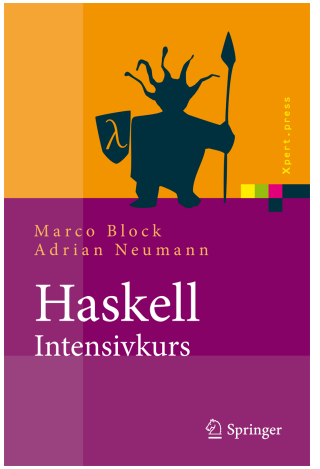
K08

Haskell

1. Grundlegende Eigenschaften
2. GHCi und GHC
3. Typsystem

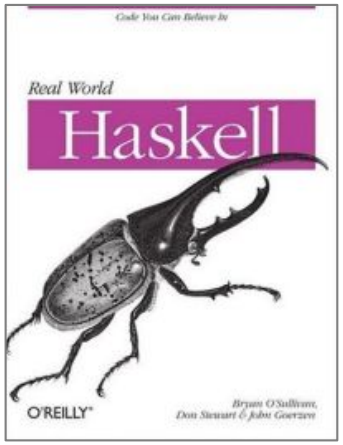
teilweise basiert auf Folien von Graham Hutton und Philip Wadler

Haskell Literatur

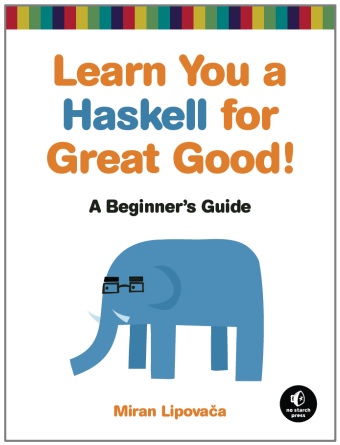


- M. Block, A. Neumann: *Haskell-Intensivkurs: Ein kompakter Einstieg in die funktionale Programmierung*. Springer, 2011. ISBN: 978-3-642-04717-6

Frei/online verfügbar:



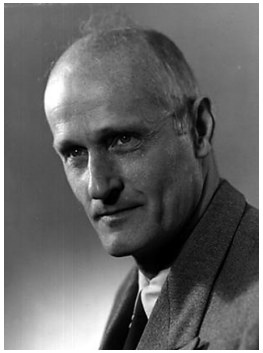
- B. O'Sullivan, D. Stewart, J. Goerzen: *Real World Haskell.* First Edition. O'Reilly, 2008. ISBN: 978-0-596-51498-3http://book.realworldhaskell.org/

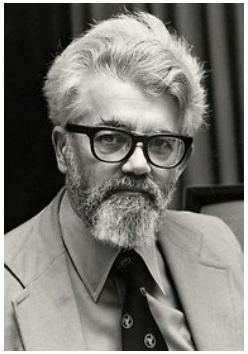


- M. Lipovača: *Learn You a Haskell for Great Good!* No Starch Press, 2011. ISBN: 978-1-59327-283-8 http://learnyouahaskell.com/

Historischer Hintergrund

- Ursprung:
  - λ-Kalkül (A. CHURCH und S. KLEENE – 1930er Jahre)
  - Currying* (HASKELL BROOKS CURRY – 1950er Jahre)
- Lisp als erste Programmiersprache mit funktionalen Eigenschaften (J. MCCARTHY 1958)
- Haskell 1.0 – (P. HUDAK, J. HUGHES, SIMON P. JONES, P. WADLER 1990); aktuelle Version: Haskell 2010



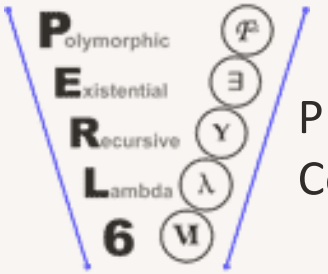
*Zuvor schon von G. FREGE und M. SCHÖNFINKEL beschrieben.

Ist FP relevant?

- Viele Konzepte der FP finden Eingang in den „Alltag“ der Softwareentwicklung bzw. in die Weiterentwicklung konventioneller Sprachen:

darcs

xmonad



PUGS PERL 6 Compiler

Projekte, die in Haskell geschrieben sind (Auswahl)

Google MapReduce

facebook

YAHOO!

twitter

Übersicht: http://www.haskell.org/haskellwiki/Haskell_in_industry

Warum gerade Haskell und nicht ...

... Erlang, F#, Lisp, ML, O'Caml, Scheme, Scala, ...

- Rein funktionale Sprache – „purely functional“
- Unter den funktionalen Programmiersprachen sehr populär
  - Grosse Menge an Bibliotheken
  - Nicht nur Gegenstand der Forschung, sondern industriell eingesetzt (man sollte es nicht als eine „akademische Spielwiese“ auffassen)
- Compiler als auch interaktive Interpreter verfügbar (z.B. *Glasgow Haskell Compiler* **`ghc`** und **`ghci`**)
  - Gute bis sehr gute Laufzeitperformance wenn kompiliert
  - Tools für Entwickler (IDEs, Debugger, Build, Profiler)
- Entworfen durch ein Komitee

Was ist Funktionale Programmierung?

Allgemein zusammengefasst:

- Programmierstil (Paradigma) dessen grundlegende Methode die **Anwendung** von **Funktionen** auf deren **Argumente** ist:

  Ein Programm ist formuliert als eine (möglicherweise aus vielen Unterfunktionen zusammengesetzte) Funktion (im math. Sinn), dessen Ausführung der Auswertung dieser Funktion entspricht.

- Eine funktionale Programmiersprache

  unterstützt / fördert / erzwingt

  diesen Programmierstil.

Gegenüberstellung

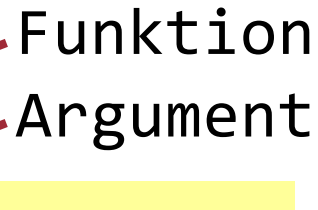
- Gesucht: Summiere die Zahlen 1...10!

In C++:

```
int sum = 0;

for (int i=1; i<=10; i++)
{
   sum += i;
}
```

Imperativ: Iteriere über die Zahlen 1 bis 10 und addiere die aktuelle Zahl zur Gesamtsumme.

In Haskell:



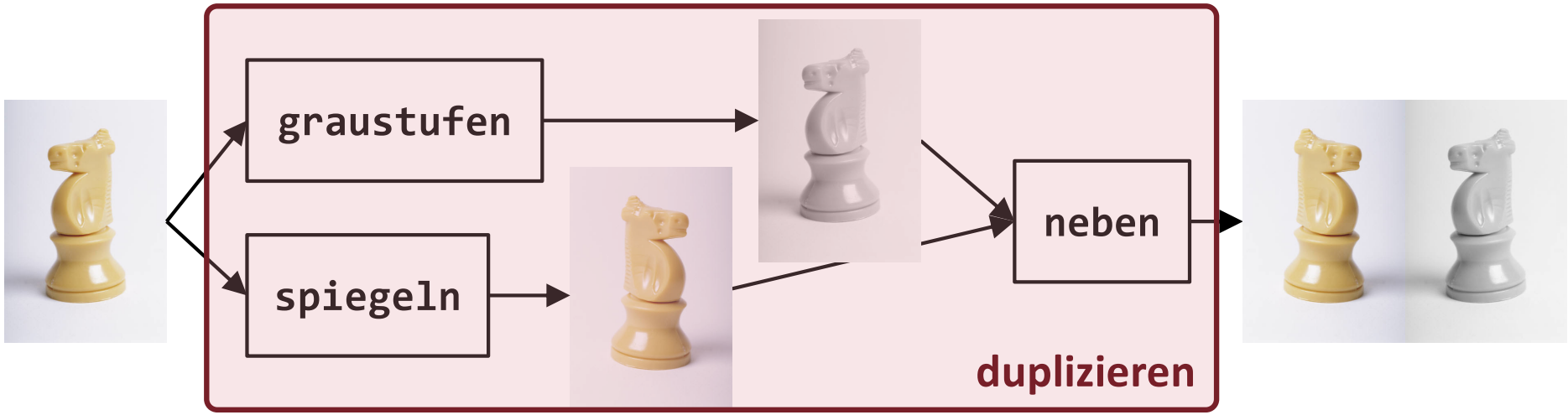

```
sum [1..10]

sum :: Num a => [a] -> a
sum []     = 0
sum [x]    = x
sum (x:xs) = x + sum [xs]
```

