, so dass er manchmal schon wie ein eingebautes syntaktisches Feature wirkt.

Für unsere eigenen Operatoren ermöglicht Haskell auch, dass eine Operatorpräzedenz definiert werden kann. Zusätzlich kann auch definiert werden, ob ein Operator linksassoziativ oder rechtsassoziativ implizit geklammert ist.

Hierzu gehören die Deklarationen mit dem Schlüsselwort `infix`. Die beiden eben gesehenen Operatoren haben im Prelude zusätzlich folgende deklarationen.

Haskell: Prelude

```
infixr 9 .  
infixr 0 $
```

Das `r` nach dem Wort `infix` zeigt an, dass beide rechtsassoziativ geklammert sind. Die Ziffer zeigt an, wie stark ihre Bindung ist. 9 bindet am stärksten und 0 am schwächsten.

So sind die entsprechenden Deklarationen für die Operatoren `+` und `*`:

Haskell: Prelude

```
infixl 6 +  
infixl 7 *
```

Sie sind beide linksassoziativ und die Multiplikation bindet stärker als die Addition.

So wie man beliebige Funktion auch infix verwenden kann, lassen sich auch Operatoren präfix anwenden. Hierzu sind die Operatorsymbole in Klammern zu setzen.

Man kann beispielsweise schreiben:

Shell

```
*FirstFunctions> (*) ((+) 17 4) 2  
42
```

## 1.8 Fallunterscheidungen in Funktionen

Eine wichtige Ausdrucksmöglichkeit in jeder Programmiersprache sind Fallunterscheidungen, nach den Wert eines bestimmten Ausdrucks. Am häufigsten finden sich Fallun-

terscheidungen nach bool'schen Werten. Auch Haskell bietet natürlich die Möglichkeit nach Fällen zu unterscheiden.

### 1.8.1 Guards

Die gebräuchlichste Art von Fallunterscheidungen in Haskell sind die sogenannten *guards*. Diese werden durch einen vertikalen Strich eingeleitet. Ein bool'scher Ausdruck folgt dem Strich und anschließend der Ergebnisausdruck, falls der bool'sche Ausdruck zu *True* ausgewertet hat.

Die *guards* werden von oben nach unten durchgetestet, bis einer zu *True* auswertet. Dessen Definition wird dann für das Ergebnis genommen.

Ein einfaches Beispiel, dass den Betrag einer Zahl berechnet, sieht wie folgt aus.

Haskell: FirstFunctions

```
> absolute1 x
> |x < 0 = (-1)*x
> |otherwise = x
```

Dabei steht *otherwise* für einen Ausdruck, der immer *True* ist. Tatsächlich ist *otherwise* gar nicht als Schlüsselwort definiert, sondern eine nullstellige Funktion, die *True* als Ergebnis hat.

Mit *guards* können nicht nur zwei unterschiedliche Fälle, sondern beliebig viele Fälle behandelt werden.

Man betrachte die wie folgt definierte Signum-Funktion:

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

Diese lässt sich mit *guards* wie folgt definieren.

Haskell: FirstFunctions

```
> signum x
> |x > 0 = 1
> |x < 0 = (-1)
> |otherwise = 0
```

### 1.8.2 Fallunterscheidungen durch if-Ausdrücke

In der Praxis selten verwendet, kennt Haskell auch Bedingungsausdrücke mit *if*. Diese entsprechen den *?:-*-Ausdrücken in Java oder C, haben also immer eine positive und eine

negative Alternative, je nachdem, wie der Test ausfällt. Es gibt also immer ein `else`. Obige mit *guards* realisierte Funktion `absolute` lässt sich also auch mit `if` formulieren:

Haskell: FirstFunctions

```
> absolute2 x = if x < 0 then (-1)*x else x
```

In Haskell werden keine Klammern zum strukturieren der Bedingung verwendet, sondern wurde das Schlüsselwort `then` eingeführt.

