

# PROGRAMMIERUNG UND MODELLIERUNG MIT HASKELL

## EINFÜHRUNG

Martin Hofmann   Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,  
Ludwig-Maximilians Universität, München

7. April 2014

# ORGANISATION

## VORLESUNG

- Anmeldung zur Vorlesung per UniworX zwingend
- Umfang: 3 SWS, d.h. einige Vorlesungstermine entfallen; Newsfeed der Vorlesungshomepage beachten!

<https://www.tcs.ifi.lmu.de/lehre/ss-2014/promo/promo>

## ÜBUNGEN

- Teilnahme freiwillig, aber empfehlenswert  
UniworX Anmeldung zur Übungsgruppe zwingend
- Hausübung bringen prozentual Bonuspunkte zur Klausur  
 $x\%$  Hausübung bringen  $x\%$  des Bonus;  
Bonus entscheidet nicht über bestehen
- Beginn der Übung in 2. Semesterwoche



# INHALT DER VORLESUNG

## Programmierung und Modellierung

- Funktionsbegriff, Basistypen
- Listen, Pattern Matching
- Rekursion und Terminierung
- Induktion & Korrektheitsbeweise
- Benutzerdefinierte Datentypen
- Tiefen- & Breitensuche
- Laufzeitbetrachtungen
- Polymorphie, Typklassen und Module
- Funktionen höherer Ordnung
- Typisierung & Typinferenz
- IO, Monaden & Funktoren
- Verzögerte Auswertung
- Semantik
- Parallele Auswertung



## LITERATUR

## DIE VORLESUNG RICHTET SICH NACH FOLGENDEN QUELLEN:

	Buch/Online	
<b>Programming in Haskell</b> by Graham Hutton	B	
<b>Learn You a Haskell for Great Good!</b> by Miran Lipovača	B	O
<b>A Gentle Introduction To Haskell</b> by Paul Hudak, John Peterson, Joseph Fasel		O
<b>Real World Haskell</b> by Bryan O'Sullivan, Don Stewart, John Goerzen	B	O
<b>School of Haskell</b> by FPComplete		O
<b>Haskell Wiki</b>		O

so wie ältere Skripte des Lehrstuhls TCS

Links/ISBN der Quellen, Skript ⇒ Vorlesungshomepage

**WEITERE WICHTIGE ONLINE-QUELLE: [www.haskell.org](http://www.haskell.org)**

Haskell-Plattform: GHC Kompiler & Interpreter,

Dokumentation der Standardbibliotheken, Hoogle



# WAS IST INFORMATIK?

Von franz. **informatique** (= **information** + **mathématiques**).  
Engl.: **computer science**, auch **informatics**.

- DUDEN Informatik: Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Computern.
- Gesellschaft f. Inf. (GI): Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Informationen.
- Association vor Computing Machinery (ACM): Systematic study of algorithms and data structures.



# TEILBEREICHE DER INFORMATIK

- Technische Informatik  
Aufbau und Wirkungsweise von Computern
- Praktische Informatik  
Konstruktion von Informationsverarbeitungssystemen sowie deren Realisierung auf Computern
- Theoretische Informatik  
Theoretische und verallgemeinerte Behandlungen von Fragen und Konzepten der Informatik
- Angewandte Informatik  
Verbindung von Informatik mit anderen Wissenschaften



# TYPISCHE ARBEITSGEBIETE

- Algorithmen und Komplexität
- Betriebssysteme
- Bioinformatik
- Datenbanken
- Grafik
- Medieninformatik
- Programmiersprachen und Compiler
- Rechnerarchitektur und Rechnernetze
- Robotik
- Simulation
- Softwareentwicklung
- Spezifikation, Verifikation, Modellierung
- Wirtschaftsinformatik



# IMPERATIV VS. DEKLARATIV

Imperative Sprachen (Assembler, C, Java, etc.) orientieren sich an den Fähigkeiten der Maschine. Programmierung wird durch Abstraktion vieler Einzelschritte vereinfacht (z.B. Schleifen).

Funktionale Programmierung ist dagegen **deklarativ**:

*Spezifiziere **was** berechnet werden soll,  
**nicht wie** es berechnet wird!*

Charakteristische Eigenschaft deklarativer Sprachen ist die **Church-Rosser Eigenschaft**:

*Die Reihenfolge der Auswertung ist unerheblich für das Ergebnis der Berechnung ist.*





# IMPERATIV VS. DEKLARATIV

Imperative Sprachen (Assembler, C, Java, etc.) orientieren sich an den Fähigkeiten der Maschine. Programmierung wird durch Abstraktion vieler Einzelschritte vereinfacht (z.B. Schleifen).

