Programmieren für Fortgeschrittene - eine Einführung in Haskell

Teil eins - Was ist Haskell

Stephan Mielke, 17.11.2014

Technische Universität Braunschweig, IPS

Überblick

- Das funktionale Paradigma
- Haskell
- Semantische Grundbegriffe
- Ausdrücke
- Einfache Datentypen

Organisatorisches

- Jeweils Montags von 9:45 - 11:15 in IZ 261 (IPS Terminal Raum)
- 5 / 6 Termine am:
  - Fest: 17.11.2014, 01.12.2014, 15.12.2014
  - Variabel: 12.01.2015, 26.01.2015
  - Oder: 05.01.2015, 19.01.2015, 02.02.2015
- Heute komplett „Vorlesung“, sonst 30-45 Minuten und danach Übung

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

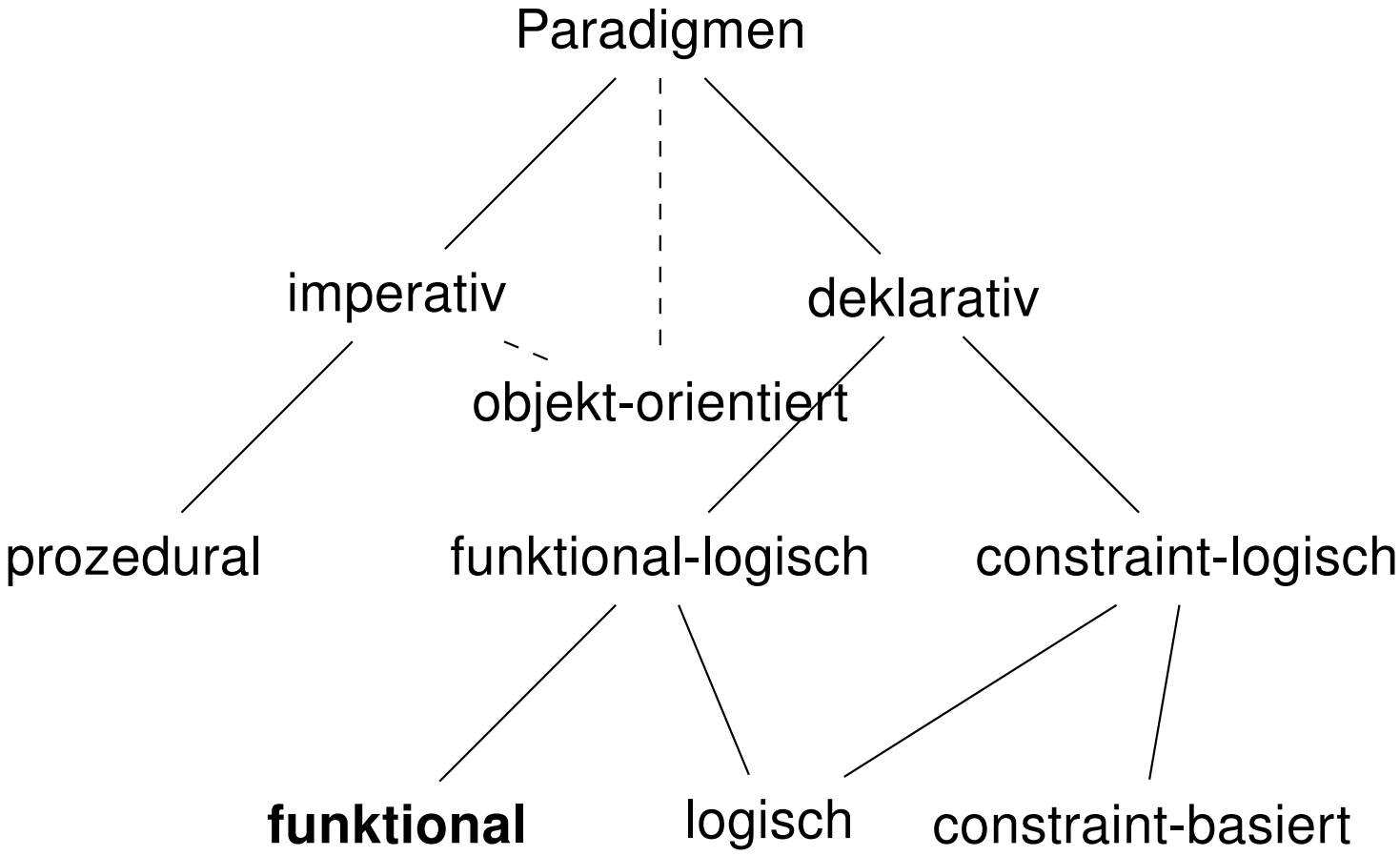
Überblick

- **Das funktionale Paradigma**
- Haskell
- Semantische Grundbegriffe
- Ausdrücke
- Einfache Datentypen

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

Überblick

- Das funktionale Paradigma
- **Haskell**
- Semantische Grundbegriffe
- Ausdrücke
- Einfache Datentypen

Haskell

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

Hello World

```haskell
module Main where

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

Hello World

```haskell
module Main where

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

Ausgabe

Hello, World!

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

Glasgow Haskell Compiler

- Original Prototyp ’89 in LML (Lazy ML)
- Bei der Entwicklung von Haskell in Haskell (API) 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 „fast“ unabhängig
- Vergleichbar mit Java
  - JVM in C bzw. C++
  - API in Java
  - Bytecode als Zwischencode