Applikativ: Wende die Funktion **sum** auf die Liste der Zahlen 1 bis 10 an, wobei **sum** die Funktion ist, die die Summe einer Liste von Zahlen berechnet.

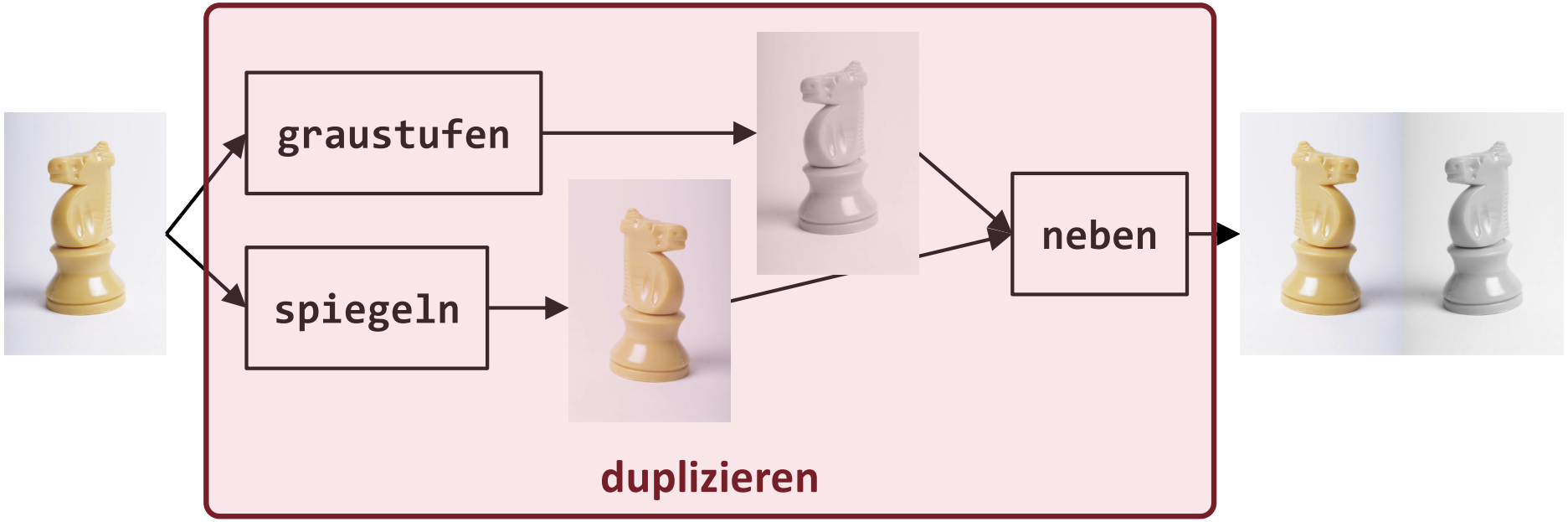
Funktionsanwendung und -Komposition

```
graustufen :: Bild -> Bild
springer :: Bild

graustufen springer
```


```
spiegeln :: Bild -> Bild
neben :: Bild -> Bild -> Bild

neben (spiegeln springer) (graustufen springer)
```



Neue Funktion aus bestehenden Funktionen definieren




```
duplizieren :: Bild -> Bild
duplizieren x = neben (spiegeln x) (graustufen x)


duplizieren springer
```

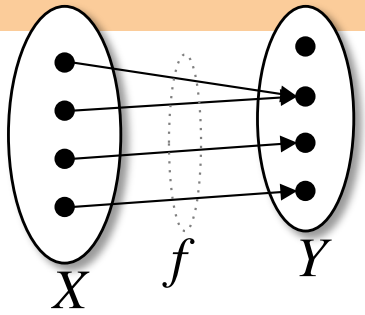
Haskell: Zurück zur Mathematik

$$x = x + 1$$

In Haskell ist dies eine rekursive Definition von x.

- In der Mathematik ist dies eine nicht lösbare Gleichung. Die Variable x ist fest an einen Wert gebunden; x ist nicht veränderbar (engl. immutable).
- In imperativen Sprachen ist dies eine Anweisung um den Zustand (Bereich im Speicher) der mit der Variable x assoziiert ist – eine Zahl in diesem Fall, die irgend etwas repräsentiert, z.B. das Alter einer Person – zu verändern.

Wiederholung: Funktion



- In der Mathematik ist eine Funktion f eine Abbildung, die ein Element x einer Menge X eindeutig (aber nicht notwendiger Weise eineindeutig) auf ein Element y einer Zielmenge Y abbildet.

$$f : X \to Y$$

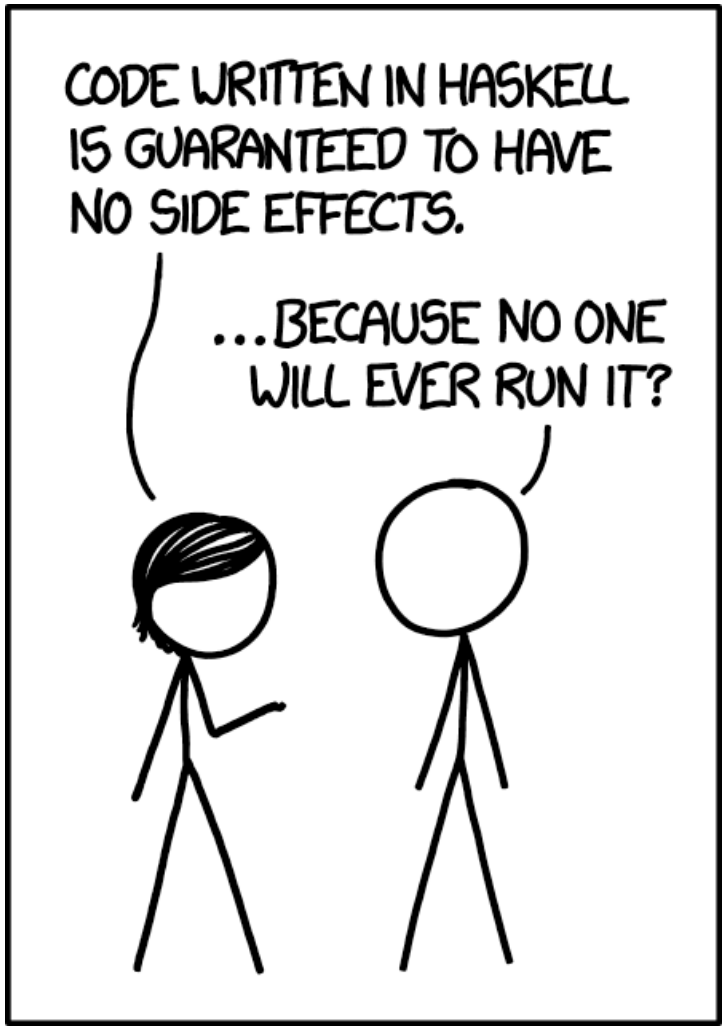
  - Linkstotal, d.h. für alle Elemente von X definiert (falls nicht, dann nennt man f eine partielle Funktion).
  - Rechtseindeutig, d.h. für jedes Element $x \in X$ gibt es höchstens ein Element in $y \in Y$ auf das x abgebildet wird.
  - X als auch Y können Produktmengen sein:

$$X = X_1 \times \ldots \times X_n, \qquad Y = Y_1 \times \ldots \times Y_m \qquad (n \geq 0, m \geq 1)$$

    wobei X_i, Y_j wiederum Mengen sind, × das kartesische Produkt und n die Stelligkeit (engl. arity) von f angibt (z.B. liefert $X = A \times B \times C$ eine dreistellige Funktion, wobei die Argumente Elemente aus A, B, C sind).

Merke

- Egal wann und egal wo f ausgewertet (oder angewendet) wird, das Ergebnis $y \in Y$ ist nur durch den Parameter $x \in X$ bestimmt (anhand der Abbildungsvorschrift die durch f vorgegeben ist) und sonst nichts!
- Ist f eine Funktion im mathematischen Sinn, dann erzeugt sie keine Seiteneffekte. Der einzige „Effekt“ der Auswertung von f ist die Berechnung des Ergebnis $y \in Y$.
  - Mit anderen Worten:
    - Der Datenfluss – d.h. die Benutzung von Ergebnissen durch nachgeordnete Funktionen – ist explizit; es gibt keine impliziten Effekte.
    - Es gibt keinen „versteckten“ (impliziten) Zustand der das Ergebnis einer Funktion beeinflusst. Demzufolge kann auch kein Zustand durch einen Funktionsaufruf verändert werden.



https://xkcd.com/1312

Referentielle Transparenz

engl. referential transparency/opaqueness

- Ist e ein Ausdruck im math. funktionalen Sinn (z.B. $g(f(x),y)$), dann ist e eindeutig durch die Argumente (und die Auswertungssemantik die für e definiert ist) bestimmt; d.h. ...
- ... gegeben die Argumente, dann kann man e durch sein Ergebnis bzw. einen äquivalenten („leichteren“) Ausdruck e' ersetzen (substituieren)
- Diese Eigenschaft bezeichnet man als referentielle Transparenz

→ Formaler mathematisch/logischer Beweis der Korrektheit eines Programms (liefert es das was es soll) wird dadurch vereinfacht bzw. überhaupt erst möglich, da keine Seiteneffekte mit einbezogen werden müssen.

Moment mal, wie kann man dann...

- ... folgende Dinge* in Haskell (bzw. rein funktionalen Programmiersprachen) realisieren:

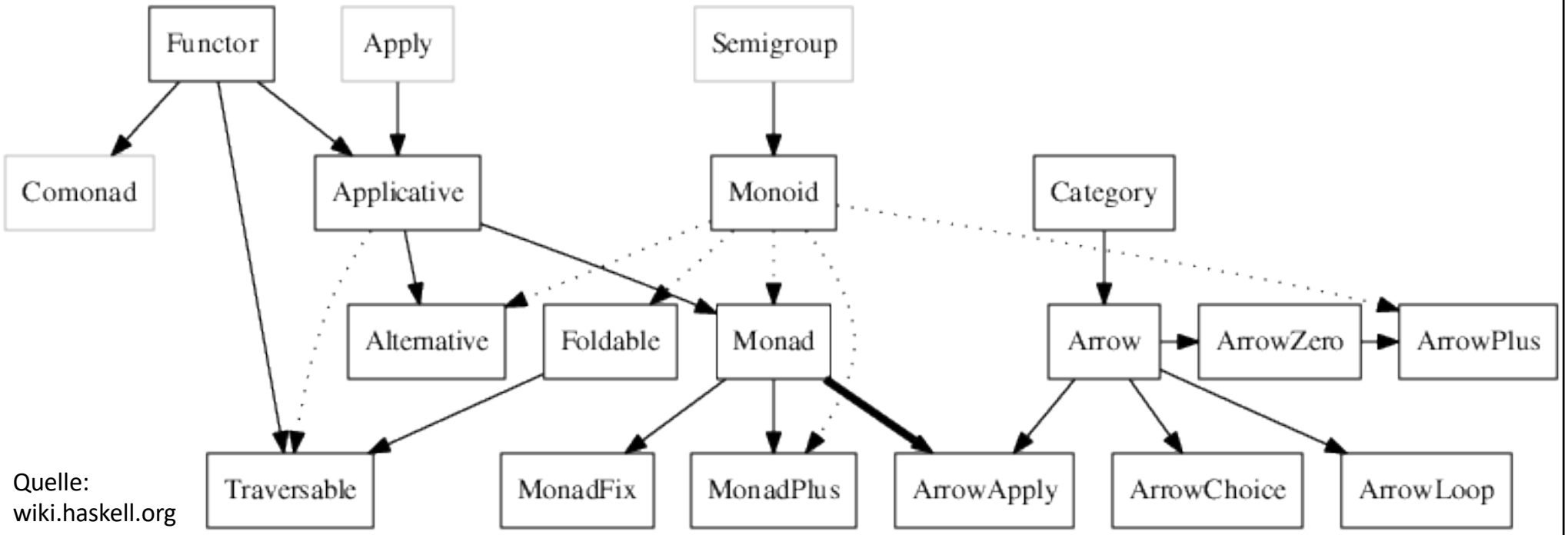
keine Funktionen im math. Sinn

- **`random()`** – erzeugt eine Zufallszahl
- **`getInput()`** – liefert ein über die Tastatur eingegebenes Zeichen
- **`currentTime()`** – liefert die aktuelle Uhrzeit/das aktuelle Datum
- **`queryGoogleFor(x)`** – liefert Suchergebnisse für Stichwort **`x`**

* An dieser Stelle sei absichtlich nicht der Begriff Funktion (bzw. Funktionalität) verwendet.

Moment mal, wie kann man dann...

- Haskell bedient sich hierfür der Monoide, Morphismen und Funktoren, welche Konzepte aus der Kategorientheorie der Mathematik sind:
  - **Action** – monadische Funktion („unreine“ Funktion mit Seiteneffekten)
  - **Arrow** – Verkettung/Komposition von Actions
  - **Monad** – Struktur aus Actions; bestimmt wie Aktionen ausgeführt werden.



Quelle:
wiki.haskell.org

$$(X \to Y) \to Z$$

Funktion höherer Ordnung

- ... ist eine Funktion bei der ein oder mehrere Argument(e) und/oder das Ergebnis wiederum eine Funktion sind.
  - Funktionen, genauso wie Argumente, sind „first-class citizens“
- Hauptanwendung liegt in der Abstraktion von mehreren Funktionen an einem Ort.

