Technische Universität Braunschweig

Programmieren für Fortgeschrittene - eine Einführung in Haskell

Tag eins - Was ist Haskell

Stephan Mielke

24.03.2014

Organisatorisches

Organisatorisches

- Jeden Wochentag vom Mo, 24.03.2014 bis Fr, 28.03.2014 im Raum IZ 261.
- 10:00 bis 16:00 Raumreservierung
- 10:00 bis 11:30 theoretische Einführung in Haskell
- 11:30 bis 12:30 Mittagspause
- 12:30 bis ca. 15:30 praktische Übungen

Organisatorisches

- Theorie (jetzt)
  Umsetzung des Stoffs der Vorlesung in Haskell
- Übungen (nachher)
  praktisches Arbeiten mit Haskell über den GHCi

Quellen

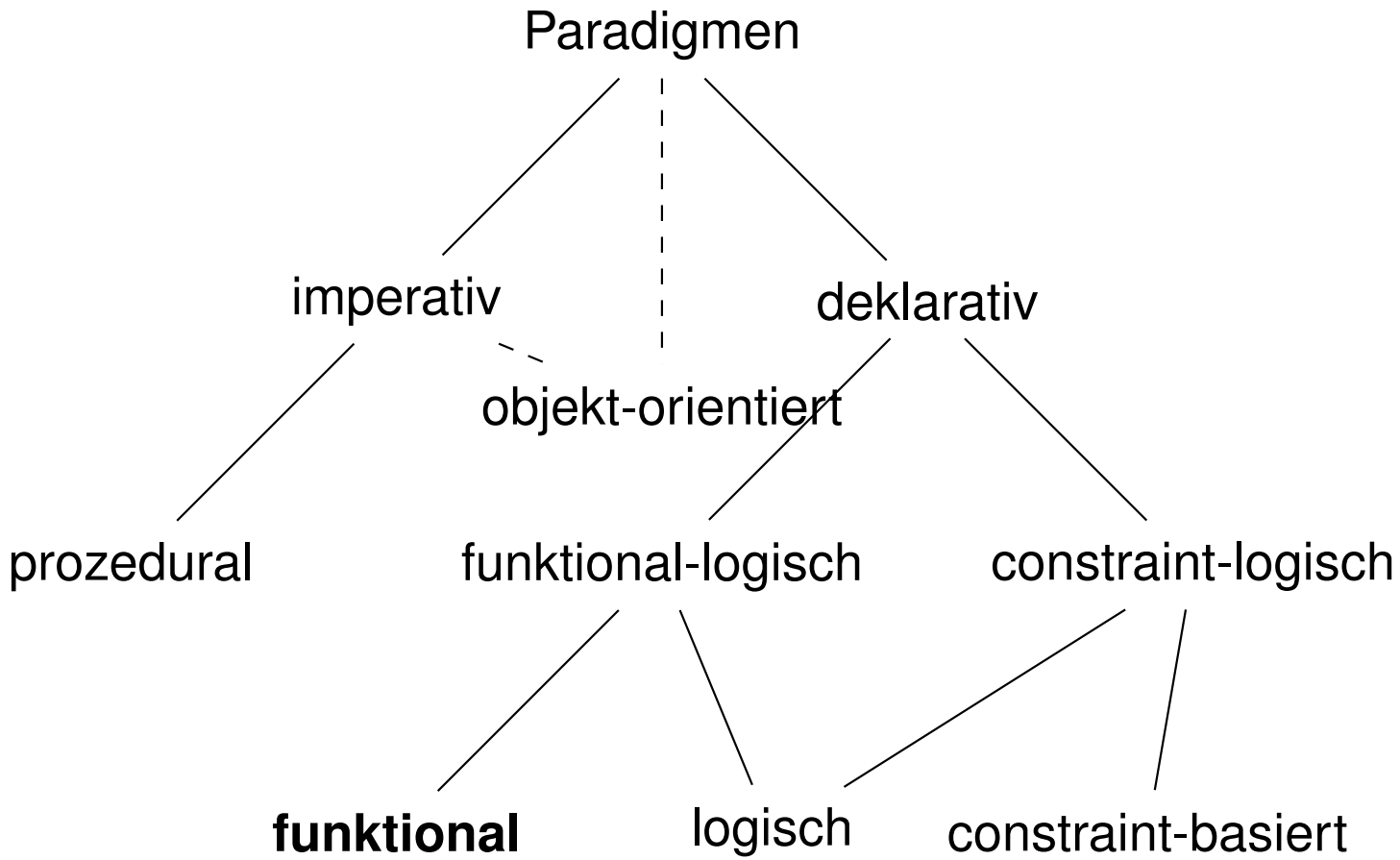
- Algorithmieren und Programmieren
  Vorlesung von Prof. Dr. Petra Hofstedt (BTU)
- Moderne Funktionale Programmierung
  Vorlesung von Prof. Dr. Petra Hofstedt (BTU)
- Eine Einführung in die funktionale Programmierung mit Haskell
  Übungsskript zu unserer Vorlesung
- Haskell - Intensivkurs

Das funktionale Paradigma

Paradigmen

- Programmierparadigma
  generelle Sicht bei der Modellierung und Lösung eines Problems
- Klassische Unterscheidung
  - Imperative Sprachen
    „**Wie**“ findet die Lösung statt
    Folge von Anweisungen zur Problemlösung
  - Deklarative Sprachen
    „**Was**“ ist die Lösung
    deklarative Beschreibung der Lösung bzw. des Problems

Paradigmen


Paradigmen
imperativ
deklarativ
objekt-orientiert
prozedural
funktional-logisch
constraint-logisch
funktional
logisch
constraint-basiert

Funktionale Paradigma

- Hohes Abstraktionsniveau
  Klare Darstellung der Programmiertechniken und Algorithmen, d.h. Konzentration auf die Konzepte statt auf die Sprache.
- Klare, elegante und kompakte Programme
  kurze Entwicklungszeiten, lesbare Programme.
- Keine Seiteneffekte
  erleichtert Verstehen, Optimierung, Verifikation.
- Saubere theoretische Fundierung
  ermöglicht Verifikation und erleichtert formale Argumentation über Programme.

Haskell

Haskell

- 1990 als Haskell 1.0 veröffentlicht
- Aktuelle Version Haskell 2010
- An Haskell 2014 (Preview) wird „gearbeitet“

Hello World

```
module Main where

main :: IO ()
main = putStrLn "Hello, World!"
```

Hello World

```
module Main where

main :: IO ()
main = putStrLn "Hello, World!"
```

Ausgabe

Hello, World!

Hello World

```
module Main where

main :: IO ()
main = putStrLn "Hello, World!"
```

Ausgabe

Hello, World!

weiteres in den Übungen

Haskell Compiler

- Hugs (Haskell User's Gofer System)
  implementiert Haskell 98
  seit ca 6 Jahren nicht weiterentwickelt
- Yhc (York Haskell Compiler)
  implementiert Haskell 98
  Projekt eingestellt
- GHC (Glasgow Haskell Compiler)
  implementiert Haskell 98 / 2010
  weit verbreitster Haskell Compiler
  besitzt den GHCi als Haskell Interpreter
  in den Übungen werden wir hauptsächlich mit dem Interpreter arbeiten

Glasgow Haskell Compiler

- Original Prototyp ’89 in LML (Lazy ML)
- Bei der Entwicklung von Haskell in Haskell neu geschrieben (’89)
- Nur kleine Teile in C bzw. C– (C verwandte Sprache zur Nutzung als Zwischencode)
- Erweitert den Haskell Standard um noch nicht standardisierte Erweiterungen
- Plattform und Architektur unabhängig

Laufzeitumgebung

- Wenn das Programm in Maschinencode übersetzt wurde, wird keine externe Laufzeitumgebung benötigt (nativer Code)
  die „Laufzeitumgebung“ wird mit in das Programm gepackt
- Bei Benutzung des Interpreters wird dieser als Laufzeitumgebung verwendet.

Interne Funktionsweise

- Erzeugung von Zwischencode „C–“
- C– ist wie C-Code jedoch „etwas“ anders
- Dieser Code wird optimiert und weiter compiliert
- „-fasm“ erzeugt Maschinencode (Standard)
- „-fvia-C“ erzeugt C-Code aus C–
  seit Version 7.0 nicht mehr unterstützt
- „-fllvm“ nutzt den LLVM als Backend-Compiler