Funktionale Programmierung ist dagegen **deklarativ**:

*Spezifiziere **was** berechnet werden soll,  
**nicht wie** es berechnet wird!*

Charakteristische Eigenschaft deklarativer Sprachen ist die **Church-Rosser Eigenschaft**:

*Die Reihenfolge der Auswertung ist unerheblich für das Ergebnis der Berechnung ist.*



# IMPERATIV VS. DEKLARATIV

Deklarative Sprachen sind fast immer durch die Sprache der Mathematik motiviert:

Prädikaten-Logik	⇒	Prolog
Relationale Algebra	⇒	SQL
$\lambda$ -Kalkül	⇒	Haskell

Rein deklarative Sprachen in der Praxis jedoch oft problematisch, da der Ressourcenverbrauch eines Programmes (Speicherplatz, Zeit, etc.) kritisch vom **wie** abhängt. (Prolog, SQL ohne join).

Funktionale Programmierung geht einen Mittelweg:

Die Berechnung bleibt **deterministisch**  
wir kümmern uns nur nicht so sehr darum



# IMPERATIV VS. DEKLARATIV

Deklarative Sprachen sind fast immer durch die Sprache der Mathematik motiviert:

Prädikaten-Logik	⇒	Prolog
Relationale Algebra	⇒	SQL
$\lambda$ -Kalkül	⇒	Haskell

Rein deklarative Sprachen in der Praxis jedoch oft problematisch, da der Ressourcenverbrauch eines Programmes (Speicherplatz, Zeit, etc.) kritisch vom **wie** abhängt. (Prolog, SQL ohne join).

Funktionale Programmierung geht einen Mittelweg:

Die Berechnung bleibt **deterministisch**

wir kümmern uns nur nicht so sehr darum



# REFERENTIELLE TRANSPARENZ

Wichtige Eigenschaft deklarativer Sprachen:

## Referentielle Transparenz

- Wert einer Variablen ist **unveränderlich**
- Aufrufe mit gleichen Argumenten liefern gleiches Ergebnis
- Keine Seiteneffekte!

## KONSEQUENZEN:

- Programme lokal verständlich und kompositional
- Testen/Verifikation reduziert sich auf Gleichheitsschließen
- Kompiler kann sehr aggressiv optimieren
- Parallele Berechnung einfacher



# VOR-/NACHTEILE FUNKTIONALER PROGRAMM.

FP ist alt, doch kaum verbreitet?!  $\lambda$ -Kalkül 1930s, Lisp 1958

- Völlig andere (mathematische) Denkweise
- Verlust von "Kontrolle" gegenüber herkömmlichen Sprachen, Maschinen-nahe Hand-Optimierung schwierig durchführbar
- Geringe kommerzielle Unterstützung IDE, Debugger,...
- Kompiler "meckert viel" bevor man testen kann

ABER:

- "Well-typed programs can't go wrong!", R.Milner
- Gut geeignet für Programmanalyse
- "Einschränkungen" vorteilhaft für *Mehrpersonen*-Projekte
- Kurzer Code  $\Rightarrow$  kurze Entwicklungszeit; Wartbarkeit
- Funktionale Merkmale inzwischen in vielen anderen Sprachen  
z.B. Funktionen höherer Ordnung, Anonyme Funktionen, Java GC optimiert auf kurzlebige, statische Objekte, etc.
- Neues kommerzielle Interesse wegen Multi-Cores

"The Downfall of Imperative Programming", Milewski



# VOR-/NACHTEILE FUNKTIONALER PROGRAMM.

FP ist alt, doch kaum verbreitet?!  $\lambda$ -Kalkül 1930s, Lisp 1958

- Völlig andere (mathematische) Denkweise
- Verlust von “Kontrolle” gegenüber herkömmlichen Sprachen, Maschinen-nahe Hand-Optimierung schwierig durchführbar
- Geringe kommerzielle Unterstützung IDE, Debugger, . . .
- Kompiler “meckert viel” bevor man testen kann

## ABER:

- “Well-typed programs can't go wrong!”, R.Milner
- Gut geeignet für Programmanalyse
- “Einschränkungen” vorteilhaft für *Mehrpersonen*-Projekte
- Kurzer Code  $\Rightarrow$  kurze Entwicklungszeit; Wartbarkeit
- Funktionale Merkmale inzwischen in vielen anderen Sprachen  
z.B. Funktionen höherer Ordnung, Anonyme Funktionen,  
Java GC optimiert auf kurzlebige, statische Objekte, etc.
- Neues kommerzielle Interesse wegen Multi-Cores