- Beispiele (informell):
  - Eine Funktion die eine Liste von Zahlen und die Quadrat-Funktion als Argumente hat, die auf jede Zahl die Quadrat-Funktion anwendet und die so entstandene Liste der Quadratzahlen zurückgibt.
  - Eine Funktion die eine Liste von Zahlen und die Maximum-Funktion als Argumente hat, die sukzessiv die Maximum-Funktion auf je zwei Zahlen anwendet, um dadurch den grössten Wert der Liste zu finden.
  - Eine Funktion, die, gegeben eine differenzierbare Funktion f, deren erste Ableitung f' als Ergebnis liefert (analytisch oder durch numerische Approx.).

Bedarfsauswertung

engl. lazy evaluation, call-by-need

- Referentielle Transparenz ermöglicht Bedarfsauswertung ...
  - ... da es keine Rolle spielt ob das Ergebnis einer Funktion bei jedem Aufruf immer (in aufwändiger Weise) berechnet wird, oder erst dann, wenn das Ergebnis wirklich gebraucht/verwendet wird, d.h. ...
  - ... ein Ausdruck wird dann ausgewertet wenn darauf zugegriffen wird.

- **Definition** (Strikte Funktion): *Sei f eine Funktion und e ein nichtterminierender Ausdruck. f ist strikt gdw. $f(e)$ nicht terminiert.*

- Haskel ist nicht strikt. Was nicht benötigt wird für eine Berechnung, wird niemals ausgewertet werden. Demzufolge sind Ausdrücke deren Auswertung unendlich lang laufen würde, oder die nicht vollständig berechenbar sind (da z.B. fehlerhaft), die aber nicht (komplett) ausgewertet werden müssen, kein Problem.

Bedarfsauswertung – Beispiel (ii)

Beispiel:

C++ (strikte Auswertung)

```
function int foo(int x) {
   return 10;          // konstante Funktion
}
foo(1 / 0);            // Ergebnis?      Absturz
```

Haskell (nicht strikte Auswertung)

```
foo :: Int -> Int
foo x = 10             -- konstante Funktion

foo (1 / 0)            -- Ergebnis?   10, statt Absturz
```

Das Ergebnis des Ausdruckes **`1 / 10`** wird nicht verwendet. Ergo wird der (in diesem Fall nicht definierte) Ausdruck nicht ausgewertet.

Bedarfsauswertung (iii)

- Ermöglicht es unendliche Strukturen zu deklarieren und damit zu arbeiten; z.B.:
  - Liste aller positiven Ganzzahlen: `[1..]`
  - Unendliche Liste von Einsen: `ones = 1 : ones`
  - Liste aller Quadratzahlen: `squares = map (^2) [1..]`

Beispiel:

```
head :: [Int] -> Int    -- Funktion welche aus einer
head []     = undefined -- Liste von Integer Zahlen
head (x:xs) = x         -- die erste Zahl zurück gibt.
head [1..]              -- terminiert und liefert “1”
head ones               --           dito
head squares            --           dito
```

- Nachteil: es ist u.U. schwieriger vorhersagbar, wann bzw. ob eine Ressource tatsächlich benutzt wird (z.B. wann ein Netzwerkzugriff erfolgt).

Weitere wesentliche Eigenschaften

- In Haskell (und anderen „rein“ funktionalen Sprachen) existieren keine Kontrollstrukturen wie z.B. Schleifen
  - Stattdessen wird Rekursion benutzt
- Strenges Typsystem: Alle Typen statisch zur Compilezeit bekannt, wodurch Typinkompatibilitäten erkennbar sind.
- Polymorphes Typsystem: Funktionen können für eine Klasse verschiedener Typen anwendbar sein.
- Automatische Speicherverwaltung: keine Zeigermanipulation; kein Anfordern und Freigeben von Speicher.

Haskell: Kostproben

Dateinamenskonvention

`hello.hs`

Das obligatorische Hello World:

```
module Main where
main :: IO()
main = putStrLn "Hello, World"
```

```
f :: Ord a => [a] -> [a]
f []     = []
f (x:xs) = f ys ++ [x] ++ f zs
           where
              ys = [a | a <- xs, a <= x]
              zs = [b | b <- xs, b > x]
```

Quick Sort ?

```
f :: Ord a => [a] -> [a]
f []     = []
f (x:xs) = f [a | a<-xs, a<=x] ++ [x] ++ f [b | b<-xs, b>x]
```

QS noch kompakter

Schlüsselwörter in Haskell

Schlüsselwörter in Haskell				
as	case	of	class	data
default	deriving	deriving instance	do	data family
forall	foreign	hiding	import	data instance
if	then	else	instance	let
in	infix	infixl	infixr	mdo
module	newtype	proc	qualified	rec
type	type family	type instance	where	

- Nur in bestimmten Kontext reserviert. Kann andernorts als Funktions- bzw. Variablen-Name benutzt werden.

K08

Haskell

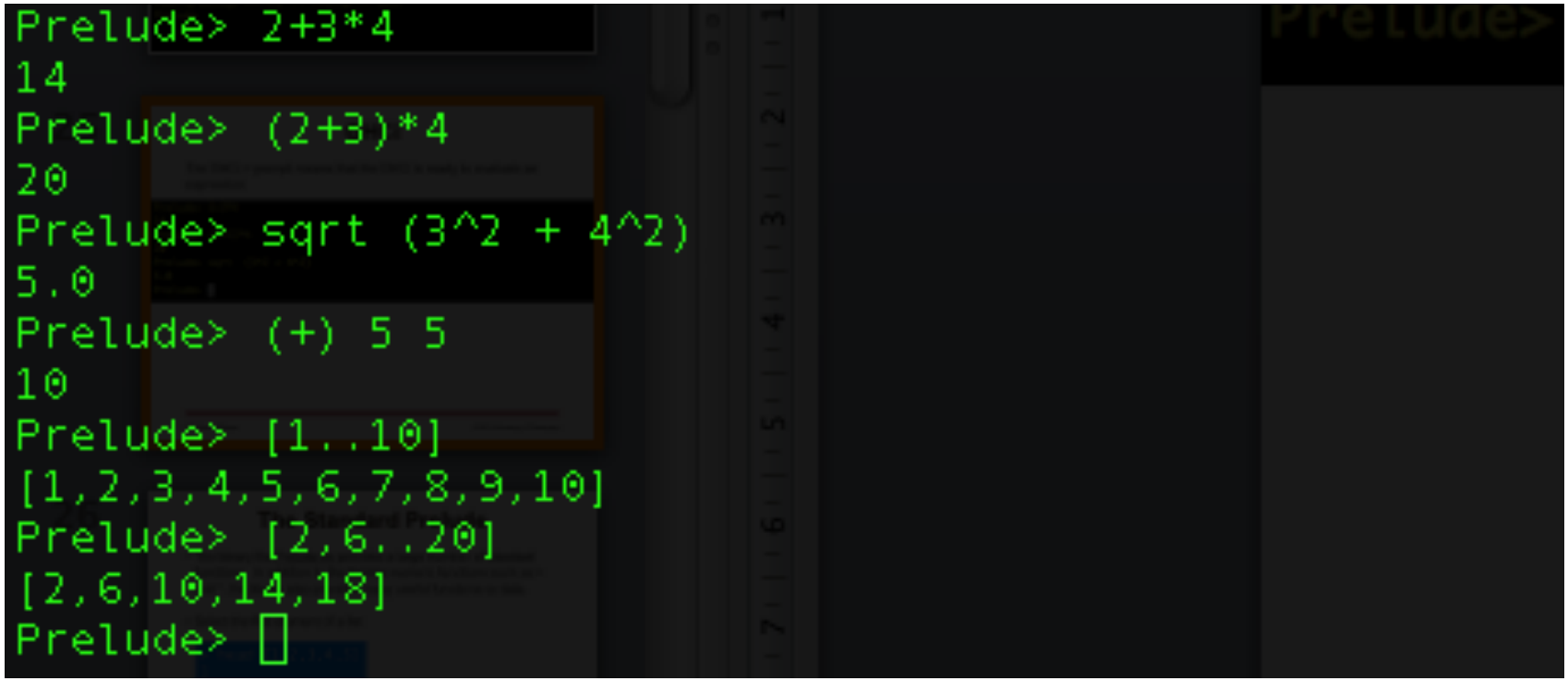
1. Grundlegende Eigenschaften
2. **GHCi und GHC**
3. Typsystem