Interne Funktionsweise

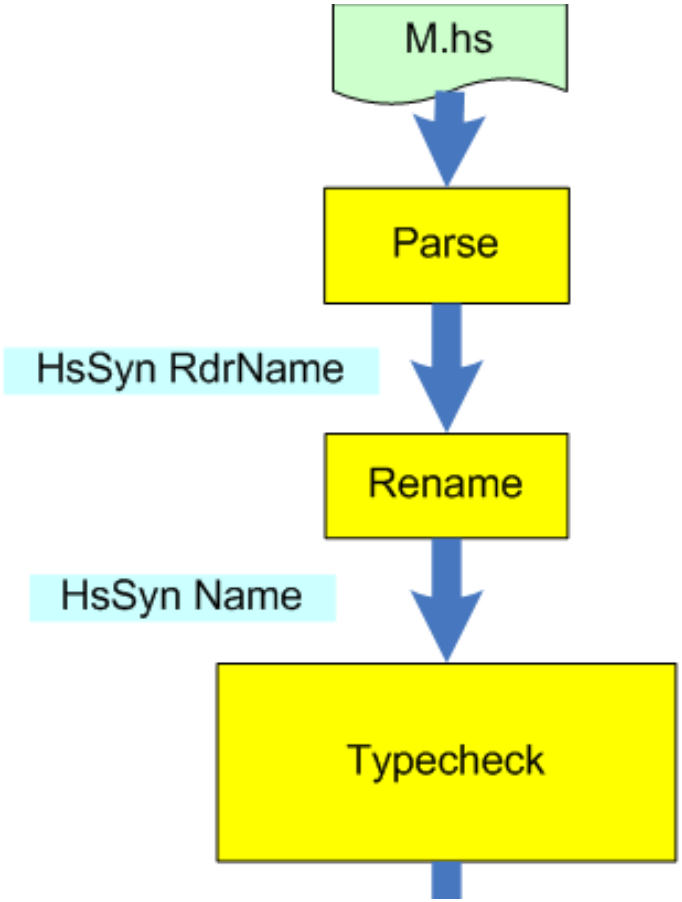


Abbildung: Compiler Teil 1 ©haskell.org

Interne Funktionsweise

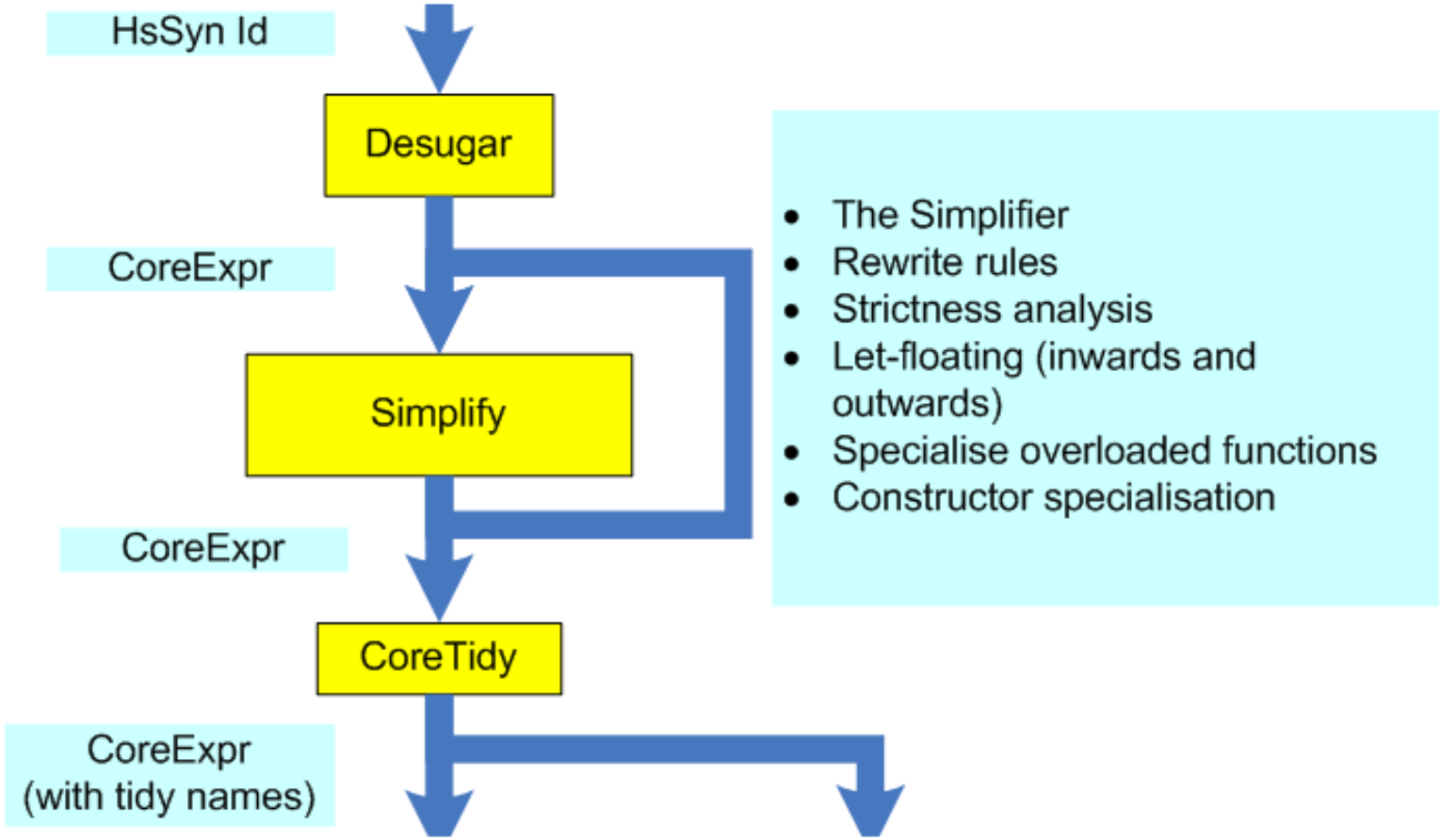


Abbildung: Compiler Teil 2 ©haskell.org

Interne Funktionsweise

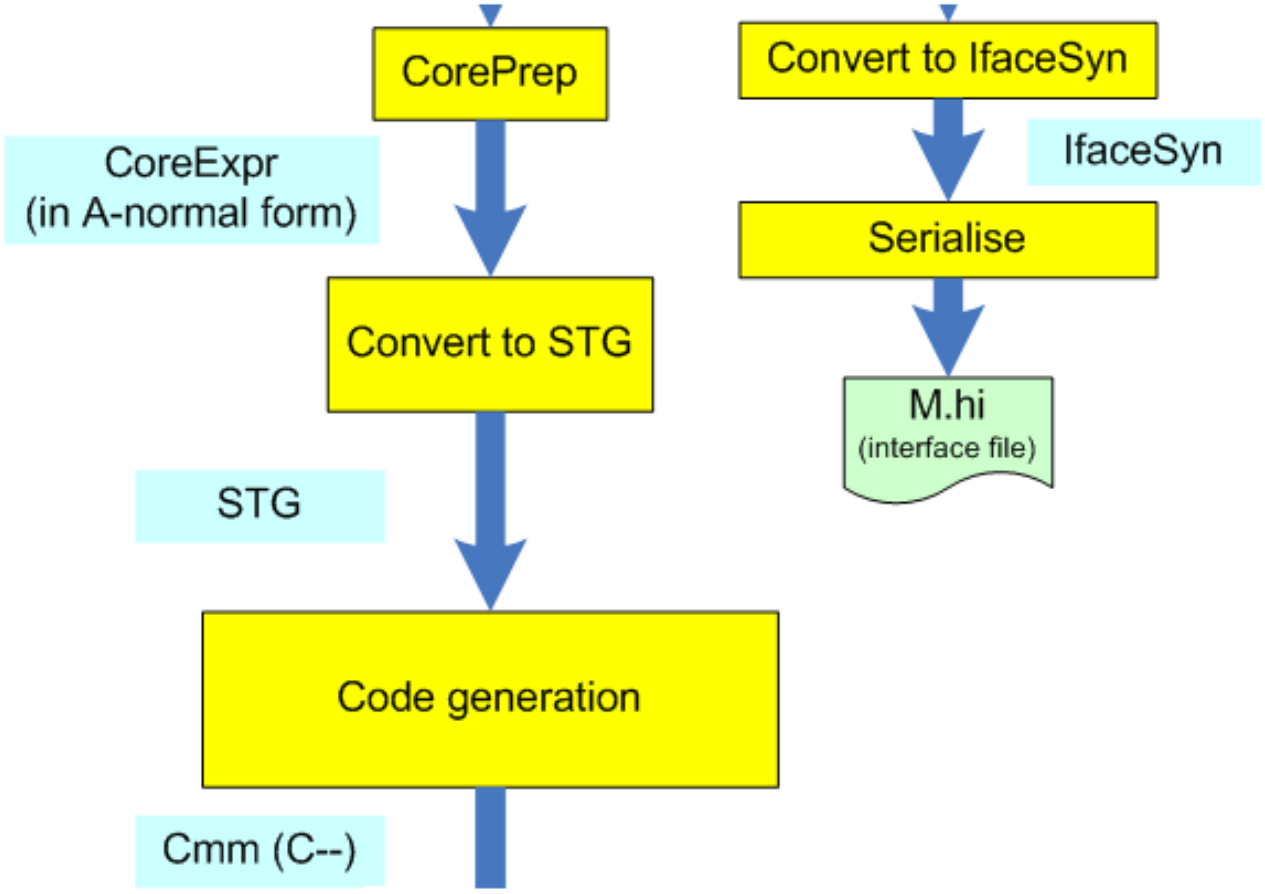


Abbildung: Compiler Teil 3 ©haskell.org

Interne Funktionsweise

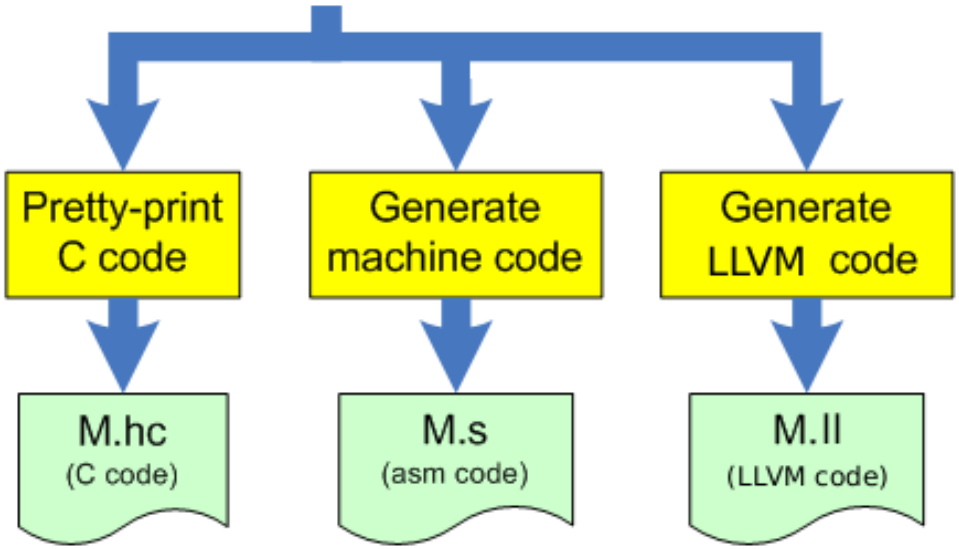


Abbildung: Compiler Teil 4 ©haskell.org

Semantische Grundbegriffe

Semantische Grundbegriffe - Namen und Attribute

Namen und Attribute

- Alle endlichen ASCII-Strings außer:
  case, class, data, default, deriving, do, else, if, import, in, infix, infixl, infixr, instance, let, module, newtype, of, then, type, where
- Bezeichner sind case-sensitiv. (pLus $\neq$ plus)
- _ (Unterstrich) ist der Platzhalter
- Module beginnen mit einem Großbuchstaben
- Funktionen mit einem Kleinbuchstaben

Semantische Grundbegriffe - Variablen und Konstanten

Variablen

- Globale Variablen existieren nicht
- Lokale Variablen existieren nur in Funktionen als Teilergebnis

Konstanten

Konstanten sind Funktionen ohne Parameter

Semantische Grundbegriffe - Ausdrücke

Ausdrücke

elementare Ausdrücke bzw. Grundterme setzten sich zusammen aus:

- Konstanten wie z.B. Zahlen (10, 9.8), Zeichen ('a', 'Z'), ...
- Andere Funktionen sin, +, ∗, ...

Infixnotation

Funktionszeichen: 3 + 4 ≡ (+) 3 4

Funktionsname: mod 100 4 ≡ 100 `mod` 4

Ausdrücke

- Elementare Ausdrücke mit Variablen sind Ausdrücke bzw. Terme
- Durch einen „Vorspann“ wie x wird die Variable x mit der λ-Notation „gebunden“
- $\lambda a \to \lambda b \to a + b$ ist ein λ-Ausdruck
- λ ist kein ASCII Zeichen, deswegen wird „\“ verwendet

Ausdrücke

Elementarer Ausdruck

```
plus = 10 + 30
```

Ausdruck

```
plus' a b = a + b
```

Lambda (λ)-Ausdruck

```
plus'' = \\a -> \\b -> a + b
```

Morgen kommt mehr zum Thema λ-Ausdrücke

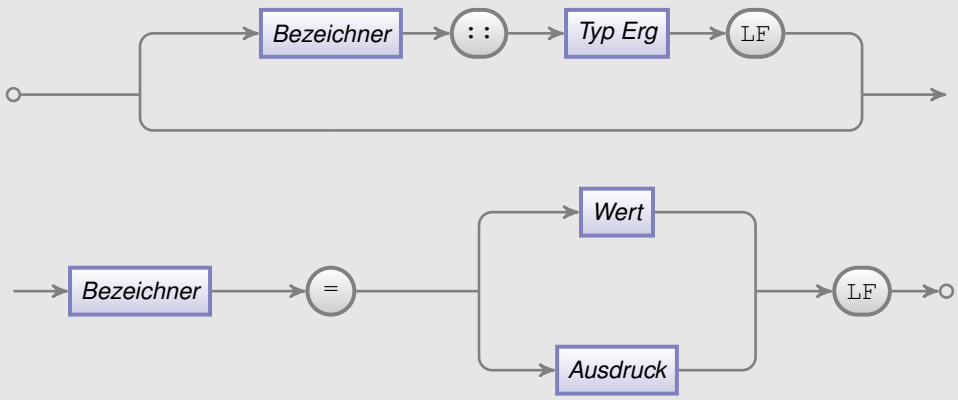
Semantische Grundbegriffe - Funktionen

Deklaration von Funktionen

- Funktion f ist ein Tripel (D_f, W_f, R_f)
- D_f Definitionsmenge
- W_f Wertemenge
- $R_f \subseteq D_f \times W_f$
- R_f muss **rechtseindeutig** sein d.h. es gibt keine zwei Paare $(a, b1) \in R_f$ und $(a, b_2) \in R_f$ mit $b_1 \neq b_2$
- Somit gilt, eine Funktion f bildet den Argumentwert x in den Resultatwert y ab

Deklaration von Funktionen

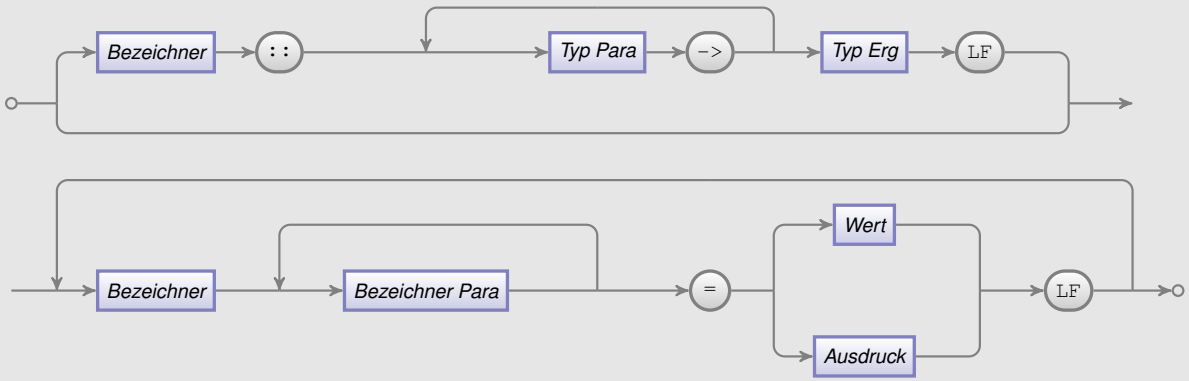
Konstante



- Funktionsköpfe sind optional, jedoch empfohlen
- Funktionsnamen beginnen mit Kleinbuchstaben
- Parameter von Funktionen beginnen mit Kleinbuchstaben

Deklaration von Funktionen

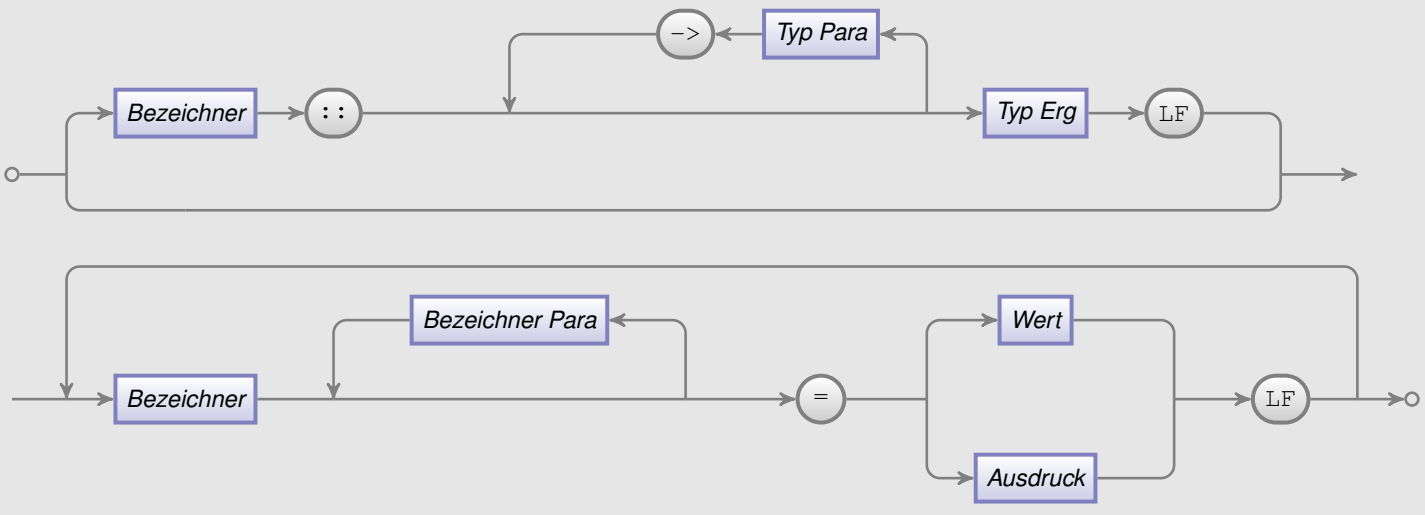
Funktion



- Funktionsköpfe sind optional, jedoch empfohlen
- Funktionsnamen beginnen mit Kleinbuchstaben
- Parameter von Funktionen beginnen mit Kleinbuchstaben

Deklaration von Funktionen

Allgemein



Deklaration von Funktionen

Konstante

```
eins :: Int
eins  = 1
```

Deklaration von Funktionen

Konstante

```
eins :: Int
eins  = 1
```

Unäre Funktion

```
successor :: Int -> Int
successor     a  = a + 1
```

Deklaration von Funktionen

Konstante

```
eins :: Int
eins  = 1
```

Unäre Funktion

```
successor :: Int -> Int
successor     a  = a + 1
```

Binäre Funktion

```
nimmDenZweiten :: Int -> Int -> Int
nimmDenZweiten  _     b    = b
```

Funktionen vs. Operatoren

- Funktionen besitzen einen Namen aus Buchstaben
- Operatoren besitzen einen Namen aus Zeichen
- Funktionen binden stärker als Operatoren (Standard)
- Operatoren werden wie Funktionen deklariert

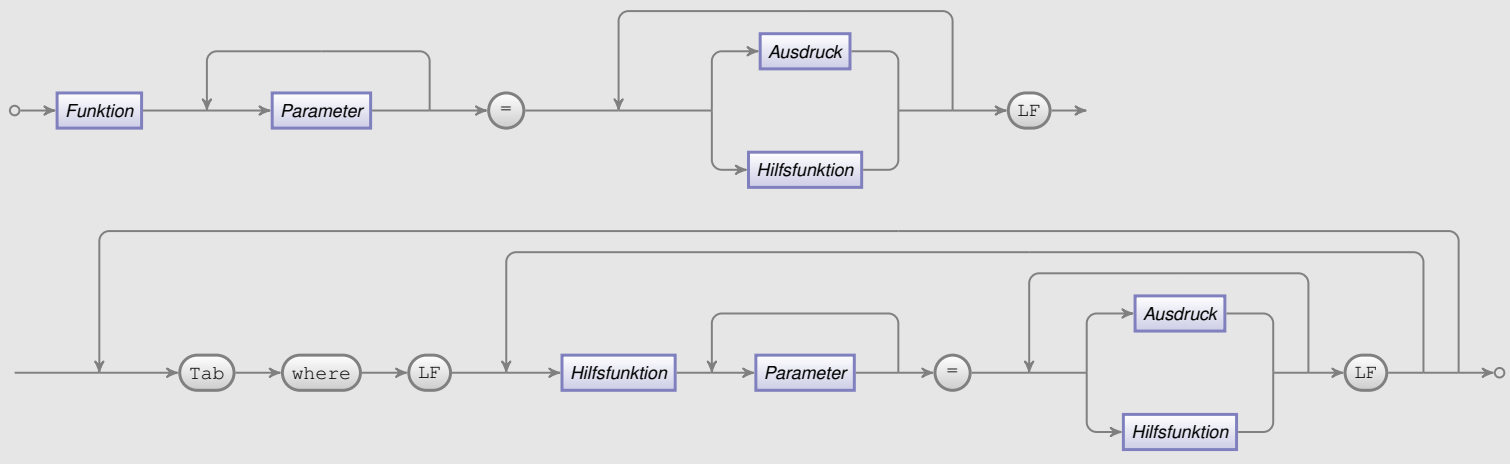
Semantische Grundbegriffe - Blöcke

Der where Block

- Zur nachträglichen Definition von internen Hilfsfunktionen (Teilfunktionen)
- Verschachtelung erlaubt
- Definiert für die ganze Funktion
- „Funktionsköpfe“ erlaubt

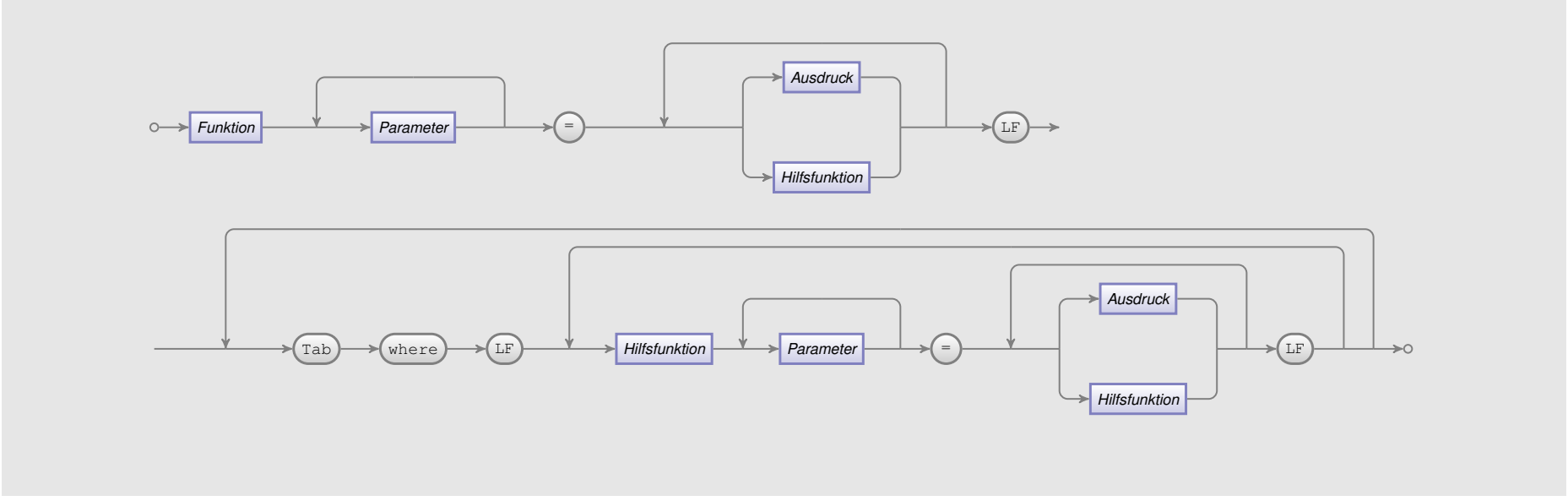
Der where Block

- Zur nachträglichen Definition von internen Hilfsfunktionen (Teilfunktionen)
- Verschachtelung erlaubt
- Definiert für die ganze Funktion
- „Funktionsköpfe“ erlaubt