“The Downfall of Imperative Programming”, Milewski



# VOR-/NACHTEILE FUNKTIONALER PROGRAMM.

FP ist alt, doch kaum verbreitet?!  $\lambda$ -Kalkül 1930s, Lisp 1958

- Völlig andere (mathematische) Denkweise
- Verlust von “Kontrolle” gegenüber herkömmlichen Sprachen, Maschinen-nahe Hand-Optimierung schwierig durchführbar
- Geringe kommerzielle Unterstützung IDE, Debugger, ...
- Kompiler “meckert viel” bevor man testen kann

## ABER:

- “Well-typed programs can’t go wrong!”, R.Milner
- Gut geeignet für Programmanalyse
- “Einschränkungen” vorteilhaft für *Mehrpersonen*-Projekte
- Kurzer Code  $\Rightarrow$  kurze Entwicklungszeit; Wartbarkeit
- Funktionale Merkmale inzwischen in vielen anderen Sprachen  
z.B. Funktionen höherer Ordnung, Anonyme Funktionen,  
Java GC optimiert auf kurzlebige, statische Objekte, etc.
- Neues kommerzielle Interesse wegen Multi-Cores

“The Downfall of Imperative Programming”, Milewski



# HASKELL

- Effektfreie funktionale Sprache mit verzögerter Auswertung für universellen Einsatz
- Benannt nach Haskell Curry (1900-82), Logiker
- Standards: Haskell98 und Haskell2010
- Viele verschiedene Implementierung verfügbar; wichtigste ist die **Haskell Platform** mit “batteries included”:  
Kompiler, Interpreter und zahlreichen Bibliotheken

GHC : Glasgow/Glorious Haskell Compiler     Hammond, 1989  
akutelle Versionen mehrmals pro Jahr

Hauptentwickler erhielten 2011 ACM SIGPLAN Programming Languages Software Award:

Simon Peyton-Jones

Simon Marlow

Microsoft Research Cambridge

Facebook





# WARUM GERADE HASKELL?

Für einen großen Teil der Vorlesungsinhalte spielt es keine Rolle, welche **funktionale** Sprache wir betrachten.

- Haskell ist im Gegensatz zu vielen anderen funktionalen Sprachen (z.B. SML) **rein funktional**, d.h. referentielle Transparenz gilt uneingeschränkt
- Haskell hat ein starkes statisches Typsystem
- Viel aktuelle Dokumentation verfügbar
- Kommerzielle Unterstützung IDE auf [fpcomplete.com](http://fpcomplete.com)
- Arithmetik mit beliebiger Präzision verfügbar
- Gute Unterstützung für Parallelität
- Produziert “flotten” Code



# GHC

In der Vorlesung arbeiten wir mit der **Haskell Platform**, welche aus GHC, Bibliotheken und Dokumentation besteht. Die meiste Dokumentation ist aber auch online verfügbar, wie zum Beispiel <http://www.haskell.org/ghc/docs/latest/html/libraries/>

Glasgow Haskell Kompiler (GHC) kennt zwei Arbeitsweisen:

**GHC** Normaler Kompiler. Ein Programm wird in mehreren Dateien geschrieben und mithilfe des GHC in ein ausführbares Programm übersetzt.

**GHCi** Interpreter Modus: Man gibt Definitionen ein und GHCi wertet diese sofort aus und zeigt den Wert an.

Wir befassen uns zunächst primär mit dem interaktiven Modus.



## GHCi

```
> ghci
```

```
GHCi, version 7.6.3: http://www.haskell.org/ghc/
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> 1 + 2
3
Prelude> 3 + 4 * 5
23
Prelude> (3 + 4) * 5
35
Prelude>
```

Der Text hinter dem Prompt `Prelude>` wurde vom Benutzer getätigt, alles andere sind Ausgaben von GHCi. Der Prompt gibt



## GHCi

```
> ghci +RTS -M1g
```

```
GHCi, version 7.6.3: http://www.haskell.org/ghc/  
Loading package ghc-prim ... linking ... done.  
Loading package integer-gmp ... linking ... done.  
Loading package base ... linking ... done.  
Prelude> 1 + 2  
3  
Prelude> 3 + 4 * 5  
23  
Prelude> (3 + 4) * 5  
35  
Prelude>
```

Der Text hinter dem Prompt `Prelude>` wurde vom Benutzer getätigt, alles andere sind Ausgaben von GHCi. Der Prompt gibt



## GHCi

Der Interpreter wertet alle eingegebenen Ausdrücke aus. Mit den Pfeiltasten kann von vorherige Eingaben durchblättern

Zur Steuerung des Interpreters stehen Befehle zur Verfügung, welche alle mit einem Doppelpunkt beginnen, z.B. `:?` für die Hilfe.