GHCi – Interaktiver Haskell Interpreter

- *Glasgow Haskell Compiler* (GHC) kann als die Referenzimplementierung von Haskell betrachtet werden. Wird auch am häufigsten benutzt.
- GHCi ist ein interaktiver Interpreter zum Testen (Scrapbook)
  - Bezeichnet man als *Read-Eval-Print Loop* Shell (REPL)
  - Starten durch **`ghci`**

GHCi – einfache Ausdrücke auswerten

- > ist der Eingabeprompt; zeigt an, dass GHCi bereit zum Auswerten von Ausdrücken ist.



Prelude – die Standardbibliothek

- Prelude.hs ist eine Bibliothek (genauer gesagt ein `module`) welche per Default zur Verfügung steht.
  - Bietet zahlreiche, grundlegende Funktionen und Datentypen an

```
> compare 2 3          -- Relation zweier Elemente aus Menge
LT                     -- mit definierter Ordnung (LT, GT, EQ)
```

```
> head [1,2,3,4,5]     -- erstes Element einer Liste
1                      -- analog liefert tail die Restliste
```

```
> [1,2,3,4,5] !! 2     -- n-tes Element einer Liste
3
```

```
> take 3 [1,2,3,4,5]   -- ersten n Elemente einer Liste
[1,2,3]
```

Prelude – weitere Beispiele

```
> drop 3 [1,2,3,4,5]  -- Restliste ohne die ersten n Elemente
[4,5]
```

```
> length [1,2,3,4,5]  -- Länge der Liste (Anzahl Elemente)
5
```

```
> product [1,2,3,4,5] -- Produkt aller Listenelemente
120
```

```
> [1,2,3] ++ [4,5]    -- Konkatenation zweier Listen
[1,2,3,4,5]
```

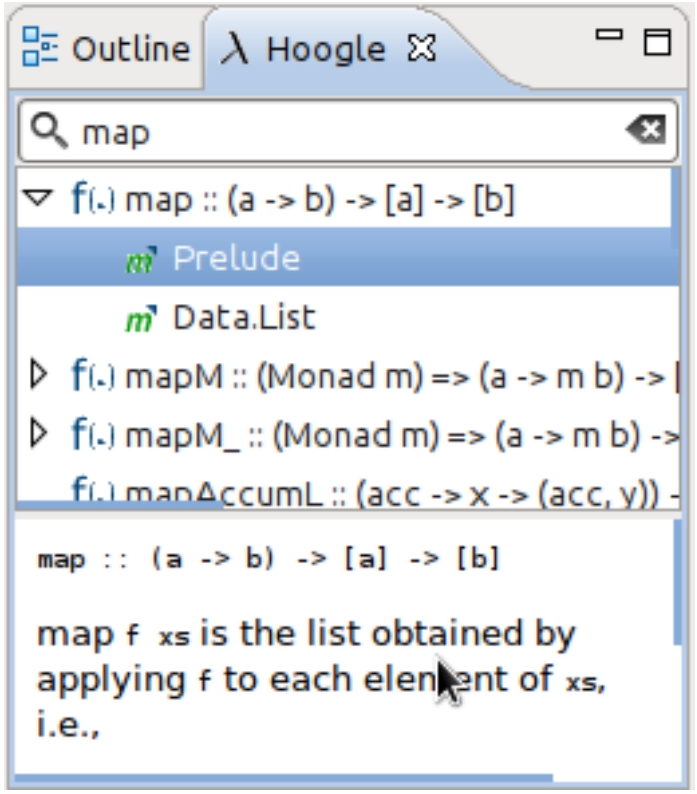
```
> reverse [1,2,3,4,5] -- umgekehrt geordnete Liste
[5,4,3,2,1]
```

```
> even 11             -- ist die gegebene Zahl gerade
False
> max 33 2            -- grössere der beiden Zahlen
33
```

Überblick gewinnen

Hoogλe

- Wie findet man schnell öffentliche/freie Bibliotheken und die darin angebotenen Funktionen und Datentypen?
- Hoogle is your friend ... http://www.haskell.org/hoogle/
  - Zentralisiertes Archiv mit Suchfunktion, in dem alle registrierten APIs dokumentiert sind
  - Suche nach:
    - Funktionsname, z.B. **`map`**
    - Signatur, z.B. **`(a -> b) -> [a] -> [b]`**
  - Eclipse-Integration über EclipseFP
  - Auch mit EMACS integrierbar



Syntax Funktionsanwendung

- In der Mathematik:

$$f(a, b) + cd$$

Wende die Funktion **f** auf die Argumente **a** und **b** an und addiere das Ergebnis zum Produkt von **c** und **d**.

f ist hier ein Funktionsname; a, b, c, d sind Variablennamen

- In Haskell:

```
f a b + c * d
```

Argumente von **f** folgen ohne Klammern; alle Operationen durch ein Symbol repräsentiert → * für die Multiplikation.

Funktionsanwendung - Beispiele

- Gegenüberstellung:

	Mathematik	**Haskell**
	$f(x)$	`f x`
	$f(x, y)$	`f x y`
	$f(g(x))$	`f (g x)`
	$f(x, g(y))$	`f x (g y)`
	$f(x)\ g(y)$	`f x * g y`
Runde Klammern:	ordnen Argument(e) einer Funktion zu	bilden Ausdrücke und damit eine Vorrangordnung bei Verschachtelung

Funktionen- und Operatorrangfolge

- Funktionen haben höhere Präzedenz als Operatoren

```
f a + b    -- bedeutet f(a) + b anstatt f(a + b)
```

- Es gibt 9 Präzedenzstufen für Operatoren
  - Relative Präzedenz mathematischer Operatoren entsprechend den mathematischen Regeln (z.B. hat ***** hat höhere Präzedenz als **+**)
- Funktionsanwendung hat Stufe 10

Skript/Programm in GHCi laden

myFirstScript.hs

```
double x    = x + x
quadruple x = double (double x)
```

Reihenfolge egal; Definition irgendwo, aber nicht notwendigerweise zuvor

- Skript mit GHCi laden: `> ghci myFirstScript.hs`

```
Prelude> :l myFirstScript.hs
```

- **`Prelude.hs`** und **`myFirstScript.hs`** sind dann geladen und die darin definierten Funktionen können aufgerufen werden.

```
Prelude> quadruple 10
40

Prelude> take (double 2) [1,2,3,4,5,6]
[1,2,3,4]
```

Skript/Programm neu laden

- Wenn Skript extern (in einem Editor) geändert wurde, muss es mit **`:reload`** (bzw. **`:r`**) erneut geladen werden um die Änderungen zu aktivieren.

```
Prelude> :r
Ok, modules loaded: Main.
Prelude> factorial 10
3628800

Prelude> average [1,2,3,4,5]
3
```