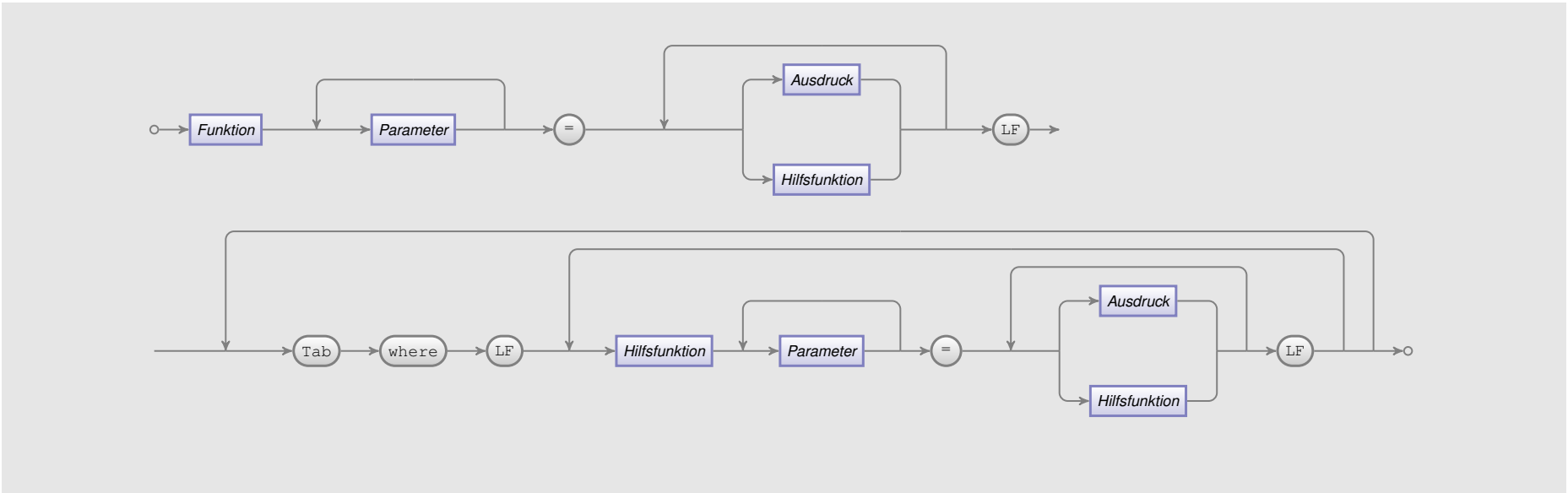
Der where Block

```
dec a = inc a - 2
    where
        inc a = a + 1
```



Der where Block

```
f :: Int -> Int
f   a    = x a `div` 3
    where x b = y b * 2
          where y b = a + b + 1
```



Der where Block

```
f :: Int -> Int
f a = x a `div` 3
  where x b = y b * 2
        where y b = a + b + 1
```

Aufruf

```
f 4
```

Der where Block

```
f :: Int -> Int
f a = x a `div` 3
  where x b = y b * 2
        where y b = a + b + 1
```

Aufruf

```
f 4
```

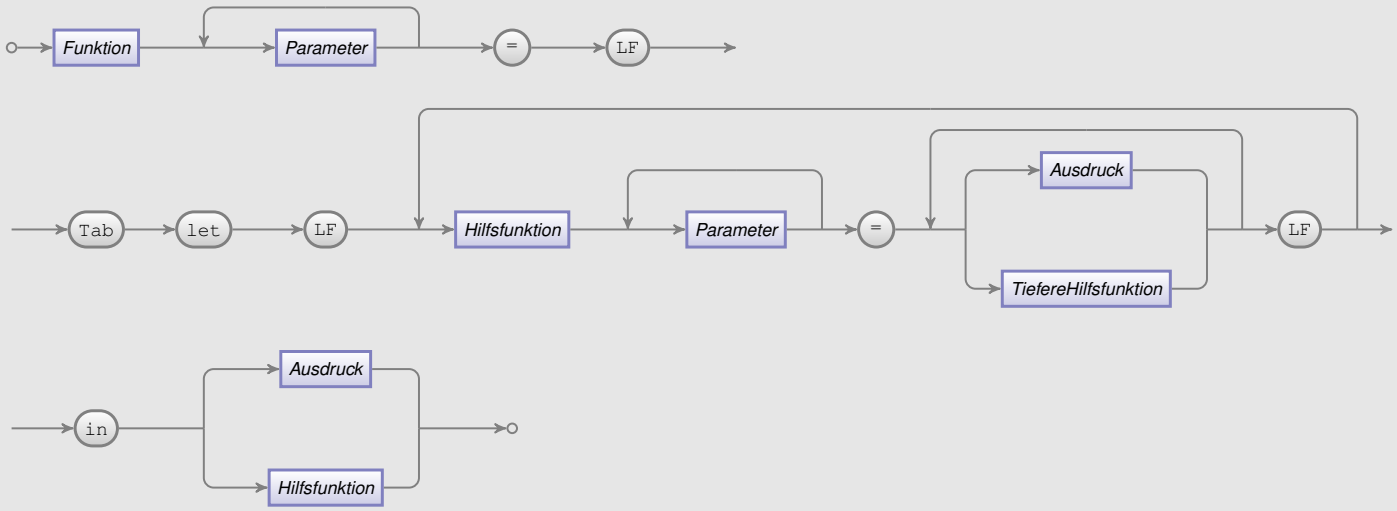
Ausgabe

```
6
```

Der let-in Block

- Zur vorherigen Definition von internen Hilfsfunktionen (Teilfunktionen)
- Kann auch zur Definition von Funktionen im Interpreter verwendet werden
- Verschachtelung erlaubt
- Definiert für den Funktionsabschnitt

Der let-in Block



Blöcke mit let-in können verschachtelt sein

Bei mehr als einer Hilfsfunktion, muss nach dem let ein Zeilenumbruch erfolgen.

Der let-in Block

```
dec a =
  let
      inc1 a = a + 1
      inc2 a = a + 2
  in  inc1 a - inc2 0
```

Zur Definition von Funktionen direkt im GHCi

```
let plus :: Int -> Int -> Int; plus a b = a + b
```

Der let-in Block

```
dec a =
  let
      inc1 a = a + 1
      inc2 a = a + 2
  in  inc1 a - inc2 0
```

Aufruf

```
dec 42
```

Der let-in Block

```
dec a =
  let
      inc1 a = a + 1
      inc2 a = a + 2
  in  inc1 a - inc2 0
```

Aufruf

```
dec 42
```

Ausgabe

```
41
```

Der let-in Block

```
outer a =
    let mid b =
        let inner c = c + 1
        in inner b + 2
    in mid a + 3
```

Aufruf

```
outer 42
```

Der let-in Block

```
outer a =
    let mid b =
        let inner c = c + 1
        in inner b + 2
    in mid a + 3
```

Aufruf

```
outer 42
```

Ausgabe

```
48
```

Einfache Datentypen

Einfache Datentypen

- Bool
- Int
- Integer
- Float
- Double
- Char

Einfache Datentypen - Warheitswerte

Bool

- Einfacher Wahrheitswert
- True oder False
- not $\equiv$ Verneinung
- && (binär), and (Liste) $\equiv$ und
- || (binär), or (Liste) $\equiv$ oder
- == $\equiv$ gleich
- /= $\equiv$ ungleich

Bool

```
myAnd :: Bool -> Bool -> Bool
myAnd True True  = True
myAnd _    _     = False

myOr :: Bool -> Bool -> Bool
myOr False False = False
myOr _     _     = True
```

Einfache Datentypen - Ganzzahlen

Int

- 32 Bit Ganzzahl (Architektur abhängig)
- Min = $-2^{31} = -2147483648$
- Max = $2^{31} - 1 = 2147483647$
- Zirkulär $(2^{31} - 1) + 1 = -2^{31}$

Achtung

Int ist nicht gleich Integer!

Integer

- Unbegrenzte Ganzzahl (RAM Größe ist die „Begrenzung“)
- Bei unendlich Arbeitsspeicher wirklich unbegrenzt

Int vs Integer

```
plus :: Int -> Int -> Int
plus a b = a + b
```

Aufruf

plus 2147483647 1

Int vs Integer

```
plus :: Int -> Int -> Int
plus a b = a + b
```

Aufruf

```
plus 2147483647 1
```

Ausgabe

```
-2147483648
```

Int vs Integer

```
plus' :: Integer -> Integer -> Integer
plus' a b = a + b
```

Aufruf

```
plus' 9876543210 9876543210
```

Int vs Integer

```
plus' :: Integer -> Integer -> Integer
plus' a b = a + b
```

Aufruf

```
plus' 9876543210 9876543210
```

Ausgabe

```
19753086420
```

Int vs Integer

```
plus' :: Integer -> Integer -> Integer
plus' a b = a + b
```

Aufruf

```
plus' 99999999999999999 99999999999999999
```

Int vs Integer

```
plus' :: Integer -> Integer -> Integer
plus' a b = a + b
```

Aufruf

```
plus' 9999999999999999999 9999999999999999999
```

Ausgabe

```
19999999999999999998
```

Int vs Integer

```
id :: Int -> Integer
id a = a

id' :: Integer -> Int
id' a = a
```

Geht nicht

Auch wenn Int für uns eine Teilmenge von Integer ist.

Int vs Integer

```
plus :: Integer -> Int -> Integer
plus a 0 = a
plus a b = plus (a + 1) (b - 1)

plus' :: Int -> Integer -> Int
plus' a 0 = a
plus' a b = plus' (a + 1) (b - 1)
```

Geht

Jedoch hat dies nichts mit interner Typkompatibilität zu tun.

Einfache Datentypen - Gleitkommazahl

Float - Double

- Float 32 Bit Gleitkommazahl
- Double 64 Bit Gleitkommazahl
- Float und Double sind ebenfalls inkompatibel zueinander wie Int und Integer

Einfache Datentypen - Zeichen

Char

- Stellt jedes Zeichen des Unicode (ISO 10646) da
- Geordnet nach der Reihenfolge des Auftretens

Ausblick für Morgen

Currying - allgemein

Gültigkeitsbereiche

Überladung und Auflösung von Namen

Ausdrücke

Technische Universität Braunschweig

Programmieren für Fortgeschrittene - eine Einführung in Haskell

Tag zwei - etwas mehr

Stephan Mielke

25.03.2014

Currying - allgemein

Deklaration von Funktionen in λ Notation

Lambda Currying

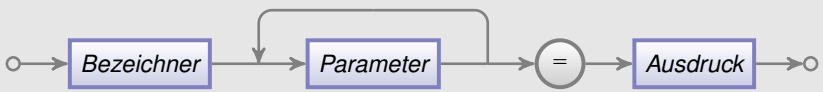
$$f = \lambda x_1 \rightarrow \lambda x_2 \rightarrow \ldots \lambda x_n \rightarrow e$$

Deklaration von Funktionen in λ Notation

Lambda Currying

$$f = \lambda x_1 \rightarrow \lambda x_2 \rightarrow \ldots \lambda x_n \rightarrow e$$

Funkion Currying

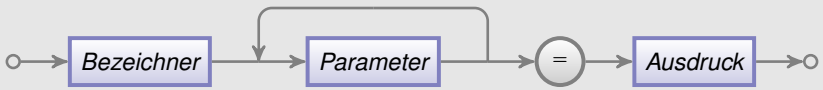


Deklaration von Funktionen in λ Notation

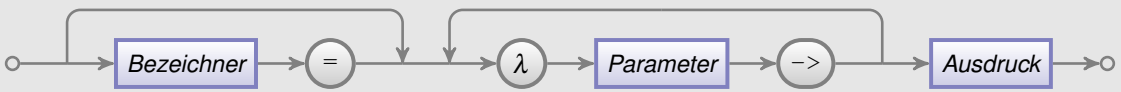
Lambda Currying

$$f = \lambda x_1 \rightarrow \lambda x_2 \rightarrow \ldots \lambda x_n \rightarrow e$$

Funkion Currying



Lambda Currying



Deklaration von Funktionen in λ Notation

Lambda Uncurrying

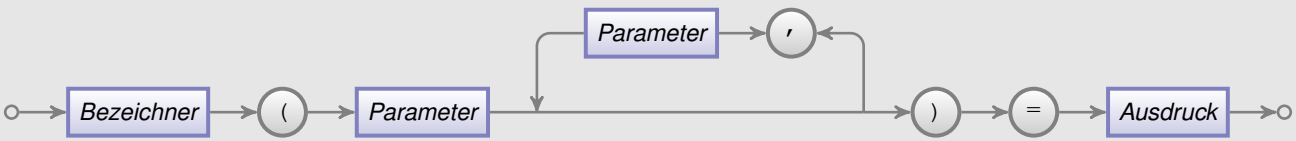
$$f = \lambda(x_1, x_2, \ldots, x_n) \rightarrow e$$

Deklaration von Funktionen in λ Notation

Lambda Uncurrying

$$f = \lambda(x_1, x_2, \ldots, x_n) \rightarrow e$$

Funkion Uncurrying



ABER

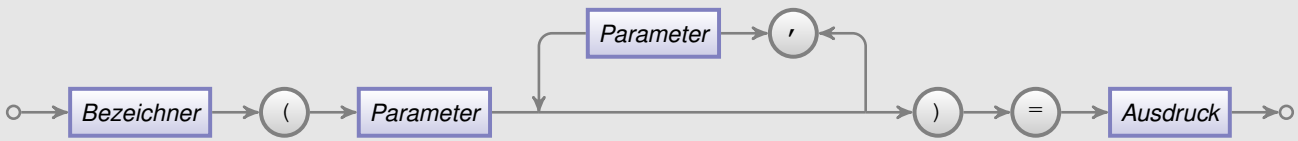
Das Tupel $(x_1, x_2, \ldots, x_n)$ ist ein eigener Datentyp

Deklaration von Funktionen in λ Notation

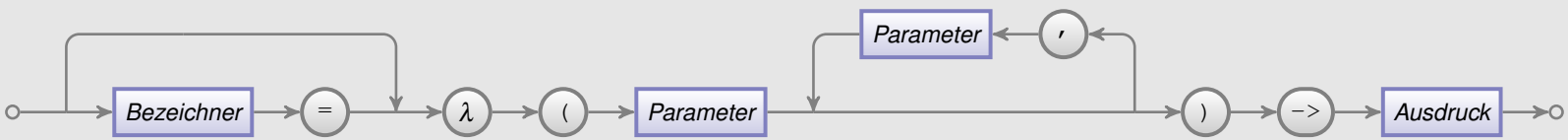
Lambda Uncurrying

$$f = \lambda(x_1, x_2, \ldots, x_n) \rightarrow e$$

Funkion Uncurrying

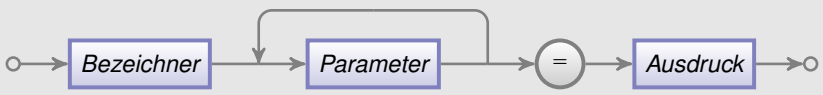


Lambda Uncurrying



Deklaration von Funktionen

```
plus :: Int -> Int -> Int
plus a b = a + b
```



Deklaration von Funktionen

```
plus :: Int -> Int -> Int
plus a b = a + b
```

Aufruf

```
plus 6 7
```

Deklaration von Funktionen

```
plus :: Int -> Int -> Int
plus a b = a + b
```

Aufruf

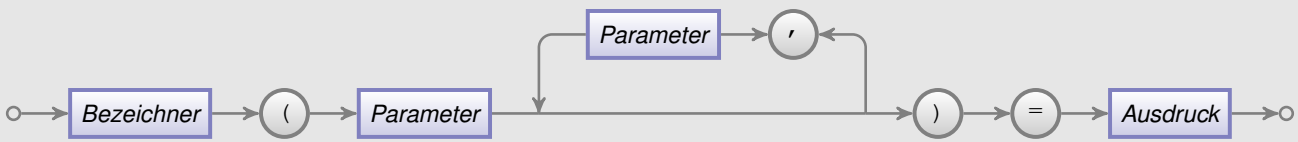
```
plus 6 7
```

Ausgabe

```
13
```

Deklaration von Funktionen

```
plus' :: (Int, Int) -> Int
plus' (a, b) = a + b
```



Deklaration von Funktionen

```
plus' :: (Int, Int) -> Int
plus' (a, b) = a + b
```

Aufruf

```
plus' (6, 7)
```

Deklaration von Funktionen

```
plus' :: (Int, Int) -> Int
plus' (a, b) = a + b
```

Aufruf

```
plus' (6, 7)
```

Ausgabe

```
13
```

Gültigkeitsbereiche

Gültigkeitsbereiche - Block

Block

- Definitionen im Block sind immer nur eine Stufe höher sichtbar (hier sind nicht let-in und where gemeint)
- Im Block ist alles Äußere sichtbar

Block - Einrückungen

In Haskell spielt das Layout des Quellcodes eine Rolle!