Wegen der lazy Auswertungsstrategie in Haskell, lassen sich auch Funktionen zur Fallunterscheidung schreiben:

Haskell: FirstFunctions

```
> wenn p a1 a2 = if p then a1 else a2
```

Da die Argumente erst ausgewertet werden, wenn sie für das Ergebnis des Ausdrucks gebraucht werden, können der Funktion `wenn` auch nichtterminierende Argumente übergeben werden, und trotzdem kommt es zu einem Ergebnis:

Shell

```
*FirstFunctions> wenn True 42 bot
42
*FirstFunctions> wenn False bot 42
42
*FirstFunctions> wenn True bot 42
```

Eine solche Funktion lässt sich in einer strikt ausgewerteten Sprache nicht schreiben.

### 1.8.3 Fallunterscheidungen mit Pattern Matching

Ein häufig verwendetes Programmierprinzip in Haskell sind Pattern. Zumeist werden diese auf strukturierten Daten angewendet. Es gibt aber auch Pattern für Zahlenwerte. Eine Funktionsdefinition kann so durch mehrere direkt aufeinander folgende Gleichungen formuliert werden. Der Parameter kann dann durch ein Pattern unterschieden werden.

Als kleines Beispiel hierzu eine kleine Funktion, die für Zahlen einen String erzeugt.

### Haskell: FirstFunctions

```
> zahlenworte 0 = "null"
> zahlenworte 1 = "eins"
> zahlenworte 2 = "zwei"
> zahlenworte 3 = "drei"
> zahlenworte 4 = "vier"
> zahlenworte 5 = "fünf"
> zahlenworte x
> |x>0 = "viele"
> |otherwise = "negativ"
```

## 1.9 Lokale Funktionsdefinitionen

In Haskell können Funktionen nicht nur auf top-level definiert werden, sondern lokal innerhalb anderer Funktionen. Hierfür stehen zwei Konstrukte zur Verfügung

### 1.9.1 Mit der where-Klausel

Lokale Funktionen können unterhalb einer Funktion definiert werden. Hierzu dient das Schlüsselwort **where**.

Dazu ein kleines Beispiel, das folgende Funktion realisiert:

$$f(x, y, k) = 3 * x^k - 5 * x^2 + 3 * y^k - 5 * y^2$$

### Haskell: FirstFunctions

```
> f1 x y k = p x + p y
> where
>   p z = 3*z^k-5*z^2
```

In Haskell sind Einrückungen relevant. Alle lokalen Funktionsdefinitionen einer Funktion beginnen in derselben Spalte.

Lokale Funktionen können, wie hier im Beispiel auf das Argument **k**, auf Variablen aus dem Kontext zugreifen.

Lokale Funktionen können auch nullstellig, also Konstanten, sein. Ebenso können sie rekursiv definiert sein.

### 1.9.2 Mit let-Ausdrücken

Die zweite Möglichkeit lokale Funktionen zu definieren sind let-Ausdrücke. Er folgt erst die Definition der lokalen Funktionen und anschließend der Kontext, in dem diese verwendet werden sollen. Obige Funktion ließe sich also auch wie folgt definieren:



#### Haskell: FirstFunctions

```
> f2 x y k =  
>   let  
>     p z = 3*z^k-5*z^2  
>   in  
>     p x + p y
```

Die meisten Haskellprogrammierer verwenden mehr Definitionen mit `where` als über `let`-Ausdrücke.

## 1.10 Tupeltypen

Oft möchte man Funktionen schreiben, die nicht nur ein Ergebnis haben, sondern mehrere. Hierzu gibt es in Haskell eine einfache Möglichkeit zur Bildung von Tupeln. Diese sind in Klammern zu setzen und die Komponenten eines Tupels durch Kommata zu trennen.

Als Beispiel definieren wir einen Operator, der sowohl das Ergebnis einer ganzzahligen Division als auch den Rest der ganzzahligen Division berechnet.

#### Haskell: FirstFunctions

```
> (/%) :: Integral b => b -> b -> (b, b)  
> x /% y = (x `div` y, x `mod` y)
```

Das Ergebnis ist das Paar der beiden Werte:

#### Shell

```
*FirstFunctions> 17 /% 4  
(4,1)
```

Es gibt Standardfunktionen, um die einzelnen Komponenten eines Tupels wieder zu selektieren. Für Paare heißen diese `fst` und `snd`.