Häufig gebrauchte GHCi-Kommandos

Kommando	Kurzform	Bedeutung
:load *name*	:l *name*	lade Skript *name*
:reload	:r	aktuelles Skript neu laden
:edit *name*	:e *name*	editiere Skript *name*
:edit	:e	editiere aktuelles Skript
:info	:i	Information zu Typ/Funktion
:type *expr*	:t *expr*	zeige Typ von *expr*
:help	:?	zeige alle Kommandos
:quit	:q	beende ghci

GHC Compiler

hello.hs

```
module Main where
main :: IO()
main = putStrLn "Hello, World"
```

Programm kompilieren:

```
> ghc -o hello hello.hs
```

```
> ./hello
> Hello, World
```

- Analog zu vielen anderen Sprachen ist die **`main`** Funktion (optional im Modul **`Main`**) – welche gleichzeitig eine sog. *Aktion* ist; dazu später mehr – der Einstiegspunkt in ein Haskell-Programm.

Einrückung als Strukturierungselement (i)

- In Haskell (ähnlich wie z.B. in Python) kann die Einrückung im Quelltext eine Rolle spielen: Blöcke können durch Einrückung gebildet werden, d.h. Whitespace (Leerzeichen und Tabulatoren) hat eine abgrenzende Semantik.
- Wenn Blockbildung durch Einrückung, dann müssen in einer Sequenz von Zeilen (z.B. Definitionen) alle in exakt derselben Spalte beginnen:

```
a = 10
b = 20
c = 30
```

```
a = 10
 b = 20
c = 30
```

```
 a = 10
b = 20
 c = 30
```

```
xs = 10
 x = 20
 c = 30
```

Einrückung als Strukturierungselement (ii)

- Blockbildung durch Einrückung vermeidet die Notwendigkeit spezielle Symbole zur Blockbegrenzung benutzen zu müssen.

```
a = b + c
    where
      b = 1
      c = 2
d = a * 2
```

bedeutet

```
a = b + c
      where
        {b = 1;
         c = 2}
d = a * 2
```

implizite Gruppierung

explizite Gruppierung

K08

Haskell

1. Grundlegende Eigenschaften
2. GHCi und GHC
3. **Typsystem**

Typen (Wiederholung)

- Ein Typ ist ein Bezeichner für eine Menge von gleichartigen Elementen bzw. Werten (engl. values).
  - Eine Klasse ist z.B. eine üblicherweise nichtleere Menge von Objekten (Instanzen) die sich der Klasse zuordnen lassen.
  - Auf einem Typ ist eine Menge von Operationen und Funktionen definiert.

- In Haskell existiert z.B. der Typ `Bool`

  welcher genau zwei Werte enthält: `False` `True`

Typfehler in Haskell

- Die Anwendung einer Funktion auf inkompatible Argumente ist ein Typfehler (engl. type error).

```
> 1 + False
Error
```

1 ist eine Zahl and **False** ein Bool; jedoch erwartet **+** zwei Zahlen.

Typen in Haskell

- Wenn die Auswertung eines Ausdrucks e einen Wert (Ergebnis) vom Typ t liefert, dann ist e vom Typ t

```
e :: t
```

- Jeder syntaktisch korrekte Ausdruck hat in Haskell demzufolge einen Typ, welcher sich automatisch zur Compilezeit ableiten lässt. Diesen Prozess nennt man Typinferenz (engl. type inference).
  - Haskell hat ein statisches und strenges Typsystem
    - Es können keine Typfehler zur Laufzeit auftreten
    - Geringere Fehleranfälligkeit und bessere Laufzeitperformance (da keine Checks zur Laufzeit notwendig sind)

Typ eines Ausdrucks abfragen

- Der Typ eines Ausdrucks e kann in GHCi mit dem Kommando

  `:type` (bzw. **`:t`** als Kurzform)

  abgefragt werden, ohne dass e dabei ausgewertet wird.

```
> :t "foo"
"foo" :: [Char]

> :t not False
not False :: Bool

> :t 3 + 2
3 + 2 :: Num a => a
```

Allgemeine Datentypen in Haskell

Typ	Breite/Präzision	Wertebereich
`Bool`		True, False
`Int`	fest (32, 64 bit)[1]	mindestens -2^{29} bis $2^{29}-1$
`Integer`	theoretisch unbeschränkt	**`-∞`** bis **`+∞`**
`Float`[2]	einfach (32 bit)	$\approx 1.175\times10^{-38}$ bis $\approx 3.4\times10^{38}$
`Double`	doppelt (64 bit)	$\approx 4.9406564584124654\times10^{-32}$[4] bis $\approx 1.7976931348623157\times10^{308}$
`Char`	Einzelnes Zeichen	UNICODE
`String`	Zeichenkette (Liste von Zeichen)	
`Rational`[3]	Quotient zweier **`Integer`**-Werte	
`Complex`		

[1] Maschinenabhängig

[2] Laut „Real World Haskell“ Verwendung nicht empfohlen „... is much slower ...“

[3] **`Ratio`** Bibliothek

Listentypen

- Eine Liste ist eine Folge von Werten desselben Typs:

```
[False,True,False] :: [Bool]

['a','b','c','d']  :: [Char]
```

- Generell gilt:

  $[t]$ ist der Typ von Listen deren Elemente vom Typ t sind

- Elemente können wiederum Listen desselben Typs sein:

```
[['a'],['b','c']] :: [[Char]]
```

- Typ ist unabhängig von der Länge der Liste
  - Spezialfall: leere Liste

```
> :t []
[] :: [a]
```

Typeltypen

- Ein Tupel ist eine Folge von Elementen deren jeweiliger Typ unterschiedlich sein kann:

```
(False,True)      :: (Bool,Bool)
(False,'a',True)  :: (Bool,Char,Bool)
("foo",1,2)       :: (Num t1, Num t) => ([Char],t,t1)
('a',('b','c'))   :: (Char,(Char,Char))
```

- Generell gilt:

  $(t_1, t_2, \ldots, t_n)$ ist der Typ von n-Tupeln deren i-tes Element vom Typ t_i ist $(1 \leq i \leq n)$

- Aus dem Typ kann man also die Länge des Tupels ableiten.