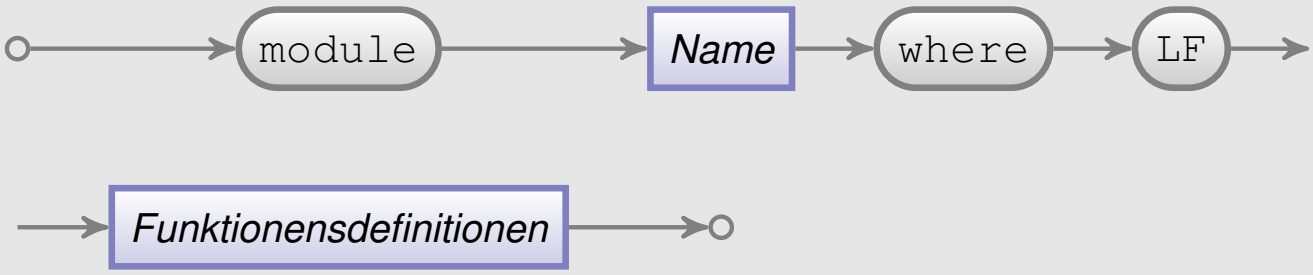
- Blöcke werden durch gleiche Einrückungstiefe kenntlich gemacht
- Einzelne Deklarationen werden durch Zeilenumbrüche getrennt
- Beginnt eine neue Zeile gegenüber dem aktuellen Block
  - Rechts eingerückt: aktuelle Zeile wird fortgesetzt
  - Links eingerückt: aktueller Block wird beendet
  - Direkt an seinem „linken Rand darunter“, so wird der Block fortgesetzt bzw. eine neue Deklaration eingeleitet

Gültigkeitsbereiche - Module

Module

- Das Programm kann in Module aufgeteilt werden
- Der Standard Modulname ist Main
- Module müssen mit einem Großbuchstaben beginnen
- Vorteile:
  - Vereinfachung des Programmdesigns, Strukturierung
  - Einfachere Isolation von Fehlern
  - Einfaches Ändern von Teilkomponenten ohne Einfluss auf andere Teile
  - Wiederverwendung von Code

Module



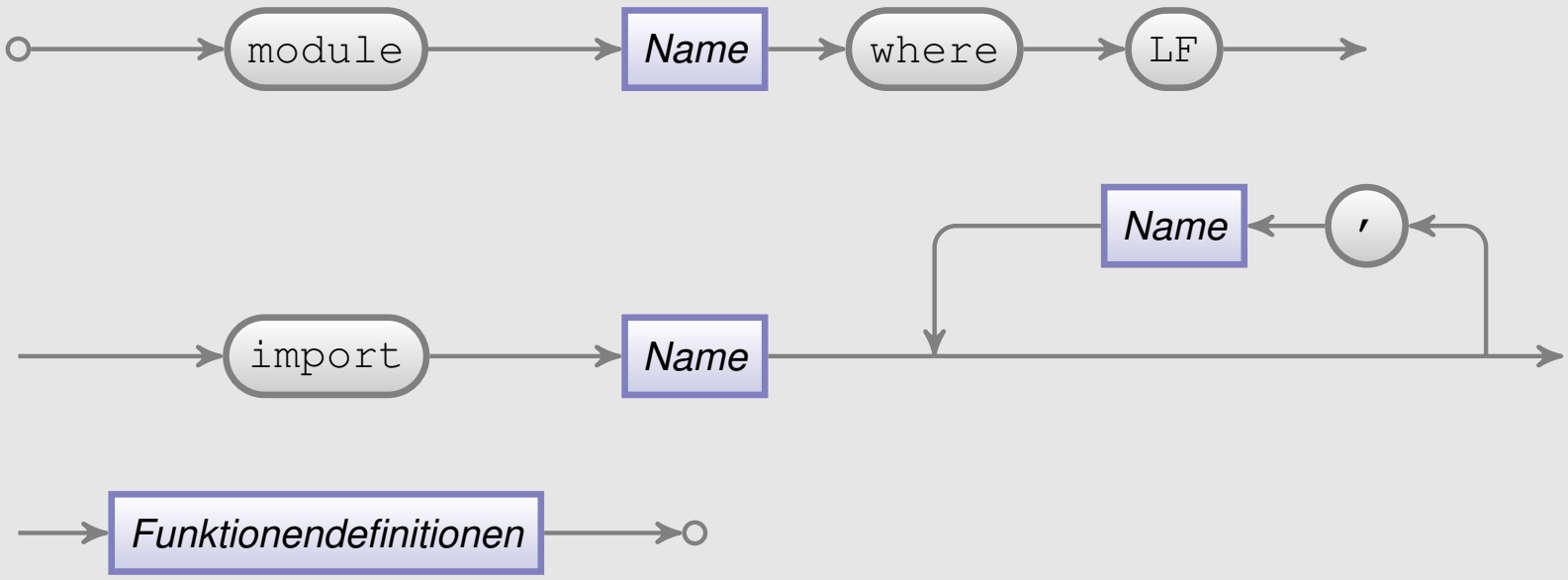
Module

```
module Wurf where
weite :: Double -> Double -> Double
weite v0 phi = ((square v0) / 9.81) * sin (2 * phi)
square :: Double -> Double
square x = x * x
```

```
module Foo where
import Wurf
foo ... = ... (weite v w) ...
bar ... = ... (square a) ...
```

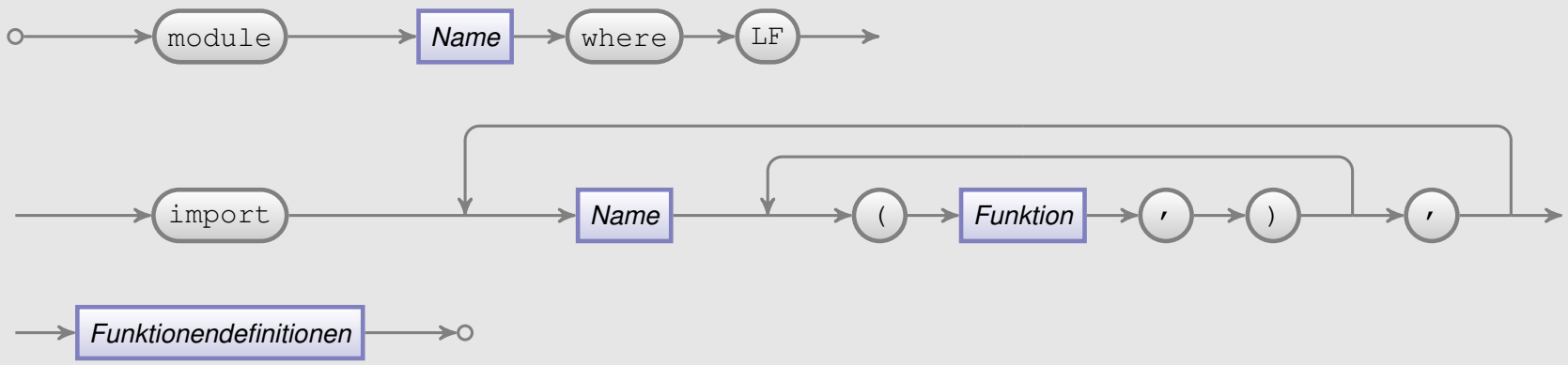
Module - Interfaces

Import



Module - Interfaces

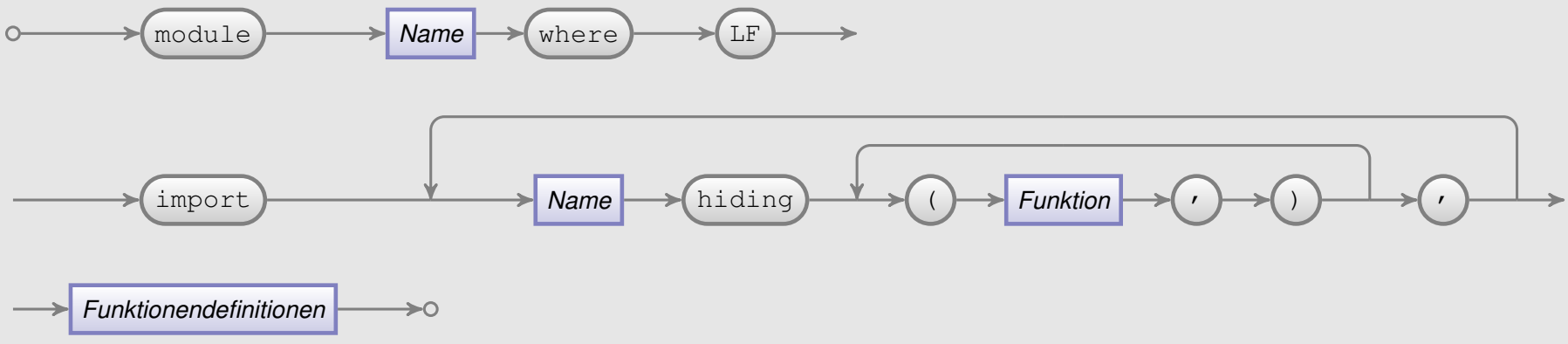
Selektiver Import



am Ende steht natürlich kein Komma

Module - Interfaces

Negativ selektiver Import



am Ende steht natürlich kein Komma

Module - Interfaces

```
module Wurf where
weite :: Double -> Double -> Double
weite v0 phi = ((square v0) / 9.81) * sin (2 * phi)
square :: Double -> Double
square x = x * x
```

```
module Foo where
import Wurf(weite)
foo ... = ... (weite v w) ...
bar ... = ... (square a) ...
```

Achtung

square ist für bar nicht definiert!

Module - Interfaces

```
module Wurf where
weite :: Double -> Double -> Double
weite v0 phi = ((square v0) / 9.81) * sin (2 * phi)
square :: Double -> Double
square x = x * x
```

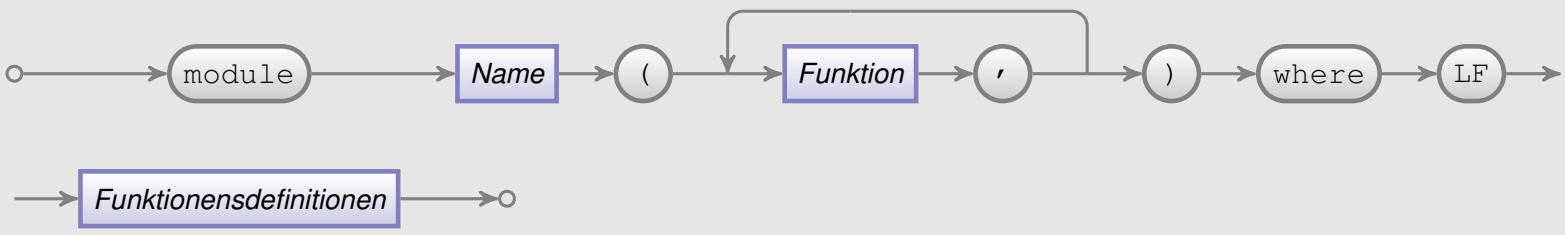
```
module Foo where
import Wurf hiding (weite)
foo ... = ... (weite v w) ...
bar ... = ... (square a) ...
```

Achtung

weite ist für foo nicht definiert!

Module - Sichtbarkeit

Module können festlegen was importiert werden darf



Am Ende steht natürlich kein Komma

Module - Sichtbarkeit

```
module Wurf(weite) where
weite :: Double -> Double -> Double
weite v0 phi = ((square v0) / 9.81) * sin (2 * phi)
square :: Double -> Double
square x = x * x
```

```
module Foo where
import Wurf
foo ... = ... (weite v w) ...
bar ... = ... (square a) ...
```

Achtung

In Wurf ist nur weite sichtbar

Bezeichner

Überladung von Namen

- Funktionen können in Haskell nicht im selben Modul überladen werden
- Funktionen können nur flach in Blöcken überdeckt werden
- Überladene Funktionen müssen mit dem Modul Bezeichner angesprochen werden.
- Für Polymorphie werden Typklassen verwendet

Überladung von Namen

```
maximum :: Int -> Int -> Int
maximum a b | a < b  = b
            | otherwise = a

maximum :: Bool -> Bool -> Bool
maximum a b = a || b
```

Fehler

Mehrfach-Definitionen sind unzulässig

Überladung von Namen

```
maximum :: Int -> Int -> Int
maximum a b | a < b  = b
            | otherwise = a

max :: Bool -> Bool -> Bool
max a b =
  let maximum a b = a || b
  in maximum a b
```

Achtung

Prelude.max für das durch Prelude definierte oder Modulname.max für unser max

Auflösen von Namen

- Ohne Modul Angabe werden Funktionen nur im „Import“ gesucht
- Prelude wird immer Importiert

Ausdrücke

Ausdrücke - Ausdrücke allgemein

Ausdrücke - Wiederholung

Elementarer Ausdruck

```
plus = 10 + 30
```

Ausdruck

```
plus' a b = a + b
```

Lambda (λ)-Ausdruck

```
plus'' = \a -> \b -> a + b
```

Ausdrücke allgemein

- In Haskell besteht ein Ausdruck aus nur wenigen Grundelementen
  - Konstante Werte
  - Variablen Werte (Variablen)
  - Funktionen
  - Verzweigungen wie:
    Guards, If-Then-Else, Case-Of ...
- Jeder Operator ist eine Funktion, die umdefiniert werden kann

Primitive Ausdrücke

sind benannte bzw. unbenannte (anonyme) Funktionen mit konstanten Ergebnissen

```
pi :: Double
pi = 3.14
```

Ausdrücke

sind „Berechnungen“ mit Variablen
besitzen „fest“ definierte Parameter

```
plus :: Int -> Int -> Int
plus a b = a + b
```

Ausdrücke - Lambda - Ausdrck

Lambda - Ausdruck

- Sind fast dasselbe wie normale Ausdrücke
- Parameter bzw. Variablen werden in λ-Notation angegeben

```
let f = \x y -> x + y
```

Aufruf

```
f 31 11
```

Lambda - Ausdruck

- Sind fast dasselbe wie normale Ausdrücke
- Parameter bzw. Variablen werden in λ-Notation angegeben

```
let f = \x y -> x + y
```

Aufruf

```
f 31 11
```

Ausgabe

```
42
```

Ausdrücke - Strukturierte Anweisungen

Strukturierte Anweisungen

- If-Then-Else
- Case-Of
- Pattern-Matching
- Guards als erweitertes Pattern-Matching

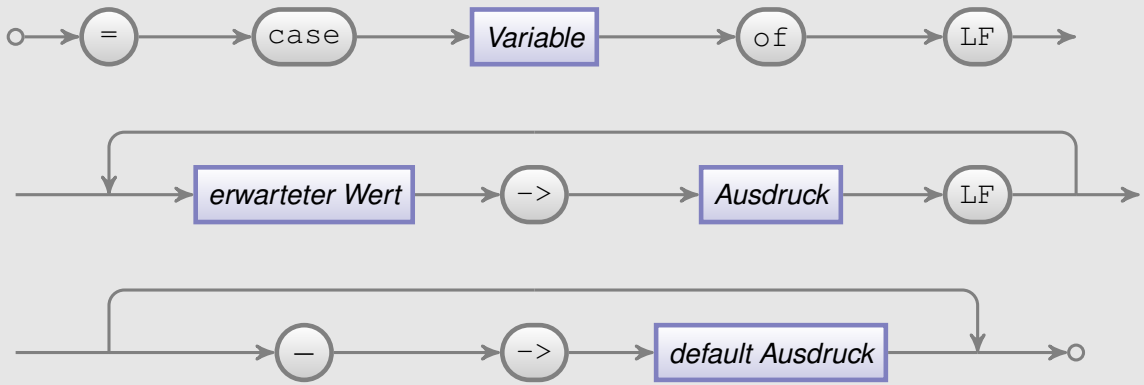
If-Then-Else

- Setzt das gewohnte If-Then-Else im Funktionsrumpf um
- Kann verschachtelt werden


```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = if mod x 2 == 1 && x > 20
             then x + sum xs
             else sum xs
```

Case-Of

- Setzt das gewohnte Case-Of innerhalb von Funktionsrümpfen um
- Erster „Treffer“ gewinnt



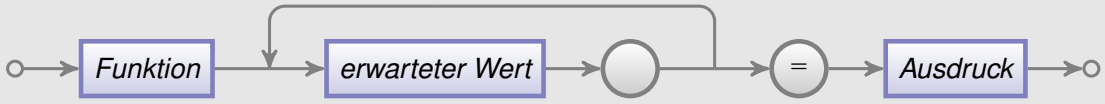
Case-Of

- Setzt das gewohnte Case-Of innerhalb von Funktionsrümpfen um
- Erster „Treffer“ gewinnt

```
not :: Bool -> Bool
not a = case a of
        True  -> False
        False -> True
```