Alle Befehle kann man abkürzen. So kann man den Interpreter sowohl mit `:quit` also auch mit `:q` verlassen. Für uneindeutige Abkürzungen gibt es voreingestellte Defaults.

Auch für GHCi macht es Sinn, Programmdefinitionen mit einem gewöhnlichen Texteditor in eine separate Datei speichern und dann in den Interpreter zu laden, um nicht immer alles neu eintippen zu müssen.

```
:l datei.hs -- lade Definition aus Datei datei.hs
:r          -- erneut alle offenen Dateien einlesen
```



# FALLSTRICKE

Haskell beachtet Groß-/Kleinschreibung!

Haskell ist “whitespace”-sensitiv: Veränderungen an Leerzeichen, Tabulatoren und Zeilenumbruch können Fehler verursachen!

## GRUNDSÄTZLICH GILT:

Beginnt die nächste Zeile in ...

- **Spalte weiter rechts:** vorheriger Zeile geht weiter
- **gleicher Spalte:** nächstes Element eines Blocks beginnt
- **Spalte weiter links:** Block beendet

## DARAUS FOLGT:

- Alle Top-level Definition müssen in gleicher Spalte beginnen
- Einrückung kann viele Klammern sparen
- Tabulatorweite in Editor und GHC muss übereinstimmen

Anstatt Einrückung können auch `{ }` und `;` benutzt werden.



# AUSDRÜCKE & WERTE

GHCi wertet Ausdrücke aus. Ein Ausdruck kann

- atomar sein, wie z.B. `1337`, `'a'`
- zusammengesetzt sein, wie z.B. `12+1`, `27 * (7 + 5)`

Ein Ausdruck kann also aus mehreren Unterausdrücken bestehen. Mit Klammern können wir explizit angeben, in welcher Reihenfolge ein Ausdruck aus seinen Unterausdrücken zusammengesetzt ist.

Ein Ausdruck hat (meistens) auch einen **Wert**; so ist `13` der Wert des Ausdrucks `12+1`.

Jeder korrekt gebildete Ausdruck besitzt einen **Typ**.  
Zum Beispiel: der Ausdruck `12+1` hat den Typ `Int`.  
Wir schreiben kurz `12+1 :: Int`



# TYPEN

Ein Typ ist eine Menge von Werten; so bezeichnet der Typ `Int` meistens die Menge der ganzen Zahlen von  $-2^{29}$  bis  $2^{29} - 1$ .

Haskell-Syntax:

- **Typnamen** beginnen immer mit **Großbuchstaben**
- **Typvariablen** beginnen immer mit **Kleinbuchstaben**

GHCI Befehl `:t` zeigt Typ eines Ausdrucks:

```
Prelude> :t 'a'  
'a' :: Char
```

`e :: A` gelesen als “Ausdruck `e` hat Typ `A`”

Mit dem Befehl `:set +t` wird der Typ jedes ausgewerteten Ausdrucks angezeigt, mit `:unset +t` stellt man das wieder ab.

⇒ dauerhaft einstellbar in `ghci.conf`





# WICHTIGE NUMERISCHE TYPEN

**INT** Ganze Zahlen (“fixed precision integers”),  
maschinenabhängig, mindestens von  $-2^{29}$  bis  
 $2^{29} - 1$ . Es wird nicht auf Überläufe geprüft.

**INTEGER** Ganze Zahlen beliebiger Größe  
“arbitrary precision integers”

**FLOAT** Fließkommazahlen mindestens nach IEEE standard  
“single-precision floating point numbers”

**DOUBLE** Fließkommazahlen mit mindestens doppelter  
Genauigkeit nach IEEE standard  
“double-precision floating point numbers”

**RATIONAL** Rationale Zahlen beliebiger Genauigkeit, werden mit  
dem Prozentzeichen konstruiert:  $1 \% 5 \approx 0.2$   
'%' nicht im Prelude-Modul enthalten

Haskell kennt viele weitere numerische Datentypen, z.B. komplexe  
Zahlen, Uhrzeiten oder Festkommazahlen.



# WICHTIGE TYPEN

Neben Zahlen gibt es noch weitere wichtige grundlegende Typen:

**BOOL** Boole'sche (logische) Wahrheitswerte: **True** und **False**

**CHAR** Unicode Zeichen, z.B. **'q'**. Diese werden immer in Apostrophen eingeschlossen.

**STRING** Zeichenketten, z.B. **"Hallo!"**. Diese werden immer in Anführungszeichen eingeschlossen.

Haskell ist so knapp wie nötig formuliert. Viele Dinge, welche in anderen Sprachen "speziell" oder "eingebaut" sind, werden in Haskell in einer gewöhnlichen Bibliothek definiert.