#### Shell

```
*FirstFunctions> fst`divmod 17 4  
4
```

Auch Argumente können von Tupeltypen sein. Dann sehen Funktionen fast aus wie in herkömmlichen Programmiersprachen.

#### Haskell: FirstFunctions

```
> f3(x,y) = 2 * x^2 + y^3
```

Insbesondere der Aufruf sieht so aus, als folge die Argumentliste in Klammern und mit Komma getrennt.

Shell

```
*FirstFunctions> f3(17,4)
642
```

Tatsächlich verwendet die Funktion `f3` auch Pattern Matching. Sie hat das Tupelpattern auf der linken Seite der definierenden Gleichung.

Normaler Weise nutzt man in Haskell nicht die Tupeltypen für die Argumente einer Funktion. Alleine schon, um flexibler zu bleiben und das Currying nutzen zu können.

## 2 Aufgaben

So, und nun wird es Zeit, die ersten eigenen Haskell-Funktionen zu schreiben und auszuprobieren.<sup>2</sup>

**Aufgabe 1** Zum Warmwerden beginnen wir mit ein paar ersten typische rekursiven Funktionen.

- a) Schreiben Sie die rekursive Version der Fakultät.

Haskell: FirstFunctions

```
> factorial :: (Num a, Eq a) => a -> a
> factorial _ = 0
```

- b) Setzen Sie die rekursive Definition der Fibonacci-Zahlen um.

Haskell: FirstFunctions

```
> fib :: (Ord a, Num a) => a -> a
> fib x = 0
```

- c) Setzen Sie die folgende Formel zur Berechnung der n-ten Fibonacci-Zahl in Haskell um:

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

---

<sup>2</sup>Dank für Anregungen für die Aufgaben geht an meine Kollegen Marc Zschiegner und Adrian Ulges aus den Mathemodulen unseres Studiengangs.

#### Haskell: FirstFunctions

```
> fib2 :: (Integral b1, Integral b2) => b2 -> b1
> fib2 x = 0
```

- d) Berechnen Sie rekursiv die Quersumme einer Zahl, solange bis eine einstellige Zahl vorliegt.

#### Haskell: FirstFunctions

```
> quersumme :: (Ord a, Integral a) => a -> a
> quersumme x = 0
```

- e) Schließlich soll noch die Ackermannfunktion in der 1935 von Rózsa Péter definierten Version, die nur zwei Parameter besitzt und zudem ohne die Hilfsfunktion auskommt, umgesetzt werden.

Die Definition ist:

$$\begin{aligned} a(0, m) &= m + 1 \\ a(n + 1, 0) &= a(n, 1) \\ a(n + 1, m + 1) &= a(n, a(n + 1, m)) \end{aligned}$$

Implementieren Sie die Funktion in Haskell.

#### Haskell: FirstFunctions

```
> a :: (Num a, Num t, Eq a, Eq t) => a -> t -> t
> a 0 m = 0
```

**Aufgabe 2** Das Heronverfahren definiert eine Folge, die sich der Quadratwurzel einer Zahl annähert.

$$\text{heron}(n, a) = \begin{cases} \frac{\text{heron}(n-1, a) + \frac{a}{\text{heron}(n-1, a)}}{2} & \text{für } n > 0 \\ a & \text{sonst} \end{cases}$$

Setzen Sie die obige Gleichung als Haskell-Funktion um.

#### Haskell: FirstFunctions

```
> heron :: (Ord a, Num a, Fractional p) => a -> p -> p
> heron _ _ = 0
```

**Aufgabe 3** Der erweiterte Euklidische Algorithmus ist rekursiv wie folgt formuliert:

$$\text{eEuclid}(a, b) = \begin{cases} (a, 1, 0) & \text{wenn } b = 0 \\ (d', t', s' - t' * (a \text{ div } b)) & \text{mit } (d', s', t') = \text{eEuclid}(b, a \bmod b) \end{cases}$$

Setzen Sie diesen in Haskell um:

Haskell: FirstFunctions

```
> gcdExt :: Integer -> Integer -> (Integer, Integer, Integer)
> gcdExt a b = (1, 1, 1)
```

**Aufgabe 4** Vergessen wir für einen Moment, dass im Haskell Prelude bereits mit  $\wedge$  ein Potenzoperator definiert ist.