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

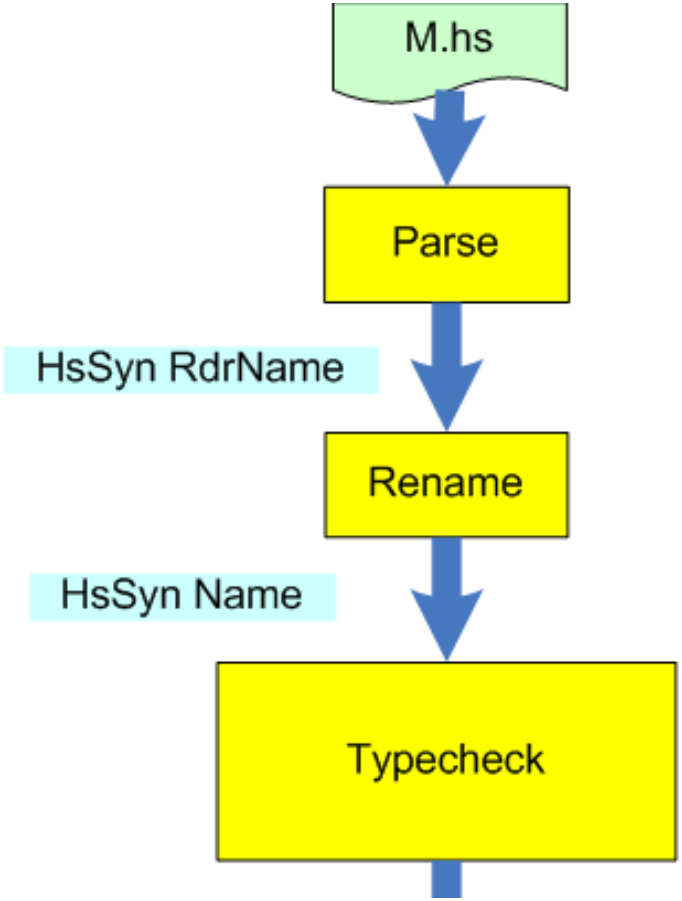


Abbildung 1: Compiler Teil 1 ©haskell.org

Interne Funktionsweise

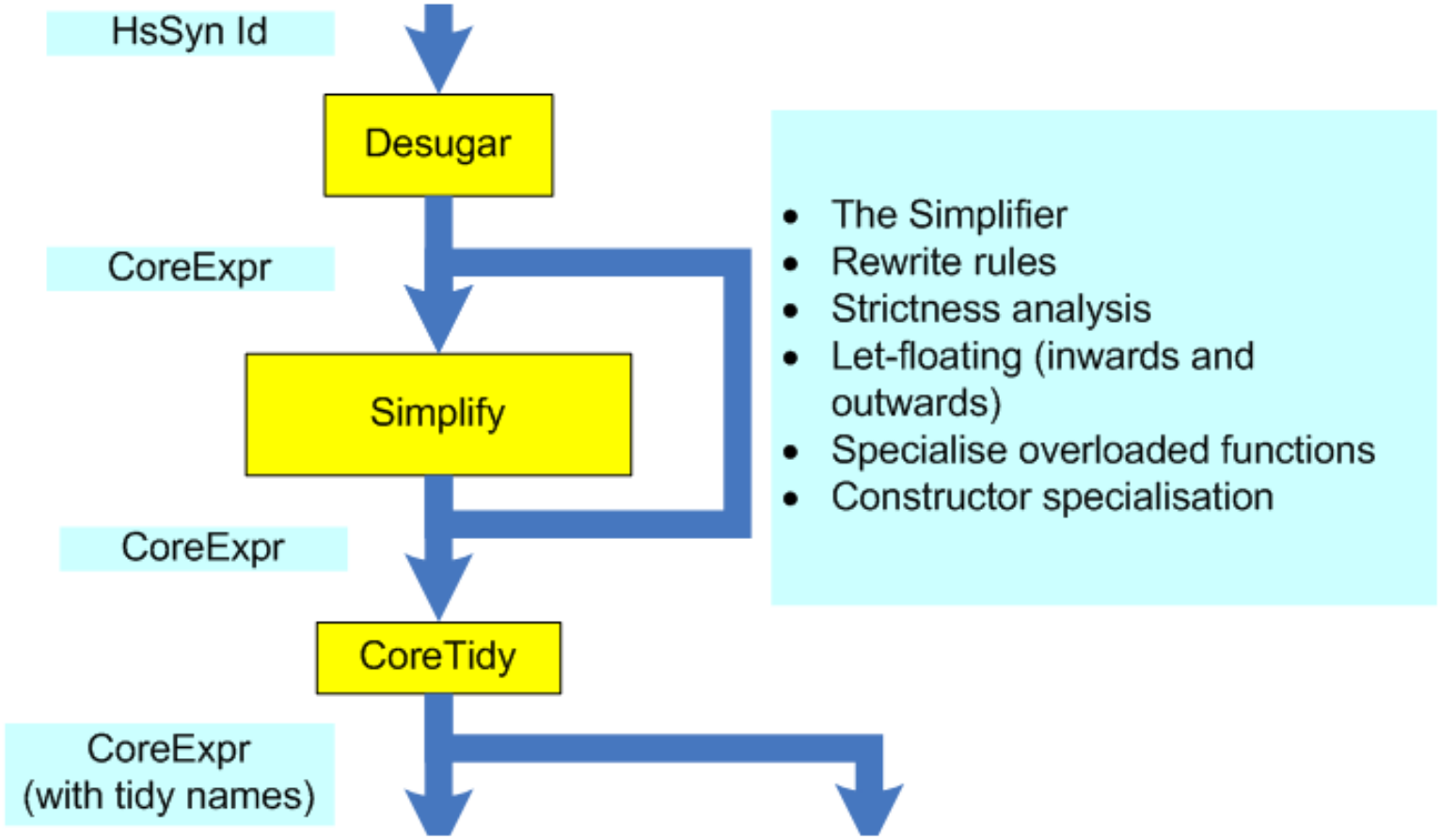


Abbildung 2: Compiler Teil 2 ©haskell.org

Interne Funktionsweise

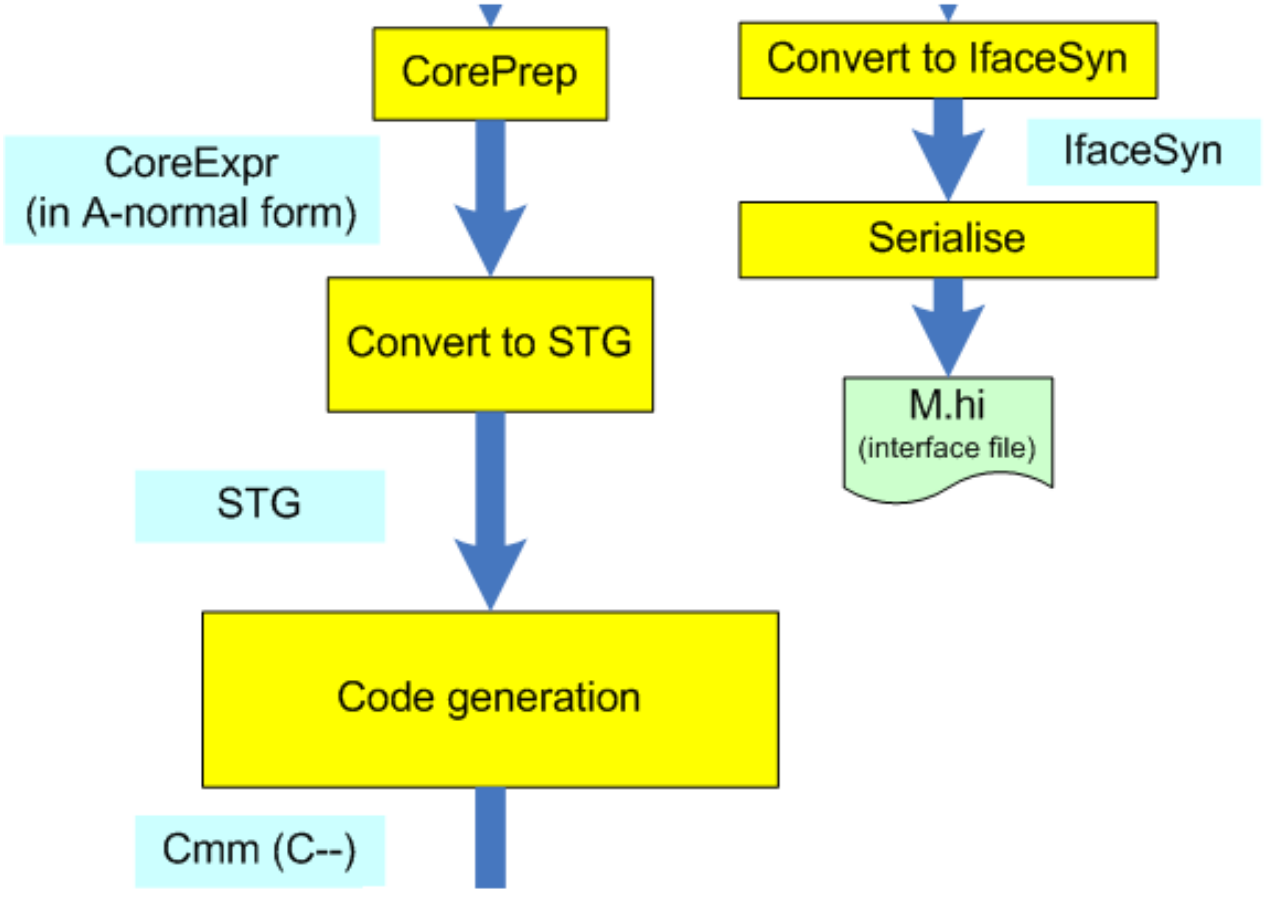


Abbildung 3: Compiler Teil 3 ©haskell.org

Interne Funktionsweise

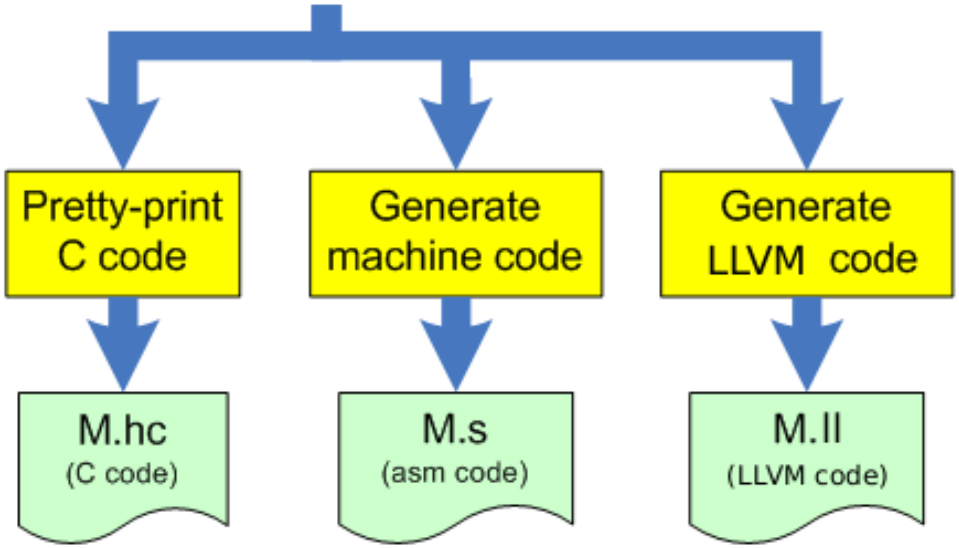


Abbildung 4: Compiler Teil 4 ©haskell.org

Überblick

- Das funktionale Paradigma
- Haskell
- **Semantische Grundbegriffe**