Von diesen drei Typen ist lediglich **Char** etwas grundlegender, die beiden anderen kann man leicht selbst definieren; doch dazu lernen wir später noch mehr.



# KARTESISCHES PRODUKT

Ein Beispiel für einen **zusammengesetzten Typen** hat als Vorlage aus der Mathematik das kartesische Produkt:

Aus den Mengen  $A_1$  und  $A_2$  Mengen bildet man als **kartesisches Produkt** die Menge

$$A_1 \times A_2 = \{(a_1, a_2) \mid a_1 \in A_1 \text{ und } a_2 \in A_2\}$$

Beispiel:  $(5, 8) \in \mathbb{N} \times \mathbb{N}$

## DEFINITION (KARTESISCHES PRODUKT)

Sind  $A_1, \dots, A_n$  Mengen, so ist das kartesische Produkt definiert als  $A_1 \times \dots \times A_n = \{(a_1, \dots, a_n) \mid a_i \in A_i \text{ für } i = 1 \dots n\}$

Die Elemente von  $A_1 \times \dots \times A_n$  heißen allgemein  **$n$ -Tupel**, spezieller auch Paare, Tripel, Quadrupel, Quintupel, Sextupel,...



## TUPEL

In Haskell schreiben wir das genau so hin, lediglich im Typ verwenden wir anstelle des Symbols  $\times$  ebenfalls Klammern und Kommas. Aus  $\mathbb{Z} \times \mathbb{Z}$  wird also `(Integer, Integer)`:

```
Prelude> :set +t
Prelude> (7,6)
(7,6)
it :: (Integer, Integer)
Prelude> (42, 'a')
(42, 'a')
it :: (Integer, Char)
Prelude> (True, 4.5, "Hi!")
(True, 4.5, "Hi!")
it :: (Bool, Double, String)
Prelude>
```

Zusammengesetzten Typen und deren Werte können wir ganz genauso wie jeden anderen Typen verwenden.



# LISTEN

Einer der wichtigsten Typen in Haskell sind Listen, also geordnete Folgen von Werten. Listen schreiben wir in eckigen Klammern, wobei die Elemente durch Kommas getrennt werden:

```
[1,2,3] :: [Int]
```

```
[1,2,2,3,3,3] :: [Int]
```

```
["Hello","World","!"] :: [String]
```

Im Gegensatz zu einem Tupel wissen wir anhand des Typen nicht, wie viele Werte eine Liste enthält. Eine Liste kann sogar ganz leer sein, geschrieben `[]`.

Allerdings wissen wir, dass alle Elemente einer Liste den gleichen Typ haben — diesen schreiben wir ebenfalls in eckigen Klammern um den Listentyp hinzuschreiben; Listen sind daher **homogene** Datenstrukturen.



## LISTEN

Listen und Tupel kann man beliebig ineinander verschachteln:

```
[(1, 'a'), (2, 'z'), (-4, 'w')] :: [(Integer, Char)]
```

```
[[1, 2, 3], [], [4]] :: [[Integer]]
```

```
(4.5, [(True, 'a', [5, 7], ())])  
:: (Double, [(Bool, Char, [Integer], ())])
```

Achtung: [] und [[]] und [[], []] und [[[]]] und [[], [[]]]  
sind alles verschiedene Werte.

Das 0-Tupel ist ebenfalls erlaubt: () :: ().

Der Typ () wird als **Unit**-Typ bezeichnet, und hat nur den  
einzigsten Wert (). Aus dem Kontext wird fast immer klar, ob mit  
() der Typ oder der Wert gemeint ist.



## LISTEN

Listen und Tupel kann man beliebig ineinander verschachteln:

```
[(1, 'a'), (2, 'z'), (-4, 'w')] :: [(Integer, Char)]
```

```
[[1, 2, 3], [], [4]] :: [[Integer]]
```

```
(4.5, [(True, 'a', [5, 7], ())])  
:: (Double, [(Bool, Char, [Integer], ())])
```

Achtung: `[]` und  `[[] ]` und  `[ [] , [] ]` und  `[ [ [] ] ]` und  `[ [] , [ [] ] ]`  
sind alles verschiedene Werte.

Das 0-Tupel ist ebenfalls erlaubt: `() :: ()`.

Der Typ `()` wird als **Unit**-Typ bezeichnet, und hat nur den einzigen Wert `()`. Aus dem Kontext wird fast immer klar, ob mit `()` der Typ oder der Wert gemeint ist.