Implementieren Sie einen eigenen Potenzoperator  $\#$  mit der Square-and-Multiply Methode, die wie folgt definiert ist:

$$x^n = \begin{cases} (x^2)^{(n/2)} & \text{falls } n \text{ gerade} \\ x * (x^2)^{((n-1)/2)} & \text{falls } n \text{ ungerade} \end{cases}$$

Haskell: FirstFunctions

```
> (#) :: (Integral a1, Num a2) => a2 -> a1 -> a2
> x#0 = 1
```

**Aufgabe 5** In dieser Aufgabe geht es um Wahrscheinlichkeitsrechnung am Beispiel des Würfelspiels *Kniffel*. Die Wahrscheinlichkeit soll dabei als rationale Zahl des Typs `Rational` berechnet werden.

Rationale Zahlen sind Bruchzahlen. Sie lassen sich durch das Bruchsymbol `%` erzeugen. Dieses Symbol stellt also nicht wie in anderen Sprachen den Modulooperator dar. Ist in einem arithmetischen Ausdruck eine rationale Zahl enthalten, so können diese auch auf rationalen Zahlen rechnen.

Shell

```
*FirstFunctions> 1%2
1 % 2
*FirstFunctions> 1%2*3/4
3 % 8
*FirstFunctions> 1/2*3/4
0.375
*FirstFunctions>
```

Berechnen Sie die Wahrscheinlichkeit, in  $n$  Versuchen (z.B. in Kniffel  $n = 3$ ) mit 5 Würfeln einen Kniffel nur aus 6en zu erreichen. Ein *Versuch* bedeutet hier, dass alle Würfel die noch nicht die Zielzahl 6 zeigen noch 1 mal gewürfelt werden dürfen.

#### Haskell: FirstFunctions

```
> kniffel :: Int -> Rational
> kniffel n = prob n 5 5
> where
>     prob n missing roll = 0
```

Die lokale Funktion `prob` hat drei Argumente.

- `n`: die Anzahl der Versuche. Im Kniffel darf man initial z.B.  $n = 3$  mal probieren.
- `missing`: die Anzahl der Würfel, die noch nicht die Zielzahl 6 zeigen.
- `roll`: der wievielte Würfel wird im aktuellen Versuch gerade gewürfelt?

Die Fälle für  $n = 0$  und `missing = 0` sind naheliegend. Im ersten kann nicht mehr zur Lösung beigetragen werden, im zweiten ist die Wahrscheinlichkeit 1.

Gilt `roll = 0`, so ist die Wahrscheinlichkeit für den nächsten Versuch  $n - 1$  mit den `missing` Würfeln zu berechnen.

Interessant ist der Fall, wenn keiner der Argumente 0 ist. Dann wird in diesem Versuch einer der Würfel geworfen (`roll - 1`). Zu  $\frac{5}{6}$  wird keine 6 geworfen. `missing` bleibt beim rekursiven Aufruf unverändert. Zu  $\frac{1}{6}$  wird eine 6 geworfen. `missing` verringert sich beim rekursiven Aufruf um 1.

Nun lässt sich die Wahrscheinlichkeit für einen Kniffel mit 6 Augen berechnen. Die Funktion `kniffel` berechnet eine rationale Zahl als Bruch. Mit der Funktion `fromRational` lässt sich diese auch als Fließkommazahl darstellen.

#### Shell

```
*FirstFunctions> kniffel 3
6240321451 % 470184984576
*FirstFunctions> fromRational $(kniffel 3)
1.3272056011374654e-2
```

**Aufgabe 6** In dieser Aufgabe sollen verschiedene Funktionen durch Taylorreihen auf rationalen Zahlen angenähert werden.

- a) Zunächst benötigen wir eine Funktion, die uns für eine Laufvariable  $n$  und einen Term `term`, in dem diese Laufvariable auftritt, eine Summe aufsteigen in Abhängigkeit von  $n$  berechnet.