- Spezialfall: leeres Tupel

```
> :t ()
() :: ()
```

Funktionstypen

- Eine Funktion ist eine Abbildung von Werten eines Typs auf Werte eines Typs.

```
not      :: Bool -> Bool
isDigit  :: Char -> Bool
```

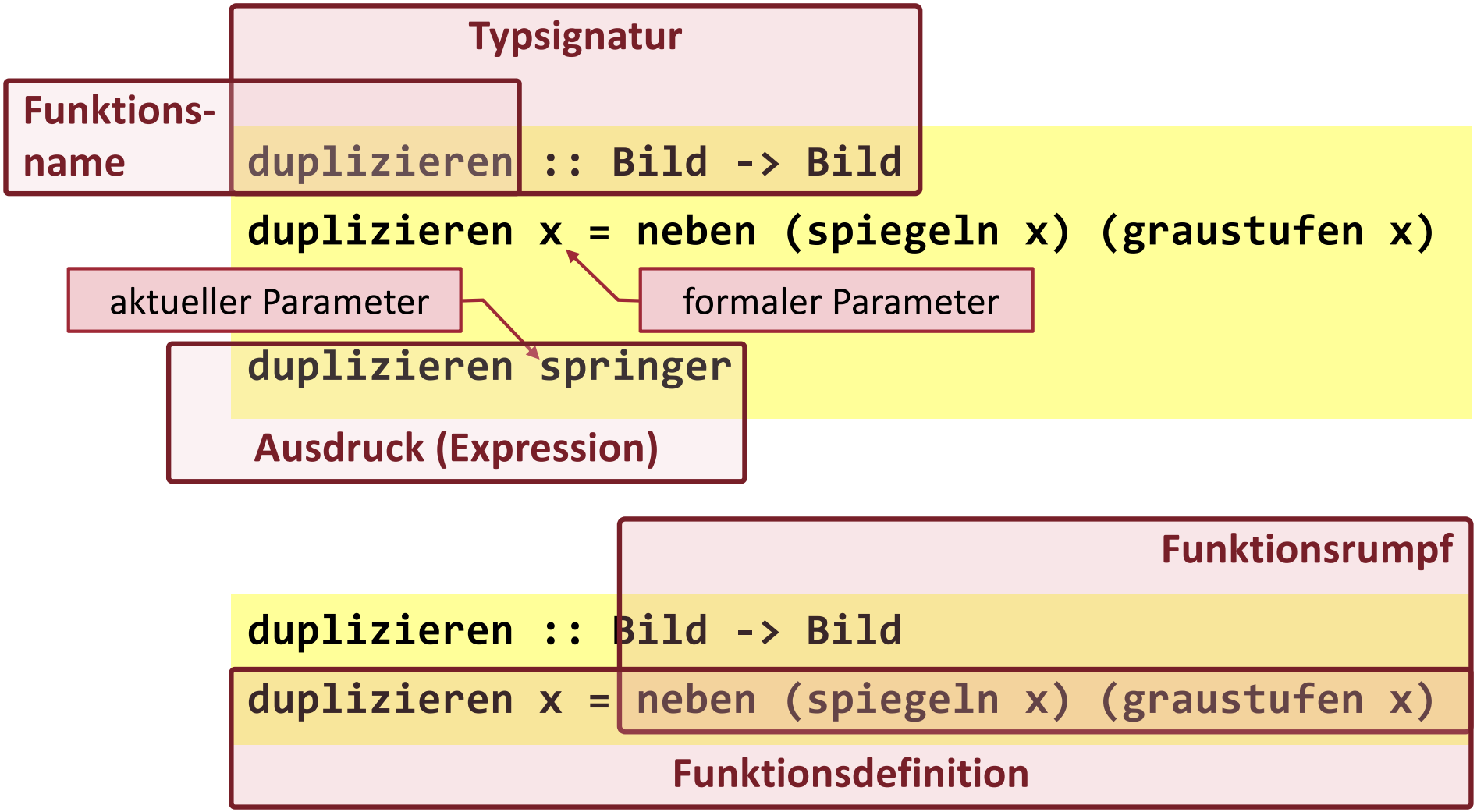
- Generell gilt:

t_1 -> t_2 ist der Typ der Klasse von Funktionen,
die Werte vom Typ t_1 auf Werte vom Typ t_2 abbilden.

- Argument und Ergebnistyp sind nicht beschränkt:

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

zeroToN     :: Int -> [Int]
zeroToN n   = [0..n]
```

Terminologie



Curryfizierte Funktionen (i)

- Funktionen mit mehreren Argumenten gibt es eigentlich gar nicht in Haskell: Alle Funktionen haben genau ein Argument (oder gar keines falls sie konstant sind).
- Eine Funktion mit mehreren Argumenten lässt sich als Funktion darstellen, die ein Argument hat und eine Funktion als Ergebnis liefert.

$$f : (A_1 \times \cdots \times A_n) \to B \quad \rightsquigarrow \quad f' : A_1 \to (A_2 \to (\ldots (A_n \to B) \ldots))$$

```
add' :: Int -> Int -> Int
add' x y = x+y
```

`add'` hat eine **`Int`**-Zahl als Argument (**`x`**) und liefert eine Funktion vom Typ **`Int -> Int`**. Diese Funktion wiederum hat eine **`Int`**-Zahl (**`y`**) als Argument und liefert eine **`Int`**-Zahl als Ergebnis **`x+y`**.

Curryfizierte Funktionen (ii)

```
add' :: Int -> Int -> Int
add  :: (Int, Int) -> Int
```

- **add** und **add'** produzieren dasselbe Endergebnis, jedoch nimmt **add** beide Argumente zur selben Zeit (über ein Tupel), währenddessen **add'** jeweils nur ein Argument verarbeitet.
- Funktionen die jeweils nur ein Argument verarbeiten bezeichnet man als Currifiziert (Prozess selten auch *Schönfinkeln* genannt).
- Hauptvorteil theoretischer Art: formale Beweise der Korrektheit von Programmen wird vereinfacht, wenn alle Funktion gleichförmig sind.

Curryfizierte Funktionen (iii)

- Beachte:
  - Abbildungspfeil in Funktionssignatur rechtsassoziativ:

```
Int -> Int -> Int -> Int
```

```
Int -> (Int -> (Int -> Int))
```

  - Funktionsapplikation linksassoziativ:

```
foo x y z
```

```
((foo x) y) z
```

Partielle Funktionsanwendung

- Curryfizierung ermöglicht partielle Funktionsanwendung:

```
> :t zip3  -- Funktion die Dreiertupel aus drei Listen bildet
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]

> zip3 "foo" "bar" "quux"
[('f','b','q'),('o','a','u'),('o','r','u')]

> let zip3foo = zip3 "foo"
> :type zip3foo
zip3foo :: [b] -> [c] -> [(Char, b, c)]

> zip3foo "aaa" "bbb"
[('f','a','b'),('o','a','b'),('o','a','b')]

> zip3foo [1,2,3] [True,False,True]
[('f',1,True),('o',2,False),('o',3,True)]
```

Polymorphe Funktionen (i)

- Eine Funktion ist polymorph wenn ihr Typ mindestens eine Typvariable enthält.

```
length :: [a] -> Int
```

Für beliebige Typen **a** liefert **length** einen Integer als Ergebnis, gegeben eine Liste mit Elementen vom Typ **a**.

Beispiel:

```
> length [False,True] -- a = Bool
2

> length [1,2,3,4]    -- a = Int
4
```

Beispiele polymorpher Funktionen

- Zahlreiche Funktionen aus **Prelude.hs** (und anderen Bibliotheken) sind polymorph, z.B.:

```
fst  :: (a,b) -> a

head :: [a] -> a

take :: Int -> [a] -> [a]

zip  :: [a] -> [b] -> [(a,b)]

id   :: a -> a

compare :: Ord a => a -> a -> Ordering
```

Überladene Funktionen

- Eine polymorphe Funktion ist überladen wenn ihr Typ mindestens eine Typnebenbedingung (engl. *type constraint*) enthält.

Beispiel:

```
sum :: Num a => [a] -> a
```

Für beliebige Typen **a** die von **Num** abgeleitet sind liefert **sum** ein Ergebnis vom Typ **a** gegeben eine Liste vom Typ **a**.