## LISTEN

Listen und Tupel kann man beliebig ineinander verschachteln:

```
[(1, 'a'), (2, 'z'), (-4, 'w')] :: [(Integer, Char)]
```

```
[[1, 2, 3], [], [4]] :: [[Integer]]
```

```
(4.5, [(True, 'a', [5, 7], ())])  
:: (Double, [(Bool, Char, [Integer], ())])
```

Achtung: `[]` und  `[[] ]` und  `[ [] , [] ]` und  `[ [ [] ] ]` und  `[ [] , [ [] ] ]`  
sind alles verschiedene Werte.

Das 0-Tupel ist ebenfalls erlaubt: `() :: ()`.

Der Typ `()` wird als **Unit**-Typ bezeichnet, und hat nur den  
einzigsten Wert `()`. Aus dem Kontext wird fast immer klar, ob mit  
`()` der Typ oder der Wert gemeint ist.





# TYPABKÜRZUNGEN

Der Typ `String` ist nur eine Abkürzung für eine `Char`-Liste:

```
type String = [Char]
```

Solche Abkürzungen darf man genau so auch selbst definieren: Hinter dem Schlüsselwort `type` schreibt man einen neuen Namen, der mit einem Großbuchstaben beginnt und hinter dem Gleichheitszeichen folgt ein bekannter Typ, z.B.

```
type MyWeirdType = (Double, [(Bool, Integer)])
```

Für die Ausführung von Programmen ist dies unerheblich.

Typabkürzungen dienen primär zur Verbesserung der Lesbarkeit.

Leider ignoriert GHC/GHCi Typabkürzungen meistens, d.h. GHCi gibt fast immer `[Char]` anstelle von `String` aus.

Es gibt noch weitere zusammengesetzte Typen, z.B. Records, Funktionstypen, etc.



# FUNKTIONEN

## DEFINITION (FUNKTION)

Für zwei Mengen  $A$  und  $B$  ist eine (totale) **Funktion**  $f$  von  $A$  nach  $B$  eine Zuordnung, welche jedem Element  $x \in A$  genau ein Element  $y \in B$  zuordnet; geschrieben  $f(x) = y$ .

Die Menge  $A$  bezeichnen wir als den **Definitionsbereich** von  $f$ , die Menge  $B$  als **Zielbereich**.

Für eine Funktion  $f$  von  $A$  nach  $B$  schreiben wir kurz:  $f : A \rightarrow B$ .

Funktionsanwendung wird auch gerne ohne Klammer in **Präfixnotation**  $f x$  geschrieben, die **Postfixnotation**  $x f$  findet dagegen seltener Verwendung.

Den Definitionsbereich einer Funktion  $f$  (engl. Domain) bezeichnen wir üblicherweise mit  $\text{dom}(f)$ .



# BEISPIEL: FUNKTIONEN

## TOTALE FUNKTIONEN

- Minus :  $\mathbb{Z} \rightarrow \mathbb{Z}$
- Nachfolgerfunktion  $f\ x = x + 1 : \mathbb{N} \rightarrow \mathbb{N}$
- Signumfunktion

$$\text{sgn } arg = \begin{cases} +1 & \text{falls } x > 0 \\ 0 & \text{falls } x = 0 \\ -1 & \text{falls } x < 0 \end{cases} : \mathbb{R} \rightarrow \{-1, 0, +1\}$$

## PARTIELLE FUNKTIONEN

- Wurzelfunktion  $\sqrt{\cdot} : \mathbb{R} \rightarrow \mathbb{R}$  mit Definitionsbereich  $\mathbb{R}^+$
- Kehrwert  $kw\ x = 1/x : \mathbb{R} \rightarrow \mathbb{R}$  mit Definitionsbereich  $\mathbb{R} \setminus \{0\}$

- $foo\ x = \begin{cases} 2x & x > 1 \\ 0 & x < 1 \end{cases} : \mathbb{N} \rightarrow \mathbb{N}$

mit Definitionsbereich  $\mathbb{N} \setminus \{-1, 0, +1\}$



# PARTIELLE FUNKTIONEN

## DEFINITION (PARTIELLE FUNKTION)

Eine **partielle Funktion**  $f : A \rightarrow B$  ordnet nur einer Teilmenge  $A' \subset A$  einen Wert aus  $B$  zu, und ist ansonsten undefiniert. In diesem Falle bezeichnen wir  $A$  als **Quellbereich** und  $A'$  als Definitionsbereich.

Ob eine Funktion partiell oder total ist, kann man nicht immer leicht feststellen.