#### Haskell: FirstFunctions

```
> summeNbis
> :: (Ord t1, Num a, Num t1) =>
>   t1 -> t1 -> (t1 -> t2 -> a) -> t2 -> a
> summeNbis start ende term = term start
```

Meistens werden Summen von 1 bis  $n$  oder von 0 bis  $n$  benötigt, für die eigene Funktion bereit gehalten werden-

#### Haskell: FirstFunctions

```
> summe0bis = summeNbis 0
> summe1bis = summeNbis 1
```

- b) Die Exponentialfunktion lässt sich durch folgende Taylorreihe definieren.

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \quad \text{für alle } x \in \mathbb{R}$$

Definieren Sie die Funktion für die einzelnen Terme dieser Reihe:

#### Haskell: FirstFunctions

```
> exTerm :: Integral a => a -> a -> Ratio a
> exTerm n = \x -> 0
```

Damit lässt sich die Exponentialfunktion definieren als:

#### Haskell: FirstFunctions

```
> eHoch :: Integer -> Rational
> eHoch = summe0bis 100 exTerm
```

Soll das Ergebnis nicht als rationale Zahl, sondern als Fließkommazahl vorliegen, so ist das Ergebnis noch entsprechend zu konvertieren.

#### Haskell: FirstFunctions

```
> eHoch2 :: Integer -> Double
> eHoch2 = fromRational.eHoch
```

Werfen wir einmal einen Blick auf die Eulersche Zahl:

Shell

```
*FirstFunctions> eHoch 1
4299778907798767752801199122242037634663518280784714275131782813346597523870956
720660008227544949996496057758175050906671347686438130409774741771022426508339
% 1581800261761765299689817607733339066223045468539257876032705744952135592072
867052362959959587319129243555798012243658052856289689600000000000000000000000
*FirstFunctions> eHoch2 1
2.718281828459045
*FirstFunctions>
```

- c) Auch für die Funktion des Logarithmus gibt es eine Reihenentwicklung:

$$\ln(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} (x-1)^n = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \dots \quad \text{für } 0 < x \leq 2$$

Definieren Sie die Funktion für die einzelnen Terme dieser Reihe:

Haskell: FirstFunctions

```
> lnTerm :: Integral a => a -> Ratio a -> Ratio a
> lnTerm n = \x -> 0
```

Damit lässt sich die Logarithmusfunktion definieren als:

Haskell: FirstFunctions

```
> ln = fromRational.summe1bis 10 lnTerm
```

- d) Als nächstes betrachten Sie die Reihenentwicklung für die Sinusfunktion:

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} = x - \frac{x^3}{6} + \frac{x^5}{120} - \dots \quad \text{für alle } x$$

Definieren Sie die Funktion für die einzelnen Terme dieser Reihe:

Haskell: FirstFunctions

```
> sinTerm :: Integer -> Rational -> Rational
> sinTerm n = \x -> 0
```

Um durchgängig auf rationalen Zahlen zu rechnen, verwenden Sie für die Berechnung der Fakultät, die Variante, deren Ergebnis als rationale Zahl zurück gegeben wird.

Haskell: FirstFunctions

```
> facR = toRational.factorial
```

Damit lässt sich die Sinusfunktion definieren als:

Haskell: FirstFunctions

```
> sinus :: Double -> Double  
> sinus = fromRational.summe0bis 10 sinTerm.toRational
```

e) Analog gibt es die entsprechende Reihung für die Cosinusfunktion:

$$\cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \dots \text{für alle } x$$

Definieren Sie die Funktion für die einzelnen Terme dieser Reihe:

Haskell: FirstFunctions

```
> cosTerm :: Integer -> Rational -> Rational  
> cosTerm n = \x -> 0
```

Damit lässt sich die Logarithmusfunktion definieren als:

Haskell: FirstFunctions

```
> cosinus :: Double -> Double  
> cosinus = fromRational.summe0bis 10 cosTerm.toRational
```

### 3 Lernzuwachs

- Definieren von Funktionen.
- Lazyness.
- Funktionen höherer Ordnung.
- Typinferenz.
- Operatoren.
- Unterschiedliche numerische Typen wie Integer, Rational, Double.