```
> sum [1,2,3]          -- Int
6

> sum [1.1,2.2,3.3]    -- Float
6.6

> sum ['a','b','c']    -- Char
ERROR                  -- kein Num
```

Num ist eine Typklasse die als Oberklasse für numerische Datentypen vorgesehen ist.

Typklassen

- Typklassen in Haskell sind oberflächlich betrachtet vergleichbar mit Interfaces in Java.

`Num`	Numerische Typen
`Eq`	Typen für deren Elemente Gleichheit definiert ist.
`Ord`	Typen auf deren Elementen eine Ordnung definiert ist.

Beispiele:

```
(+)  :: Num a => a -> a -> a
(==) :: Eq a  => a -> a -> Bool
(<)  :: Ord a => a -> a -> Bool
```

Namensregeln

- Haskell unterscheidet zwischen Gross- und Kleinschreibung

```
name        naMe        Name
```

unterschiedliche Namen/Bezeichner

- Typen und Konstruktoren beginnen mit einem Grossbuchstabe
- Variablen und Funktionen beginnen mit einem Kleinbuchstabe
- Zusätzlich darf auch der Unterstrich _ und weitere Sonderzeichen für Funktionen verwendet werden:

```
_myFun    myFun    fun1    arg_2    x'
```

- Neuere Versionen unterstützen auch UNICODE-Zeichen:

```
пять    два    умножить
```

Operatoren

- Sind auch Funktionen:
  binärer Operator $a \circ b$ ist zweistellige Funktion $\circ(a, b)$
- Standardmässig vorhandene binäre/unäre Operatoren
  - Arithmetische Operatoren:

    `+`, `*`, `^`, `-`, `div`, `mod`, `abs`, `negate`
  - Vergleichsoperatoren:

    `>`, `>=`, `==`, `/=`, `<=`, `<`

    beachte: nicht `!=` wie in C/C++/Java
- Für die Definition neuer Operatoren gelten im Prinzip dieselben Namensregeln wie für Funktionen
  - Verwendbare Zeichen:

    `! # $ % & * + . / < = > ? @ \ : - | ~ ^`
  - Reservierte Namen (Bezeichner):

    `_  ..  :  ::  =  \  |  <-  ->  @  ~  =>`

Fixity: Assoziativität und Präzedenz

- Die Kombination aus Assoziativität und Präzedenz von Operatoren wird auch als **Fixity** bezeichnet:
  - `infixl` – Infixoperator, linksassoziativ, z.B.:

    `a+b+c+d` → **`((a+b)+c)+d`**

  - `infixr` – Infixoperator, rechtsassoziativ, z.B.:

    `(f.g.h) a` → **`f (g (h a))`**

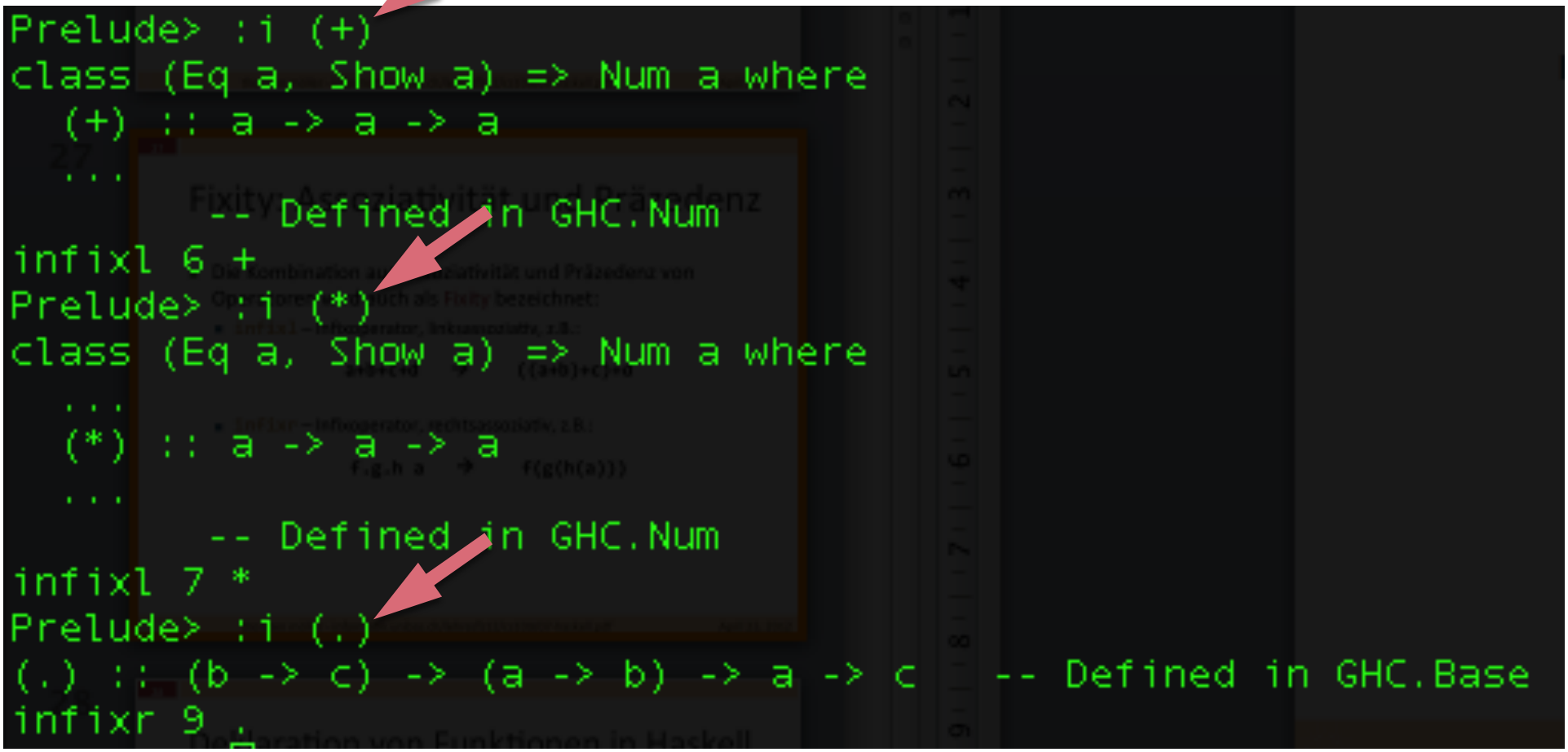
  - `infix` – Infixoperator, nichtassoziativ, z.B.:

    `a<b<c` → weder **`(a<b)<c`** noch **`a<(b<c)`** gilt

- Deklaration:

```
infixr 5 ++ -- rechtsassoz.; Präzedenz 5
infixl 9 .  -- linksassoz.; Präzedenz 9
```

Fixity - Beispiele



```
Prelude> :i (+)
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  ...
        -- Defined in GHC.Num
infixl 6 +
Prelude> :i (*)
class (Eq a, Show a) => Num a where
  ...
  (*) :: a -> a -> a
  ...
        -- Defined in GHC.Num
infixl 7 *
Prelude> :i (.)
(.) :: (b -> c) -> (a -> b) -> a -> c    -- Defined in GHC.Base
infixr 9 .
```

Beispiel: Operator definieren

- Operator +++ und *** definieren:

myOp.hs

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

(+++) :: Int -> Int -> Int
a +++ b = a + 2*b

(***) :: Int -> Int -> Int
a *** b = a - 4*b
```

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

- Anwendung (GHCi):

```
Prelude> l: myOp.hs
Prelude> 1 +++ 2 *** 3
-19
```

```
(1 + 2*(2 - 4*3))
```

```
((1 + 2*2) - 4*3)
```