Wir sprechen von partiellen Funktionen, wenn bei der Berechnung einer Funktion für ein bestimmtes Argument ein Fehler auftritt



# BEISPIEL: FUNKTIONEN

## TOTALE FUNKTIONEN

- Minus :  $\mathbb{Z} \rightarrow \mathbb{Z}$
- Nachfolgerfunktion  $f\ x = x + 1 : \mathbb{N} \rightarrow \mathbb{N}$
- Signumfunktion

$$\text{sgn } arg = \begin{cases} +1 & \text{falls } x > 0 \\ 0 & \text{falls } x = 0 \\ -1 & \text{falls } x < 0 \end{cases} : \mathbb{R} \rightarrow \{-1, 0, +1\}$$

## PARTIELLE FUNKTIONEN

- Wurzelfunktion  $\sqrt{\cdot} : \mathbb{R} \rightarrow \mathbb{R}$  mit Definitionsbereich  $\mathbb{R}^+$
- Kehrwert  $\text{kw } x = 1/x : \mathbb{R} \rightarrow \mathbb{R}$  mit Definitionsbereich  $\mathbb{R} \setminus \{0\}$

- $\text{foo } x = \begin{cases} 2x & x > 1 \\ 0 & x < 1 \end{cases} : \mathbb{N} \rightarrow \mathbb{N}$

mit Definitionsbereich  $\mathbb{N} \setminus \{-1, 0, +1\}$



# ANONYME FUNKTIONEN

Merkwürdig:  $\sqrt{\cdot} :: \mathbb{R} \rightarrow \mathbb{R}$  Was bedeutet der Punkt?

Es ist oft praktisch, simplen Funktionen keinen besonderen Namen zu geben, bzw. die Wurzelfunktion hat ja bereits ein eigenes Symbol.

Aus dem Lambda-Kalkül (Alonzo Church, 1936), der mathematischen Grundlage der funktionalen Programmierung, nehmen wir daher die  $\lambda$ -Notation für anonyme Funktionen:

Nachfolgerfunktion	$\lambda x . x + 1$
Wurzelfunktion	$\lambda y . \sqrt{y}$

In Haskell schreiben wir anstelle von  $\lambda$  einfach einen Rückstrich `\`

```
\x -> x + 1  
\y -> sqrt y
```



# FUNKTIONEN MIT MEHREREN ARGUMENTEN

Eine Funktion, deren Quellbereich ein  $n$ -stelliges kartesisches Produkt ist, nennt man eine Funktion der **Stelligkeit**  $n$ , oder auch  **$n$ -stellige** Funktion “ $n$ -ary function”, “arity”.

$$f :: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$
$$f(x, y) = x + 2y^2$$

Für die Anwendung zweistelliger Funktionen wird öfters die **Infixnotation** verwendet:  $xfy$

Bekannte Beispiele für zweistellige Funktionen mit gebräuchlicher Infixnotation sind  $+$ ,  $-$ ,  $*$ ,  $/$ .



# FUNKTIONEN MIT MEHREREN ARGUMENTEN

Eine Funktion, deren Quellbereich ein  $n$ -stelliges kartesisches Produkt ist, nennt man eine Funktion der **Stelligkeit**  $n$ , oder auch  **$n$ -stellige** Funktion “ $n$ -ary function”, “arity”.

$$f :: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$f(x, y) = x + 2y^2$$

Für die Anwendung zweistelliger Funktionen wird öfters die **Infixnotation** verwendet:  $xfy$

Bekannte Beispiele für zweistellige Funktionen mit gebräuchlicher Infixnotation sind  $+$ ,  $-$ ,  $*$ ,  $/$ .





# FUNKTIONEN MIT MEHREREN ARGUMENTEN

Eine Funktion, deren Quellbereich ein  $n$ -stelliges kartesisches Produkt ist, nennt man eine Funktion der **Stelligkeit**  $n$ , oder auch  **$n$ -stellige** Funktion “ $n$ -ary function”, “arity”.

$$f :: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$
$$f(x, y) = x + 2y^2$$

Für die Anwendung zweistelliger Funktionen wird öfters die **Infixnotation** verwendet:  $xfy$

Bekannte Beispiele für zweistellige Funktionen mit gebräuchlicher Infixnotation sind  $+$ ,  $-$ ,  $*$ ,  $/$ .



# FUNKTIONEN IN HASKELL

In Haskell schreiben wir Funktion ganz natürlich hin:

```
succ :: Int -> Int  
succ x = x + 1
```

Wir definieren eine Funktion mit dem Namen `succ`, welche eine ganze Zahl auf Ihren Nachfolger abbildet.

Den Namen des Argumentes, hier `x`, können wir frei wählen.

“Sprechende” Namen empfehlen sich.