Pattern-Matching

- Testen auf erwartete Werte
- Erster „Treffer“ gewinnt



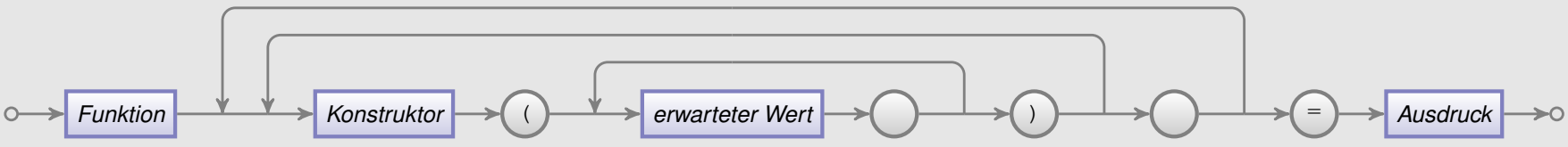
Pattern-Matching

- Testen auf erwartete Werte
- Erster „Treffer“ gewinnt

```
xor :: Bool -> Bool -> Bool
xor True  True  = False
xor False False = False
xor _     _     = True
```

Pattern-Matching

- Testen auf erwartete Struktur
- Aufspalten des Datentypes
- Erster „Treffer“ gewinnt



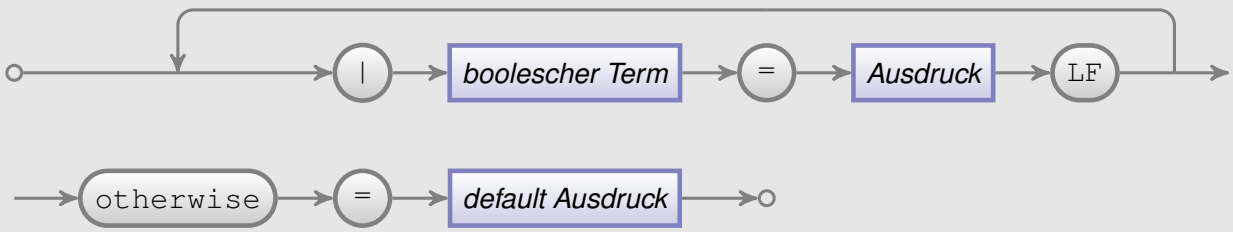
Pattern-Matching

- Testen auf erwartete Struktur
- Aufspalten des Datentypes
- Erster „Treffer“ gewinnt

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

Guards

- Erweitern das Pattern-Matching um boolesche Auswertungen in der Funktionsdefinition
- Erster „Treffer“ gewinnt



Guards

- Erweitern das Pattern-Matching um boolesche Auswertungen in der Funktionsdefinition
- Erster „Treffer“ gewinnt

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) | mod x 2 == 1 && x > 20 = x + sum xs
           | otherwise              = sum xs
```

Sturkturierte Anweisungen Beispiel ggt a b

If-Then-Else

```
ggT a b = if b == 0 then a else ggT b (mod a b)
```

Sturkturierte Anweisungen Beispiel ggt a b

If-Then-Else

```
ggT a b = if b == 0 then a else ggT b (mod a b)
```

Case-Of

```
ggT' a b = case b of
             0 -> a
             _ -> ggT' b (mod a b)
```

Sturkturierte Anweisungen Beispiel ggt a b

If-Then-Else

```
ggT a b = if b == 0 then a else ggT b (mod a b)
```

Pattern-Matching

```
ggT'' a 0 = a
ggT'' a b = ggT'' b (mod a b)
```

Sturkturierte Anweisungen Beispiel ggt a b

If-Then-Else

```
ggT a b = if b == 0 then a else ggT b (mod a b)
```

Guards

```
ggT''' a b | b == 0     = a
           | otherwise = ggT''' b (mod a b)
```

Ausdrücke - Iterationsanweisungen

Iterationsanweisungen

- In Haskell existieren keine Schleifen wie
- while, while do, for, repeat until ...
- Jede „Schleife“ muss über rekursive Funktionen realisiert werden
- Hierzu werden Funktionen höherer Ordnung benötigt

```
filter :: (Int -> Bool) -> [Int] -> [Int]
filter do []     = []
filter do (x:xs) | do x       = x : filter do xs
                 | otherwise  = filter do xs
```

Wendet do auf jedes Element der Liste an.

Ausdrücke - Sprunganweisungen

Sprunganweisungen

Es existieren keine Sprunganweisungen.

Ausblick für Morgen

Typkonstruktoren

Typüberprüfung und -berechnung

Typkonversion (Cast Anweisungen)

Polymorphismus

Technische Universität Braunschweig

Programmieren für Fortgeschrittene - eine Einführung in Haskell

Tag drei - noch etwas mehr

Stephan Mielke

26.03.2014

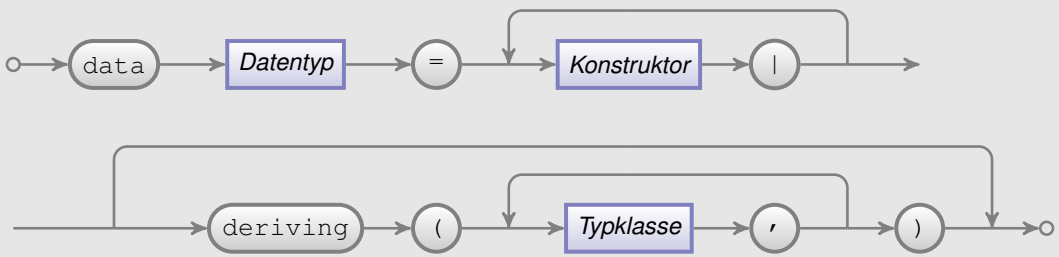
Typkonstruktoren

Typkonstruktoren - Eigene Datentypen

Typkonstruktoren - Aufzählungstyp

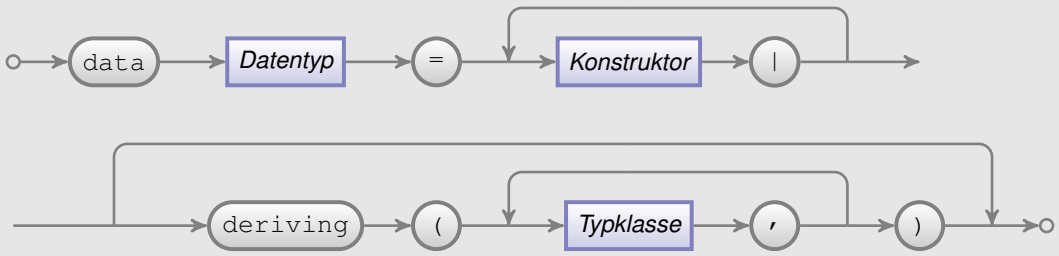
Aufzählungstypen sind mit Enums aus C bzw. C++ zu vergleichen und fassen inhaltlich Elemente zusammen

data

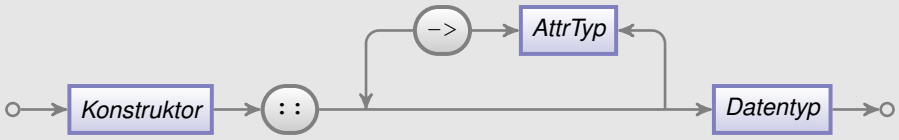


Typkonstruktoren - Aufzählungstyp

data



Konstruktor



Typkonstruktoren - Aufzählungstyp

```
data Color = Blue | Cyan | Yellow | Orange | Green
```

Typkonstruktoren - Aufzählungstyp

```
data Color = Blue | Cyan | Yellow | Orange | Green
```

Wie werden die Konstuktoren aussehen ?

Typkonstruktoren - Aufzählungstyp

```
data Color = Blue | Cyan | Yellow | Orange | Green
```

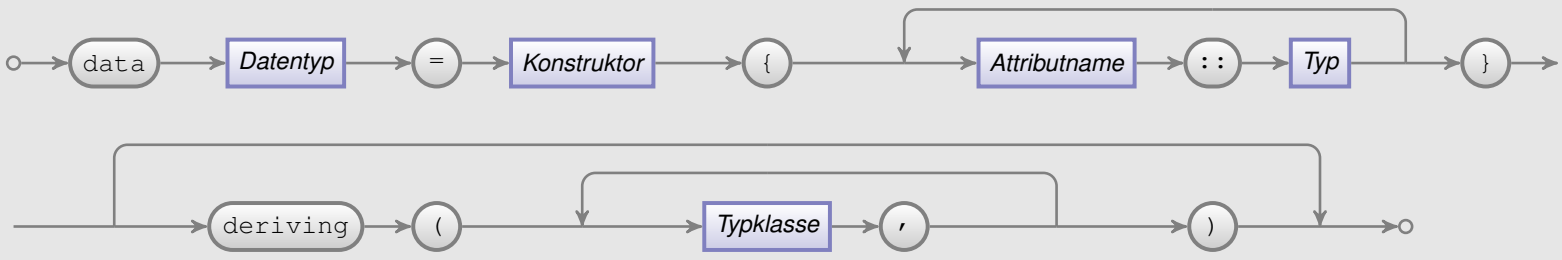
Konstruktoren:

```
Blue   :: Color
Cyan   :: Color
Yellow :: Color
Orange :: Color
Green  :: Color
```

Typkonstruktoren - Produkttyp

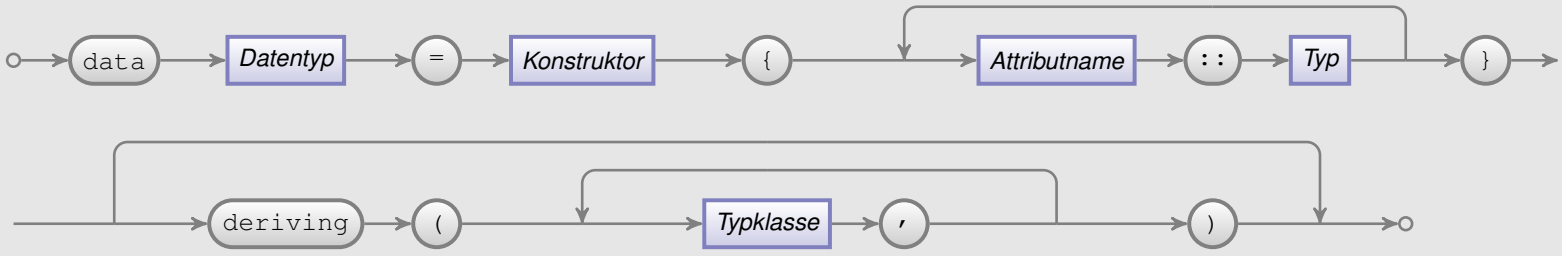
- Produkttyp ist ein Tupel der einzelnen Attribute
- Tupel fassen Gruppen von Daten zusammen, die logisch zusammen gehören und gemeinsam etwas Neues und Eigenständiges bilden

data mit allen Angaben

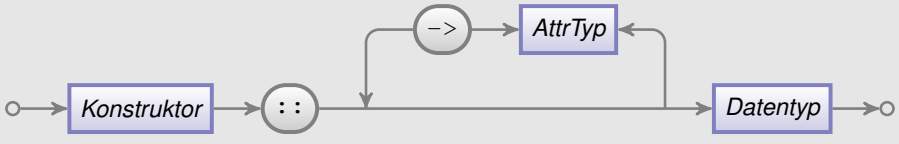


Typkonstruktoren - Produkttyp

data mit allen Angaben

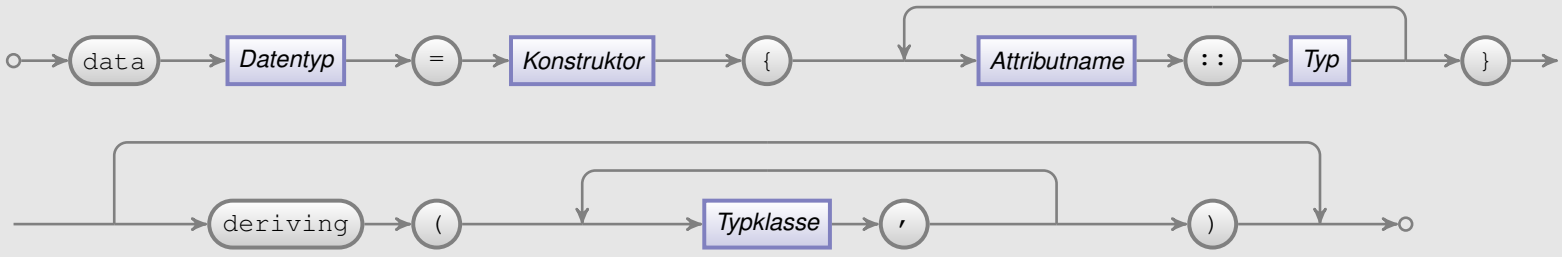


Konstruktor



Typkonstruktoren - Produkttyp

data mit allen Angaben



Selektor (bei allen Angaben automatisch erstellt)

Typkonstruktoren - Produkttyp

```
data Point = Point{x :: Double, y :: Double}
```

```
data Circle = Circle{center :: Point, radius :: Double}
```

Typkonstruktoren - Produkttyp

```
data Point = Point{x :: Double, y :: Double}
```

Kurz

```
data Point = Point Double Double
```

```
data Circle = Circle{center :: Point, radius :: Double}
```

Kurz

```
data Circle = Circle Point Double
```

Kurz Schreibweise

Bei der kurz Schreibweise werden keine Selektoren erstellt

Typkonstruktoren - Produkttyp

```
data Point = Point{x :: Double, y :: Double}
data Circle = Circle{center :: Point, radius :: Double}
```

Konstruktorfunktionen: ?

Typkonstruktoren - Produkttyp

```
data Point = Point{x :: Double, y :: Double}
data Circle = Circle{center :: Point, radius :: Double}
```

Konstruktorfunktionen:

```
Point :: Double -> Double -> Point
Circle :: Point -> Double -> Circle
```

Typkonstruktoren - Produkttyp

```
data Point = Point{x :: Double, y :: Double}
data Circle = Circle{center :: Point, radius :: Double}
```

Selektorfunktionen: ?

Typkonstruktoren - Produkttyp

```
data Point = Point{x :: Double, y :: Double}
data Circle = Circle{center :: Point, radius :: Double}
```

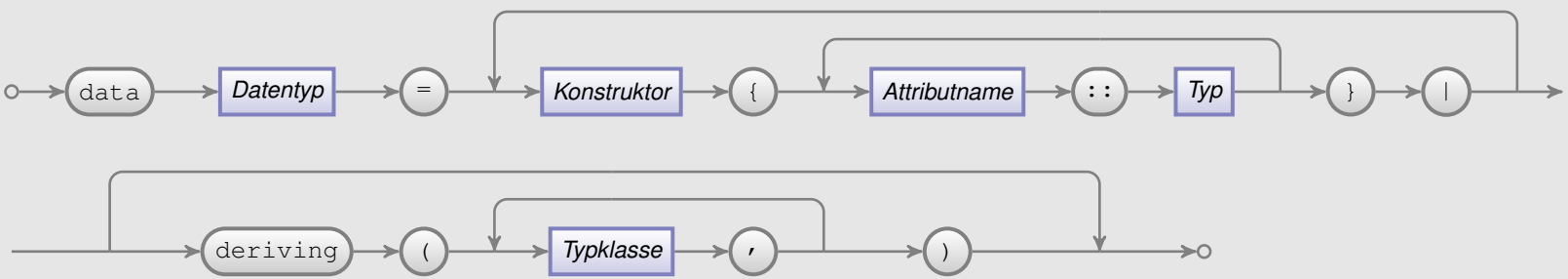
Selektorfunktionen:

```
x :: Point -> Double
y :: Point -> Double
center :: Circle -> Point
radius :: Circle -> Double
```

Typkonstruktoren - Summentyp

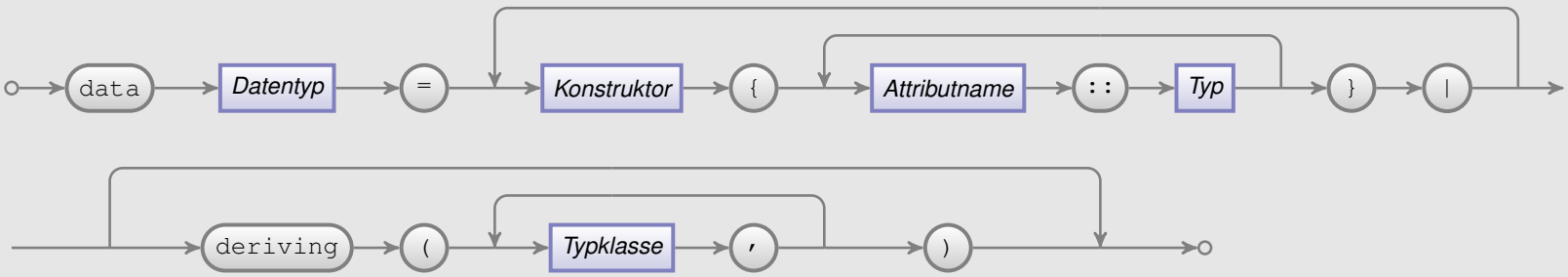
- Summentypen fassen inhaltlich verwandte (aber struktuell verschiedene) Elemente zusammen
- Sind eine Fusion von Aufzählungs- und Produkttyp

data mit allen Angaben

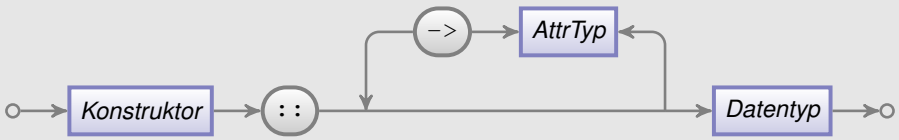


Typkonstruktoren - Summentyp

data mit allen Angaben

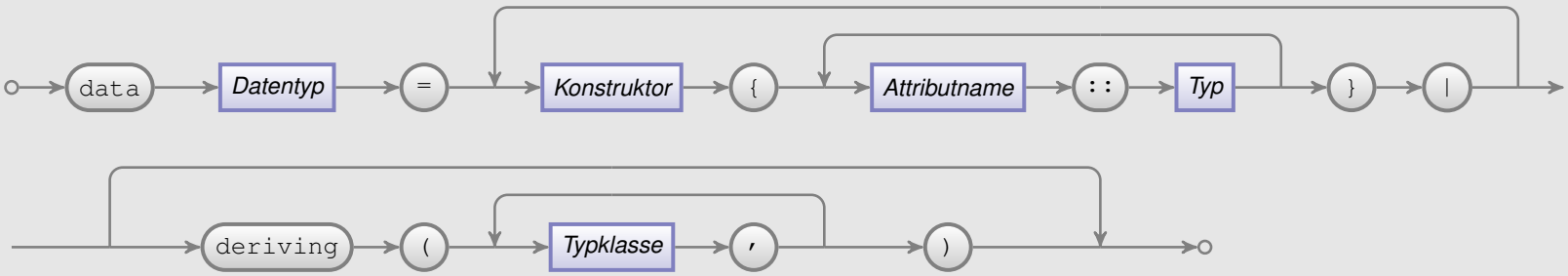


Konstruktor



Typkonstruktoren - Summentyp

data mit allen Angaben



Selektor (bei allen Angaben automatisch erstellt)

Typkonstruktoren - Summentyp

```
data Point = Point{x :: Double, y :: Double}

data Shape = Circle{center :: Point,
    radius :: Double}
  | Rectangle{point :: Point, width :: Double,
    height :: Double}
  | Triangle{point1 :: Point, point2 :: Point,
    point3 :: Point}
```

Frage

Was sind hier Selektoren und Konstruktoren?

Typkonstruktoren - Summentyp

```
data Shape = Circle{center :: Point, radius :: Double}
  | Rectangle{point :: Point, width :: Double,
    height :: Double}
  | Triangle{point1 :: Point, point2 :: Point,
    point3 :: Point}
```

Konstruktoren: ?

Typkonstruktoren - Summentyp

```
data Shape = Circle{center :: Point, radius :: Double}
  | Rectangle{point :: Point, width :: Double,
    height :: Double}
  | Triangle{point1 :: Point, point2 :: Point,
    point3 :: Point}
```

Konstruktoren:

```
Point :: Double -> Double -> Point
Circle :: Point -> Double -> Shape
Rectangle :: Point -> Double -> Double -> Shape
Triangle :: Point -> Point -> Point -> Shape
```

Typkonstruktoren - Summentyp

```
data Shape = Circle{center :: Point, radius :: Double}
  | Rectangle{point :: Point, width :: Double,
    height :: Double}
  | Triangle{point1 :: Point, point2 :: Point,
    point3 :: Point}
```

Selektoren: ?

Typkonstruktoren - Summentyp

```
data Shape = Circle{center :: Point, radius :: Double}
  | Rectangle{point :: Point, width :: Double,
    height :: Double}
  | Triangle{point1 :: Point, point2 :: Point,
    point3 :: Point}
```

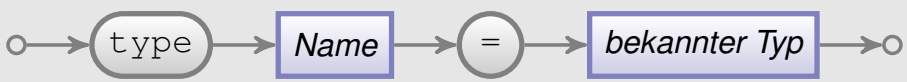
Selektoren:

```
x :: Point -> Double
y :: Point -> Double
center :: Shape -> Point
radius :: Shape -> Double
point :: Shape -> Point
width :: Shape -> Double
...
```

Typkonstruktoren - Typ-Synonyme

Typ-Synonyme

- Typ-Synonyme sind keine eigenen Typen sondern führen nur neue Namen für bekannte Typen ein
- Vorteile:
  - Verbesserte Lesbarkeit
  - Intern wird bei `type Euro = Int` wieder `Int`



Typ-Synonyme

```
type Euro = Int
type Cent = Int
type Preis = (Euro, Cent)

type Tupel = (Int, Int)
```

Achtung

Preis und Tupel sind für uns und intern (Int, Int)

Typ-Synonyme

```
type Euro = Int
type Cent = Int

add :: Euro -> Euro -> Euro
add a b = a + b

add' :: Euro -> Cent -> Int
add' a b = a + b
```

Achtung

add' 5 (add 5 8) funktioniert

Typ-Synonyme mit Typsicherheit

- newtype wird statt type verwendet, wenn Typsicherheit benötigt wird
- newtype verhält sich somit geauso wie data
- Jedes newtype kann durch data ersetzt werden
- Jedoch data kann nur in Ausnahmefällen durch newtype ersetzt werden

Typ-Synonyme mit Typsicherheit

- newtype wird statt type verwendet, wenn Typsicherheit benötigt wird
- newtype verhält sich somit geauso wie data
- Jedes newtype kann durch data ersetzt werden
- Jedoch data kann nur in Ausnahmefällen durch newtype ersetzt werden

```
newtype Euro = Euro Int
newtype Cent = Cent Int
```

Achtung

Euro und Cent sind nicht kompatibel

Typkonstruktoren - Rekursive Datenstrukturen

Rekursive Datenstrukturen

- Datenstrukturen können auf sich selbst verweisen
- Es sind „unendliche“ Rekursionen erlaubt, solange der Arbeitsspeicher mitspielt

Rekursive Datenstrukturen

```
data List = []
          | Cons {head :: Int, tail :: List}

data Tree = Nil
          | Tree {left :: Tree, value :: Int,
            right :: Tree}

data Nat = Zero
         | Succ {pred :: Nat}
```

Typkonstruktoren - Listen

Listen

```
data [a] = []
         | Cons {head :: a, tail :: [a]}
```

- Listen sind Folgen von Elementen gleichen Types
- a ist hier der Platzhalter für einen Typ
  somit kann das a für Int, Integer usw stehen

Konstruktoren: ?

Listen

```
data [a] = []
         | Cons {head :: a, tail :: [a]}
```

- Listen sind Folgen von Elementen gleichen Types
- a ist hier der Platzhalter für einen Typ
  somit kann das a für Int, Integer usw stehen

Konstruktoren:

```
[]   :: [a]
Cons :: a -> [a] -> [a]
```

Listen

```
data [a] = []
         | Cons {head :: a, tail :: [a]}
```

Selektoren: ?

Listen

```
data [a] = []
         | Cons {head :: a, tail :: [a]}
```

Selektoren:

```
head :: [a] -> a
tail :: [a] -> [a]
```

Listen in Funktionen

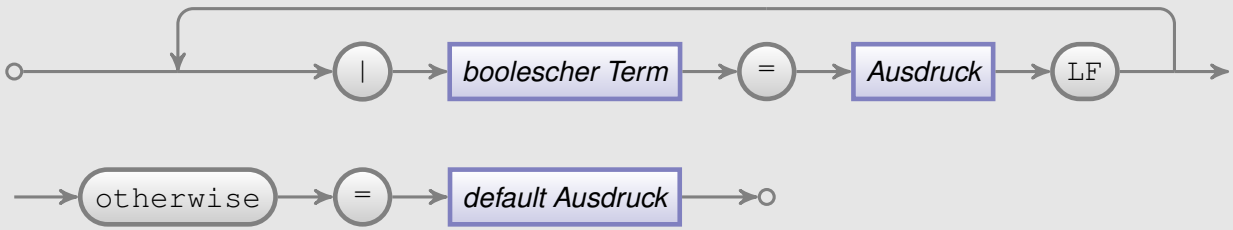
```
length :: [Int] -> Int
length []     = 0
length (_:xs) = 1 + length xs

append :: [Int] -> [Int] -> [Int]
append []     ys = ys
append (x:xs) ys = x : append xs ys

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

Listen in Funktionen

```
filter :: [Int] -> [Int]
filter []     = []
filter (x:xs) | ok x      = x : filter xs
              | otherwise = filter xs
              where
                ok x = (mod x 2) == 1
```

Was macht diese Funktion?

Was ist der Funktionskopf von ok?



Listen in Funktionen

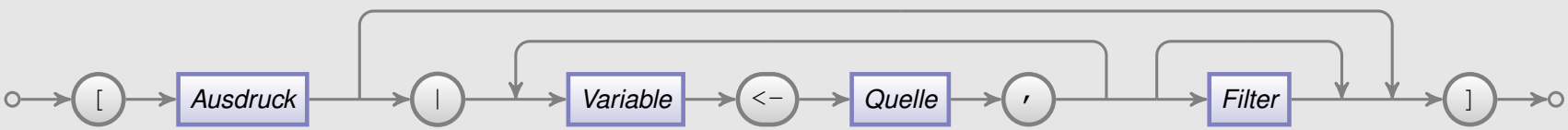
```
filter :: [Int] -> [Int]
filter []     = []
filter (x:xs) | ok x      = x : filter xs
              | otherwise = filter xs
              where
                ok x = (mod x 2) == 1
```

Was macht diese Funktion?
Was ist der Funktionskopf von ok?

```
ok :: Int -> Bool
ok     x =  (mod x 2) == 1
```

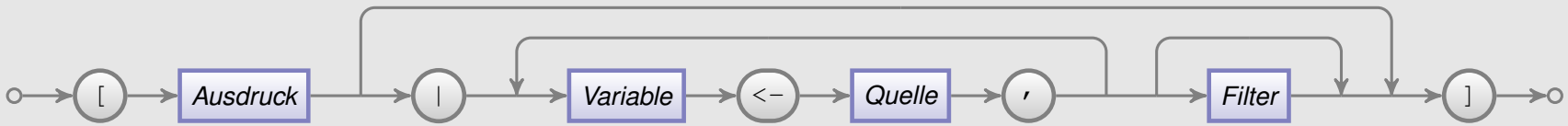
Listen Generatoren

- Für Listen existieren Generatoren
- Sonst müsste jedes Element von Hand aufgeschrieben werden
- [0..] generiert eine unendliche Liste
- Warum es unendliche Listen geben kann kommt später
- [0..10] generiert [0,1,2,3,4,5,6,7,8,9,10]



Listen Generatoren

```
[x * x| x <- [1..5]]
[(i,j)|i <- [1,2], j <- [1..4]]

[y | y <- [1..], even y]
[a * a | a <- [1..], odd a]
[square x | x <- [0..], square x < 10]
```



Listen Generatoren

Berechnung aller Primzahlen

```
primes = sieves [2..]
  where
    sieves (p:xs) = p:sieves [x|x<- xs, mod x p > 0]
```

Wichtige Liste

- Ihr erinnert euch noch an Char?
- Ihr erinnert euch noch an type?
- Wie wird wohl String definiert sein?

Wichtige Liste

- Ihr erinnert euch noch an Char?
- Ihr erinnert euch noch an type?
- Wie wird wohl String definiert sein?

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

String

- Strings sind Listen von Chars
- Alle Funktionen die generisch für Listen definiert sind, sind auch für Strings definiert
- Strings werden intern nicht anders als normale Listen behandelt
- Also kein Stringspool, keine Unabänderbarkeit von Strings usw. . . .

String

```
wochenTag :: Int -> String
wochenTag 1 = "Montag"
wochenTag 2 = "Dienstag"
wochenTag 3 = "Mittwoch"
wochenTag 4 = "Donnerstag"
wochenTag 5 = "Freitag"
wochenTag 6 = "Samstag"
wochenTag 7 = "Sonntag"
wochenTag _ = undefined
```

„Bottom Element“

jeder Wert außer 1, 2, 3, 4, 5, 6, 7 führt zum „Bottom Element“ und führt zu einer Prelude.undefined Exception

Typüberprüfung und -berechnung

Typüberprüfung

- Es sind nur Typengleich, welche auch gleich sind!
- Int ist kompatibel zu Int
- Aber Int ist nicht kompatibel zu Integer

Typüberprüfung

- Typ-Synonyme sind gleich, wenn sie auf den gleichen Typ abbilden
- `type Euro = Int`
- `type Laenge = Int`
- Zwischen beiden Typen gibt es keinen Unterschied

Typberechnung

- Wenn keine Typen angegeben wurden, wird der passende Typ berechnet
- Bei Eingaben im GHCi wird der richtige Typ „erraten“

Typberechnung

- Wenn keine Typen angegeben wurden, wird der passende Typ berechnet
- Bei Eingaben im GHCi wird der richtige Typ „erraten“
- Ok statt „raten“ wird der Typcheck Algorithmus von Robin Milner verwendet

Typkonversion (Cast Anweisungen)

Typkonversion

- In Haskell existieren keine Cast Anweisungen wie in Java
- Jeder Cast wird über eine Funktion realisiert
- `toInteger :: a -> Integer`
- `fromInteger :: Integer -> a`
- ...

Polymorphismus

Polymorphismus - Typparameter

Polymorphismus

- Bisher hatten wir nur Funktionen für genau einen Typ
- Nun lernen wir Typklassen und Typparameter kennen

Typparameter

Erinnert ihr euch noch an Listen?

```
data List = []
          | Cons {head :: Int, tail :: List}
```

Damit wir nicht für jeden Datentyp eine neue Liste definieren müssen

Cons ist nun (:)

```
data List a = []
            | (:) {head :: a, tail :: List a}
```

Typparameter

Erinnert ihr euch noch an Listen?

```
data List = []
          | Cons {head :: Int, tail :: List}
```

kurz:

```
data [a] = []
         | (:) {head :: a, tail :: [a]}
```

Typparameter

Genauso in Funktionen

```
append :: [Int] -> [Int] -> [Int]
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

Typparameter

Genauso in Funktionen

```
append :: [Int] -> [Int] -> [Int]
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

Damit wir nicht für jeden Datentyp eine neue Funktion definieren müssen

```
append :: [a] -> [a] -> [a]
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

Polymorphismus - Typklassen

Typklassen

- Typklassen fassen Typen zusammen, die ähnliche Operationen unterstützen
- Alle Ausprägungen einer Funktion einer Typklassen tragen dann den gleichen Namen.
- $\rightarrow$ Overloading, d.h. der gleiche Funktionsname steht für unterschiedliche Implementierungen

Typklassen

Zu allgemein

```
(+) :: a -> a -> a
```

Typklassen

Zu allgemein

```
(+) :: a -> a -> a
```

Zu speziell

```
(+) :: Int -> Int -> Int
```

Typklassen

Zu allgemein

```
(+) :: a -> a -> a
```

Zu speziell

```
(+) :: Int -> Int -> Int
```

Genau richtig

```
(+) :: Num a =>a -> a -> a
```

Typklassen

Nutzung von Typklassen

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

Typklassen

Eigene Datentypen mit Typklassen