Die erste Zeile, also die Angabe der **Typsignatur** der Funktion ist optional, da GHC den Typ auch automatisch inferieren kann.

Man sollte immer explizite Typsignaturen hinschreiben, denn...

- ... Typsignaturen sind eine hilfreiche Dokumentation
- ... helfen dabei, Fehlermeldungen leserlicher zu machen
- ... können Programme manchmal effizienter machen



# MEHRSTELLIGE FUNKTIONEN

Mehrstellige Funktionen können wir auf zwei Arten definieren:

## 1-STELLIGE FUNKTION MIT KARTESISCHEM PRODUKT

```
foo :: (Int, Int) -> Int
foo (x, y) = (x + 1) * y
```

Hier verwenden wir ganz einfach Tupel

## ECHTE N-STELLIGE FUNKTION

```
bar :: Int -> (Int -> Int)
bar x y = (x + 1) * y
```

Hier schreiben wir einfach alle Argumente durch Leerzeichen getrennt hintereinander.

Die zweite Variante wird oft bevorzugt; doch da eine Funktion immer nur **einen einzigen** Wert zurück liefern kann, bietet es sich manchmal an, Tupel zu verwenden.

Man kann auch leicht wechseln, Stichwort “Currying”.



# FUNKTIONSANWENDUNG

Die Funktionsanwendung verwendet in Haskell primär die Präfixnotation, d.h. man schreibt hinter dem Namen der Funktion einfach alle Argumente, durch Leerzeichen getrennt:

```
Prelude> succ 5  
6
```

```
Prelude> foo (2,7)  
21
```

```
Prelude> bar 1 8  
16
```

Dies kann man auch mithilfe von Klammern verschachteln:

```
Prelude> bar (succ 5) 2  
14
```



# INFIXNOTATION IN HASKELL

Haskell bietet ebenfalls die Infixnotationen an, um die Lesbarkeit des Codes zu erhöhen.

Der Programmierer gibt bei der Funktionsdefinition an, welche Notation er bevorzugt:

```
infixl 6  +  
(+) :: Int -> Int  
(+) x
```

```
succ :: Int -> Int  
succ x = x + 1
```

`infixl` oder `infixr` und die Zahl dahinter zeigen Haskell, in welcher Reihenfolge mehrere Unterausdrücke ohne explizite Klammerung zu lesen sind.

So wird zum Beispiel die Punkt-vor-Strich Regel eingehalten:

```
infixl 7  *
```



# INFIXNOTATION IN HASKELL

Zwischen Infix- und Präfixnotation kann man jederzeit auch ad hoc wechseln:

- Mit Klammern macht man aus einer Infix-Funktion eine gewöhnliche Präfix-Funktion:

```
Prelude> (+) 1 2  
3
```

- Mit Rückstrichen “backquotes” macht man aus einer Funktion in Präfixnotation eine Infix-Funktion:

```
Prelude> 1 'bar' 8  
16
```

Man verwendet die Notation, welche die Lesbarkeit erhöht.



# FUNKTIONSTYPEN

Der Typ einer Funktion ist ein zusammengesetzter Funktionstyp, der immer aus genau zwei Typen mit einem Pfeil dazwischen besteht.

## KLAMMERKONVENTION

Funktionstypen sind implizit rechtsgeklammert, d.h. man darf die Klammern manchmal weglassen:

**Int** -> **Int** -> **Int** wird gelesen als **Int** -> (**Int** -> **Int**)

Entsprechend ist die Funktionsanwendung implizit linksgeklammert:

bar 1 8 wird gelesen als (bar 1) 8

Das bedeutet: (bar 1) ist eine Funktion des Typs **Int** -> **Int**!  
Funktionen sind also normale Werte in einer funktionalen Sprache!



# ZUSAMMENFASSUNG

- Haskell ist eine universelle Programmiersprache
- “Haskell Platform” installieren, nicht nur GHC
  - Anzeige von Typen ggf. in `ghci.conf` dauerhaft aktivieren
  - `ghc/ghci` mit Option `+RTS -M1g` starten, um Speicherverbrauch zu begrenzen
- Haskell ist White-Space und Case sensitiv:
  - Einrückung beachten!
  - Editor sollte Tabulatoren in Leerzeichen wandeln, oder Tabulator-weite von Editor und GHC muss übereinstimmen
- Interpreter Befehle beginnen mit Doppelpunkt, z.B. `:quit`
- Funktionen bekommen immer nur ein Argument;  
Mehrstellige Funktionen:
  - **Entweder** Produkte von Argumenten als  $n$ -Tupel **oder**
  - Ergebnis ist Funktion, welche nächstes Argument verarbeitet