```
data Point = Point{x :: Double, y :: Double}
  deriving (Eq, Show)

data Circle = Circle{center :: Point,
  radius :: Double}
  deriving (Eq, Show)
```

mit deriving wird geraten wie die Implementierung von Eq, Ord, Show usw. sein sollen

Typklassen

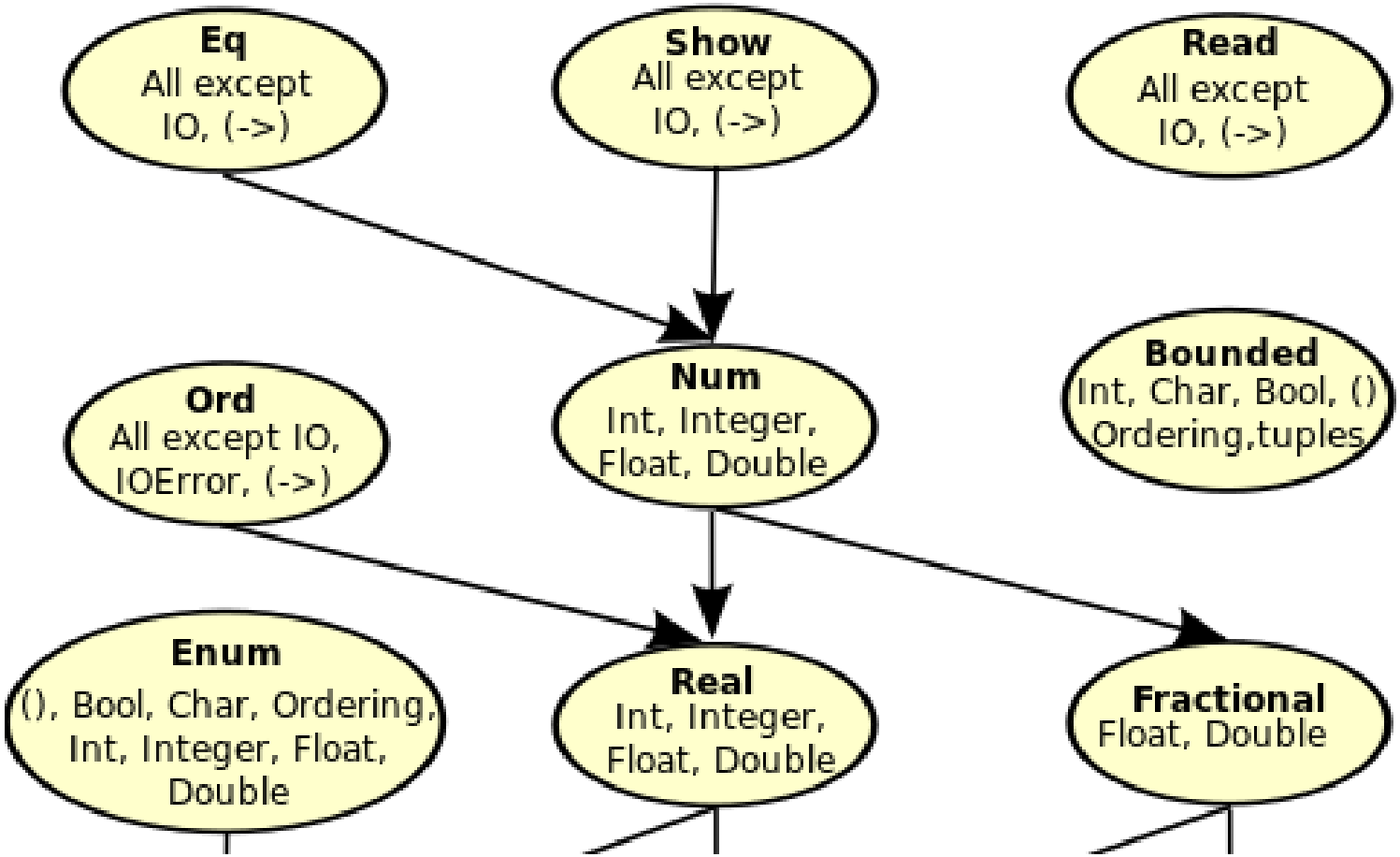


Abbildung: Typklassen Teil 1 ©wikibooks.org

Typklassen

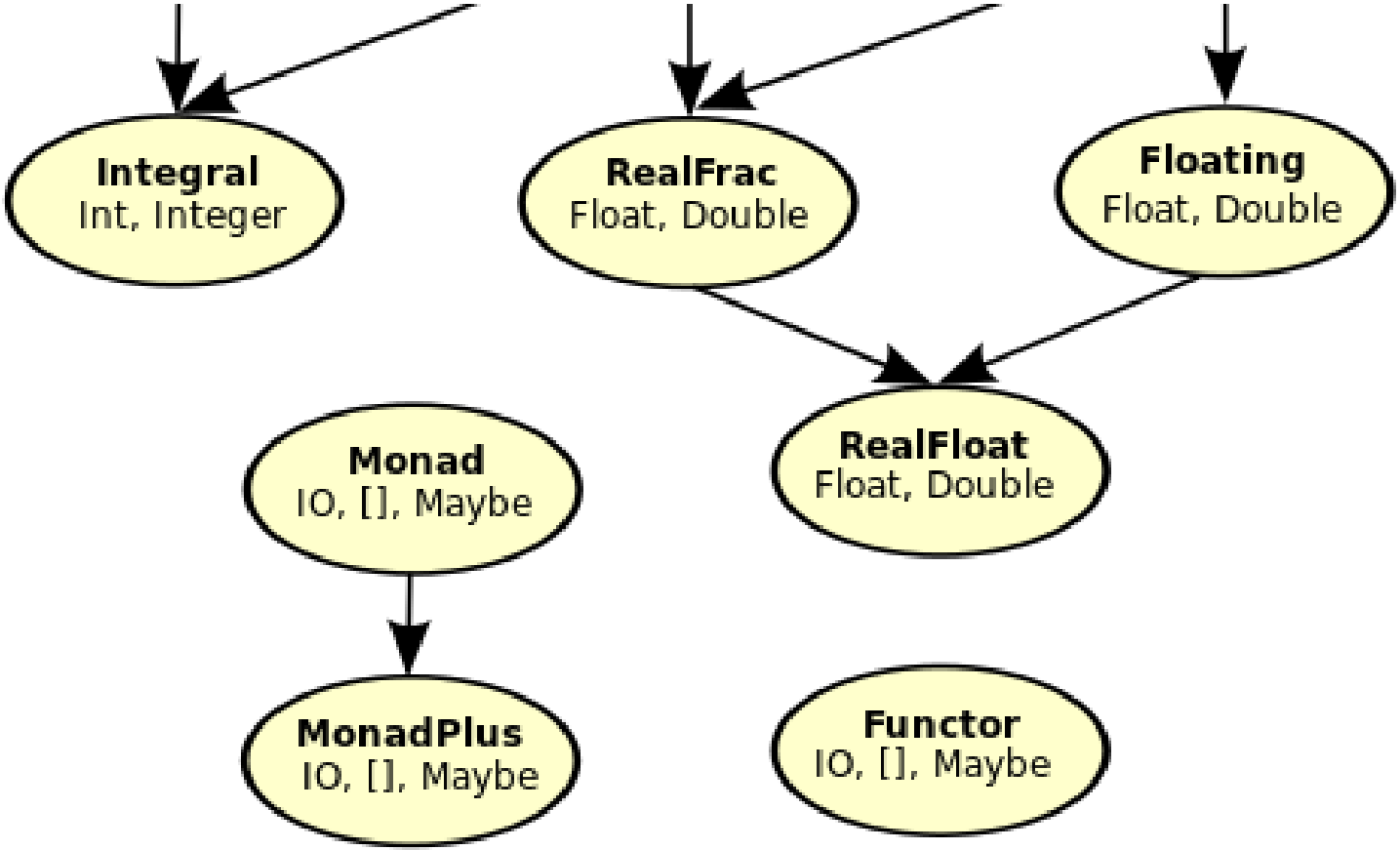


Abbildung: Typklassen Teil 2 ©wikibooks.org

Typklassen

Die Eq Typklasse

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

Achtung

Die Definition ist zirkulär!

Typklassen

Die Ord Typklasse

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a
```

Achtung

Die Definition ist zirkulär!

Typklassen

Instance von Eq und Ord für Buch

```
data Buch = Buch Int [String] String deriving (Show)

instance Eq Buch where
  (Buch isbn1 _ _) == (Buch isbn2 _ _)
    = isbn1 == isbn2

instance Ord Buch where
   (Buch isbn1 _ _) `compare` (Buch isbn2 _ _)
     = isbn1 `compare` isbn2
```

Typklassen

Instance von Eq

```
instance Eq Buch where
  (Buch isbn1 _ _) == (Buch isbn2 _ _)
    = isbn1 == isbn2
```

Aufruf

```
Buch 123 ["ich", "du"] "Hallo" == Buch 123 ["du"] "keiner"
```

Typklassen

Instance von Eq

```
instance Eq Buch where
  (Buch isbn1 _ _) == (Buch isbn2 _ _)
    = isbn1 == isbn2
```

Aufruf

```
Buch 123 ["ich", "du"] "Hallo" == Buch 123 ["du"] "keiner"
```

Ausgabe

```
True
```

Typklassen

Instance von Ord

```
instance Ord Buch where
    (Buch isbn1 _ _) `compare` (Buch isbn2 _ _)
      = isbn1 `compare` isbn2
```

Aufruf

```
Buch 123 ["ich", "du"] "Hallo" < Buch 123 ["du"] "keiner"
```

Typklassen

Instance von Ord

```
instance Ord Buch where
    (Buch isbn1 _ _) `compare` (Buch isbn2 _ _)
      = isbn1 `compare` isbn2
```

Aufruf

```
Buch 123 ["ich", "du"] "Hallo" < Buch 123 ["du"] "keiner"
```

Ausgabe

```
False
```

Ausblick für Morgen

Lazy

Funktionen höherer Ordnung (HOF)

Currying - für Fortgeschrittene

Lambda Ausdrücke

Programmieren für Fortgeschrittene - eine Einführung in Haskell

Tag vier - ein bisschen noch

Stephan Mielke

27.03.2014

Lazy

Lazy

Betrachte folgende Funktion

```
rechne :: Double -> Double -> Double
rechne a b = if a > 10
             then a + b
             else a
```

Was erwartet ihr beim Aufruf von

```
rechne 12 6
```

Lazy

Betrachte folgende Funktion

```
rechne :: Double -> Double -> Double
rechne a b = if a > 10
             then a + b
             else a
```

und bei

```
rechne 9 (10 / 0)
```

Lazy

Wir betrachten

```
prims :: [Integer]-> Int -> [Integer]
prims _ 0  = []
prims (p:xs) i  = (:) p $prims
                  [x|x<- xs, mod x p > 0] $i + 1
```

Terminiert die Funktion?

Lazy

Wir betrachten

```
prims :: [Integer]-> Int -> [Integer]
prims _ 0  = []
prims (p:xs) i  = (:) p $prims
                  [x|x<- xs, mod x p > 0] $i + 1
```

Terminiert die Funktion?

Terminiert die Funktion auch bei der Eingabe von

```
primes [2..] 1
```

Lazy

- Haskell verwendet die Lazy-Evaluation für Ausdrücke
- Lazy $\equiv$ Call-by-Need
- Dadurch sind Funktionen nicht strikt

Lazy - unendliche Listen

- Es werden vom Start an eine bestimmte Anzahl an Elemente erstellt
- Wenn weitere Elemente benötigt werden, werden diese neu Erstellt
- Wird immer nur ein Abschnitt benötigt wird der Start wieder gelöscht stellt es euch als „Ringpuffer“ vor
- Wenn jedoch alle Elemente benötigt werden $\rightarrow$ bis Speicher voll

Lazy - Parameter und Ausdrücke

- Haskell verwendet Call-by-need
- Call-by-need ist Form es Call-by-name

Lazy - Call-by-name

- Ausdrücke werden nicht sofort ausgewertet sondern nur übergeben
- max (4 + 6) (10 / 0)

Lazy - Call-by-name

- Ausdrücke werden nicht sofort ausgewertet sondern nur übergeben
- max (4 + 6) (10 / 0)
- $\Rightarrow$ if (4 + 6) > (10 / 0) then (4 + 6) else (10 / 0)

Lazy - Call-by-name

- Ausdrücke werden nicht sofort ausgewertet sondern nur übergeben
- max (4 + 6) (10 / 0)
- $\Rightarrow$ if (4 + 6) > (10 / 0) then (4 + 6) else (10 / 0)
- $\Rightarrow$ (4 + 6) > (10 / 0)

Lazy - Call-by-name

- Ausdrücke werden nicht sofort ausgewertet sondern nur übergeben
- max (4 + 6) (10 / 0)
- $\Rightarrow$ if (4 + 6) > (10 / 0) then (4 + 6) else (10 / 0)
- $\Rightarrow$ (4 + 6) > (10 / 0)
- $\Rightarrow$ 10 > (10 / 0)

Lazy - Call-by-name

- Ausdrücke werden nicht sofort ausgewertet sondern nur übergeben
- max (4 + 6) (10 / 0)
- $\Rightarrow$ if (4 + 6) > (10 / 0) then (4 + 6) else (10 / 0)
- $\Rightarrow$ (4 + 6) > (10 / 0)
- $\Rightarrow$ 10 > (10 / 0)
- $\Rightarrow$ $\lightning$

Lazy - Call-by-need

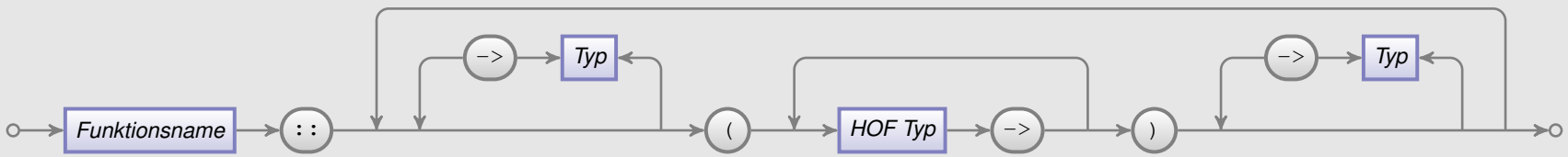
- Bei Call-by-need erweitert Call-by-name um Sharing
- Sharing: gleiche Ausdrücke werden nur einmal ausgewertet

Funktionen höherer Ordnung (HOF)

Funktionen höherer Ordnung (HOF) - Allgemeines zu HOF

Funktionen höherer Ordnung (HOF)

- Funktionen können als Parameter nicht nur Ausdrücke sondern auch Funktionen erhalten
- Dieses wird im Funktionskopf angegeben



Funktionen höherer Ordnung (HOF)

```
filter :: (Int -> Bool) -> [Int] -> [Int]
filter do []     = 0
filter do (x:xs) | do x       = x : filter do xs
                 | otherwise = filter do xs
```

Funktionen höherer Ordnung (HOF) - Funktionskomposition

Funktionskomposition

Wie vermeiden wir am besten „Klammerungswirrwarr“

```
f (f (f (f (f (f (f (f (f x ))))))))
```

Funktionskomposition

Wie vermeiden wir am besten „Klammerungswirrwarr“

```
f (f (f (f (f (f (f (f (f x ))))))))
```

Mit dem „Punkt“-Operator können wir Funktionen verbinden

```
(f.f.f.f.f.f.f.f.f) x
```

Funktionskomposition

Wie vermeiden wir am besten „Klammerungswirrwarr“

```
f (f (f (f (f (f (f (f (f x ))))))))
```

Mit dem „Punkt“-Operator können wir Funktionen verbinden

```
(f . f . f . f . f . f . f . f . f) x
```

Oder dem $-Operator die Auswertungsreihenfolge verändern

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

Funktionskomposition

Der „Punkt“-Operator ist definiert mit

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) outerFunc innerFunc x = outerFunc (innerFunc x)
```

Das Resultat der inneren Funktion wird auf die äußere angewendet.

Funktionskomposition

Der $-Operator ist definiert mit

```
($) :: (a -> b) -> a -> b
($) func x = func x
```

Die Funktion wird auf das Resultat von dem Ausdruck der „rechts“ vom Operator steht angewandt.

Currying - für Fortgeschrittene

Currying

- Currying bzw. Schönfinkeln ist das Zusammenfassen von Argumenten
- Wird in Sprachen und Kalkülen verwendet, in dem nur ein Argument erlaubt ist.
  z.B. in der λ-Notation
- Die Form und Art des Zusammenfassens ist unterschiedlich

Currying in der Lambda-Notation

- $\lambda\ x\ y\ z\ .\ x\ y\ z$
- wird aufgespalten zu

Currying in der Lambda-Notation

- $\lambda\ x\ y\ z\ .\ x\ y\ z$
- wird aufgespalten zu
- $\lambda\ x\ .\ \lambda\ y\ .\ \lambda\ z\ .\ x\ y\ z$

Currying in der Lambda-Notation

- $\lambda\ x\ y\ z\ .\ x\ y\ z$
- wird aufgespalten zu
- $\lambda\ x\ .\ \lambda\ y\ .\ \lambda\ z\ .\ x\ y\ z$
- wird ausgewertet mit den Argumenten $a\ b\ c$

Currying in der Lambda-Notation

- $\lambda\ x\ y\ z\ .\ x\ y\ z$
- wird aufgespalten zu
- $\lambda\ x\ .\ \lambda\ y\ .\ \lambda\ z\ .\ x\ y\ z$
- wird ausgewertet mit den Argumenten $a\ b\ c$
- $(\lambda\ x\ .\ \lambda\ y\ .\ \lambda\ z\ .\ x\ y\ z)a\ b\ c$

Currying in der Lambda-Notation

- $\lambda\ x\ y\ z\ .\ x\ y\ z$
- wird aufgespalten zu
- $\lambda\ x\ .\ \lambda\ y\ .\ \lambda\ z\ .\ x\ y\ z$
- wird ausgewertet mit den Argumenten $a\ b\ c$
- $(\lambda\ x\ .\ \lambda\ y\ .\ \lambda\ z\ .\ x\ y\ z)a\ b\ c$
- $(\lambda\ y\ .\ \lambda\ z\ .\ a\ y\ z)b\ c$

Currying in der Lambda-Notation

- $\lambda\ x\ y\ z\ .\ x\ y\ z$
- wird aufgespalten zu
- $\lambda\ x\ .\ \lambda\ y\ .\ \lambda\ z\ .\ x\ y\ z$
- wird ausgewertet mit den Argumenten $a\ b\ c$
- $(\lambda\ x\ .\ \lambda\ y\ .\ \lambda\ z\ .\ x\ y\ z)a\ b\ c$
- $(\lambda\ y\ .\ \lambda\ z\ .\ a\ y\ z)b\ c$
- $(\lambda\ z\ .\ a\ b\ z)c$

Currying in der Lambda-Notation

- $\lambda\ x\ y\ z\ .\ x\ y\ z$
- wird aufgespalten zu
- $\lambda\ x\ .\ \lambda\ y\ .\ \lambda\ z\ .\ x\ y\ z$
- wird ausgewertet mit den Argumenten $a\ b\ c$
- $(\lambda\ x\ .\ \lambda\ y\ .\ \lambda\ z\ .\ x\ y\ z)a\ b\ c$
- $(\lambda\ y\ .\ \lambda\ z\ .\ a\ y\ z)b\ c$
- $(\lambda\ z\ .\ a\ b\ z)c$
- $(a\ b\ c)$

Currying in Haskell

- Auch wenn wir in Haskell Funktionen mehrere Argumente übergeben können
- Intern hat jede Funktion nur ein oder kein Argument!

Currying in Haskell

Aufruf

```
:t xor True
```

Ausgabe

```
xor True :: Bool -> Bool
```

Aufruf

```
(xor True) False
```

Ausgabe

```
True
```

Currying in Haskell

Aufruf

```
:t xor True
```

Ausgabe

```
xor True :: Bool -> Bool
```

Aufruf

```
(xor True) False
```

Ausgabe

```
False
```

Currying in Haskell

- Currying erleichtert das Arbeiten mit Funktionen höherer Ordnung
- Sehen wir uns folgendes Beispiel an

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Wie würdet ihr map aufrufen um jedes Element einer Liste um 2 zu erhöhen?

Currying in Haskell

- Currying erleichtert das Arbeiten mit Funktionen höherer Ordnung
- Sehen wir uns folgendes Beispiel an

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Wie würdet ihr map aufrufen um jedes Element einer Liste um 2 zu erhöhen?

```
map (+ 2) [1..10]
```

Currying in Haskell

- Soll Currying unterbunden werden, so muss die Anzahl der Argumente von Anfang an ≤ 1 sein
- `f :: Int -> Int -> Int`
- Hier für kommen Tupel ins Spiel

Currying in Haskell

- Soll Currying unterbunden werden, so muss die Anzahl der Argumente von Anfang an ≤ 1 sein
- `f :: Int -> Int -> Int`
- Hier für kommen Tupel ins Spiel
- `f' :: (Int, Int) -> Int`
- Dieses „Abändern“ ist jedoch nur bei eigenen Funktionen möglich

Currying in Haskell

- Soll Currying unterbunden werden, so muss die Anzahl der Argumente von Anfang an ≤ 1 sein
- `f :: Int -> Int -> Int`
- Hier für kommen Tupel ins Spiel
- `f' :: (Int, Int) -> Int`
- Dieses „Abändern“ ist jedoch nur bei eigenen Funktionen möglich
- Funktionen können dies jedoch für uns übernehmen

Currying in Haskell

curry

```
curry :: ((a, b) -> c) a -> b -> c
curry f x y = f (x, y)
```

Currying in Haskell

curry

```
curry :: ((a, b) -> c) a -> b -> c
curry f x y = f (x, y)
```

uncurry

```
uncurry :: a -> b -> c -> ((a, b) -> c)
uncurry f t = f (fst t) (snd t)
```

Currying in Haskell

curry

```
curry :: ((a, b) -> c) a -> b -> c
curry f x y = f (x, y)
```

uncurry

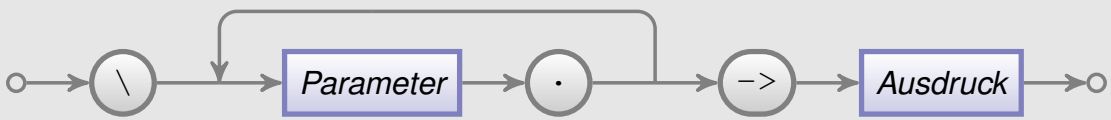
```
uncurry :: a -> b -> c -> ((a, b) -> c)
uncurry f t = f (fst t) (snd t)
```

- fst t gibt das erste Element aus t
- snd t gibt das zweite Element aus t

Lambda Ausdrücke

Anonyme Funktionen

- Haskell unterstützt anonyme Funktionen in Form von λ-Ausdrücken
- das λ-Symbol wird durch „\“ repräsentiert.
  $\lambda x \rightsquigarrow$ x
- Aufbau:
- durch das currying können λ-Ausdrücke mehrere Argumente besitzen
- es gelten alle bekannten Regeln für die λ-Notation



Beispiele - Lambda-Ausdrücke

```
plus = \x -> \y -> x + y

istKleiner = \x -> \y ->  x < y

g = \x -> \y -> (\y -> \x -> (y,x)) y x
```

Beispiele - Lambda-Ausdrücke

```
-- y "Operator"
y f = f (y f)

fac = y (\f n -> if n > 0 then n * f (n - 1) else 1)
```

Beispiele - Lambda-Ausdrücke

```haskell
-- y "Operator"
y f = f (y f)

fac = y (\f n -> if n > 0 then n * f (n - 1) else 1)
```

Aufruf

```haskell
fac 5
```

Beispiele - Lambda-Ausdrücke

```haskell
-- y "Operator"
y f = f (y f)

fac = y (\f n -> if n > 0 then n * f (n - 1) else 1)
```

Aufruf

```haskell
fac 5
```

Ausgabe

```
120
```

Ausblick für Morgen

Das Array

Monaden

IO

Technische Universität Braunschweig

Programmieren für Fortgeschrittene - eine Einführung in Haskell

Tag vier - ein bisschen noch

Stephan Mielke

28.03.2014

Das Array

das Array

- in Haskell existieren nicht nur Listen zur Speicherung und Verarbeitung von Daten sondern auch zwei Array Formen
- Arrays in Haskell besitzen immer eine feste Größe die bei der Erstellung angegeben wird

statische Arrays

- es muss das Modul *Data.Array.IArray* importiert werden
- die einzelnen Elemente eines Arrays werden mit dem ! Operator angesprochen
  z.B. *a*!5 gibt das Element mit dem Index 5 aus dem Array *a* wieder
- erstellt werden Arrays mit $(listArray(< Start >, < Ende >) < Liste >) :: Array < IndexTyp >< ElemTyp >$
  als Array Index kann jeder Datentyp verwendet werden, welcher die Typklasse *Ix* implementiert
- oder mit $(array(< Start >, < Ende >)[Liste\ mit\ Tupeln]) :: Array < IndexTyp >< ElemTyp >$
  in diesem Fall wird dem Array eine Liste mit Tupeln übergeben bei dem das erste Element der „Primarykey“ ist

statische Arrays

```
listArray :: (Ix i, IArray a e) => (i, i)
    -> [e] -> a i e
myArray1 = (listArray ('a','e') [10..15])
    :: Array Char Int
```

statische Arrays

```
listArray :: (Ix i, IArray a e) => (i, i)
    -> [e] -> a i e
myArray1 = (listArray ('a','e') [10..15])
    :: Array Char Int
```

```
array :: (Ix i, IArray a e) => (i, i)
    -> [(i, e)] -> a i e
myArray2 = (array (1,5) [(k,k*2)|
    k <- [1..5])::Array Int Int
```

Achtung

Die Anzahl der Listen Elemente und der Platz müssen nicht übereinstimmen, solange das Array nicht ausgegeben (*show a*) wird.

statische Arrays

```
accumArray :: (IX i, IArray a e) => (e -> e' -> e)
    -> e -> (i,i) -> [(i, e')] -> a i e

myArray3 = (accumArray (+) 0 (0,4) [(i `mod` 5
    , 1) | i <- [1..123]]) :: Array Int Int
```

statische Arrays

```
accumArray :: (IX i, IArray a e) => (e -> e' -> e)
    -> e -> (i,i) -> [(i, e')] -> a i e

myArray3 = (accumArray (+) 0 (0,4) [(i `mod` 5
    , 1) | i <- [1..123]]) :: Array Int Int
```

$$\mathit{array}\ (0,4)[(0,24),(1,25),(2,25),(3,25),(4,24)]$$

wichtige Array Funktionen

- *amap* ist die Array-Form der *map* Funktion für Listen
- *elems* wandelt das Array in eine Liste um (nur die Werte)
- *assocs* wandelt das Array in eine Liste von Tupeln der Form (k, v) um
- der $\backslash\backslash$ Operator (update) ändert in einem Array die gegebenen Wertpaare.

statische vs. dynamische Arrays

- bei einem Update mit dem \\ Operator (update) wird bei statischen das gesamte Array kopiert und die Änderungen vorgenommen
- somit dauert es bei statischen länger als bei dynamischen

dynamische Arrays

- import von *Data.Array.Diff*
- Funktionen heißen gleich nur Typ ist *DiffArray* statt *Array*
- besitzen zwar eine Konstante Zeit beim Update
- aber erhöhte Zugriffszeit beim Lesen
- durch geschickte Array Konstruktion kann jedoch fast vollständig auf Updates verzichtet werden

Haskell API

- für weitere Datentypen und deren Funktionen siehe: haskell.org/hoogle

Monaden

Monaten

- Monaden sind ein mathematisches Konzept aus der Kategorientheorie
- werden eingesetzt um Funktionen miteinander zu kombinieren
- ist in Haskell eine polymorphe Datenstrucktur mit speziellen Funktionen
- das Prinzip ist:
  - Sequenzialisierung gemäß des Continuation-style Programming der Kontrollfluss kehrt nicht zum Aufrufer zurück sondern geht zur Nachfolgefunktion
  - Darstellung und Transformation eines versteckten Zustands (Hiding)
  - Sicherung von Single-Threadedness dadurch, weil keine dagegen verstoßende Funktion benutzt werden kann

Monaden - Klasse

```
class  Monad m  where
    -- verbinden zweiter Funktionen
    -- Ergebnis ist Argument der zweiten Funktion
    (>>=)    :: forall a b. m a -> (a -> m b) -> m b
    -- verbindet zwei Funktionen aber verwirft jedes
    -- Ergebnis (wie in Imperativen Sprachen)
    (>>)     :: forall a b. m a -> m b -> m b
    m >> k   = m >>= \_ -> k
    -- fuegt einen Wert in den Monaden Typ ein
    return   :: a -> m a
    -- gibt eine Fehlernachricht zurueck
    fail     :: String -> m a
    fail     = error
```

Monaden - Klasse

```
add :: Maybe Int -> Maybe Int -> Maybe Int
add mA mB = case mA of
     Nothing -> Nothing
     Just a  -> case mB of
                  Nothing -> Nothing
                  Just b  -> Just (a + b)
```

Monaden - Klasse

```haskell
add :: Maybe Int -> Maybe Int -> Maybe Int
add mA mB = case mA of
    Nothing -> Nothing
    Just a  -> case mB of
                 Nothing -> Nothing
                 Just b  -> Just (a + b)
```

```haskell
add' :: Maybe Int -> Maybe Int -> Maybe Int
add' mA mB = mA >>= (\a ->
               mB >>= (\b ->
                 return (a + b)))
```

do-Notation

- mit der do-Notation werden Monaden ($>>=$) zusammengefasst pro Zeile
- somit ist es syntaktischer Zucker
- für die Verwendung der do-Notation sind 4 Regeln zu beachten

do-Notation - Regel 1

- einzelne Anweisungen benötigen keine Umformung.
- das *do* wird einfach weggelassen.

```
do
  e
```

do-Notation - Regel 1

- einzelne Anweisungen benötigen keine Umformung.
- das *do* wird einfach weggelassen.

```
do
  e
```

```
e
```

do-Notation - Regel 2

- wird der Rückgabewert nicht benötigt
- dann wird die Anweisung nach vorne gezogen

```
do
  e
  <Anweisung>
```

do-Notation - Regel 2

- wird der Rückgabewert nicht benötigt
- dann wird die Anweisung nach vorne gezogen

```
do
  e
  <Anweisung>
```

```
e >>= \_ -> do
  <Anweisungen>
```

do-Notation - Regel 3

- wird der Rückgabewert mit Pattern-Matching ausgewertet
- dann muss eine Hilfsfunktion dies übernehmen

```
do
  pattern <- e
  <Anweisungen>
```

do-Notation - Regel 3

- wird der Rückgabewert mit Pattern-Matching ausgewertet
- dann muss eine Hilfsfunktion dies übernehmen

```
do
  pattern <- e
  <Anweisungen>
```

```
let ok pattern = do
    <Anweisungen>
  ok _ = fail "Fehler"
in e >>= ok
```

do-Notation - Regel 4

- wird ein Wert mit let gespeichert, kann dies vor das *do* gezogen werden
- das *in* ist im do-Block optional

```
do
  let <Deklaration>
  in <Anweisungen>
```

do-Notation - Regel 4

- wird ein Wert mit let gespeichert, kann dies vor das *do* gezogen werden
- das *in* ist im do-Block optional

```
do
  let <Deklaration>
  in <Anweisungen>
```

```
let <Deklaration>
in do
     <Anweisungen>
```

do-Notation - If-Then-Else

wir erwarten:

```
f = do
  if <irgendwas> then
    <Anweisungen>
  else
    <Anweisungen>
```

Aber !

```
f = do
  if <irgendwas> then do
    <Anweisungen>
  else do
    <Anweisungen>
```

do-Notation - Beispiel

```
add' :: Maybe Int -> Maybe Int -> Maybe Int
add' mA mB = mA >>= (\a ->
                mB >>= (\b ->
                  return (a + b)))
```

do-Notation - Beispiel

```
add' :: Maybe Int -> Maybe Int -> Maybe Int
add' mA mB = mA >>= (\a ->
                mB >>= (\b ->
                  return (a + b)))
```

```
add :: Maybe Int -> Maybe Int -> Maybe Int
add mA mB = do
  a <- mA
  b <- mB
  return (a + b)
```

vordefinierte Monaden

- Writer - für Debug / Logging / Tracing
- Reader - zum Lesen von gemeinsamen Zuständen (global)
- State - Verknüpfung von Writer und Reader für gemeinsame Zustände für zustandsbasierte Rechnungen

IO

Hangman

```
import Data.Char
import System.IO

w = "Lambda"
maxI = 5

hangman :: String -> Int -> IO ()
hangman cs i | i > maxI = putStrLn "\nverloren"
             | all (`elem` cs) (map toLower w) =
               putStrLn "\ngewonnen"
```

Hangman

```
hangman cs i =
  do
    putStrLn "␣"
    printWord cs
    putStrLn "\nWelcher␣Buchstabe?"
    c <- getChar >>= (return.toLower)
    if (c `elem` (map toLower w)) then
      hangman (c:cs) i
    else do
      putStrLn $ "\n" ++ (show (i+1)) ++ "␣falsch\n"
      hangman cs (i+1)
```

Hangman

```
printWord :: String -> IO ()
printWord cs = mapM_ pC w
  where
    pC x  | toLower x `elem` cs = putChar x
          | otherwise = putChar '_'


main = do
  hSetBuffering stdin NoBuffering
  hangman "␣" 0
```

das O in IO

- $print :: Show\ a \Rightarrow a \rightarrow IO\ ()$
  gibt jeden Datentyp der *Show* implementiert aus
- $putChar :: Char \rightarrow IO\ ()$
  gibt ein Char aus
- $putStr :: String \rightarrow IO\ ()$
  gibt einen String aus
  $putStr = sequence\_ \cdot map\ putChar$

das O in IO

- $writeFile :: FilePath \rightarrow String \rightarrow IO\ ()$
- $type\ Filepath = String$
- schreibt den „String“ mittels Textstrom in eine Datei

das I in IO

- *readLn* :: *Read a* => *IO a*
  liest jeden Datentyp der *Read* implementiert ein
- *getChar* :: *IO Char*
  liest ein Char ein
- *getLine* :: *IO String*
  liest die ganze Zeile ein als String
- die Pufferung der Eingaben ist über *hSetBuffering* einstellbar
  für Windows bekommt es der GHC trotzdem nicht hin :(

das O in IO

- $readFile :: FilePath \rightarrow IO\ String$
- $type\ Filepath = String$
- liest die Datei als String ein

Hangman

```
isInfix :: String -> String -> Maybe String
isInfix [] _ = Nothing
isInfix t w | w == take (length w) t = Just t
            | otherwise = isInfix (tail t) w
```

Hangman

```
main = do
  putStrLn "Dateipfad: "
  filepath <- getLine
  putStrLn "gesuchtes Wort: "
  w <- getLine
  c <- readFile filepath
  case (isInfix c w) of
    Nothing -> putStrLn "nicht enthalten"
    Just s -> putStrLn $
        "an Stelle " ++ (take 100 s)
```

Zusammenfassung

- ab jetzt seid ihr auf dem Wissenstand, auf dem ich bin

- nun seid ihr dran
  - stellt Fragen
  - schlagt auf www.haskell.org/hoogle nach
  - oder vergesst alles schnell wieder