- Ausdrücke
- Einfache Datentypen

Semantische Grundbegriffe - Namen und Attribute

- **Semantische Grundbegriffe**
  - Namen und Attribute
  - Variablen und Konstanten
  - Ausdrücke
  - Funktionen
  - Blöcke

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` ≠ `plus`)
- _ (Unterstrich) ist der Platzhalter
- Module beginnen mit einem Großbuchstaben
- Funktionen mit einem Kleinbuchstaben

Semantische Grundbegriffe - Variablen und Konstanten

- **Semantische Grundbegriffe**
  - Namen und Attribute
  - Variablen und Konstanten
  - Ausdrücke
  - Funktionen
  - Blöcke

Variablen

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

Konstanten

- Konstanten sind Funktionen ohne Parameter
- Mathematisch: $\emptyset \rightarrow |W_f| = 1$

Semantische Grundbegriffe - Ausdrücke

- **Semantische Grundbegriffe**
  - Namen und Attribute
  - Variablen und Konstanten
  - Ausdrücke
  - Funktionen
  - Blöcke

Ausdrücke

Elementare Ausdrücke bzw. Grundterme setzen sich zusammen aus:

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

Infixnotation

Funktionszeichen: `3 + 4` $\equiv$ `(+) 3 4`

Funktionsname: `mod 100 4` $\equiv$ `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“
- $\lambda a \rightarrow \lambda b \rightarrow a + b$ 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
```

Semantische Grundbegriffe - Funktionen

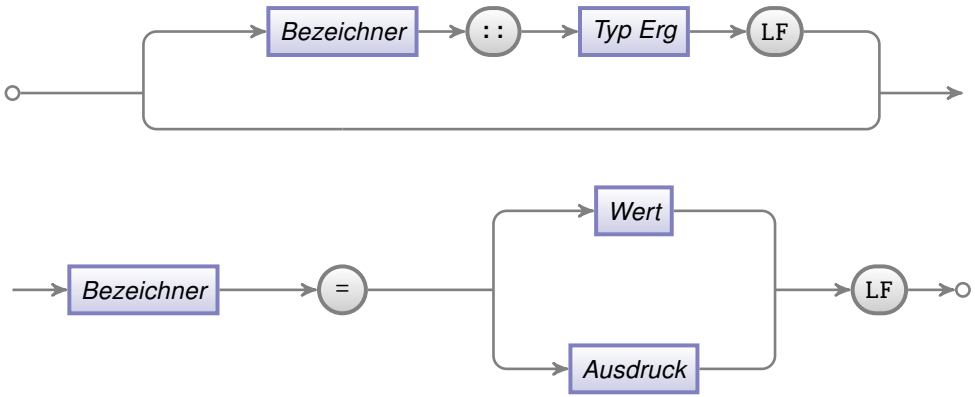
- **Semantische Grundbegriffe**
  - Namen und Attribute
  - Variablen und Konstanten
  - Ausdrücke
  - Funktionen
  - Blöcke

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 $(a, b_1) \in R_f$ 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

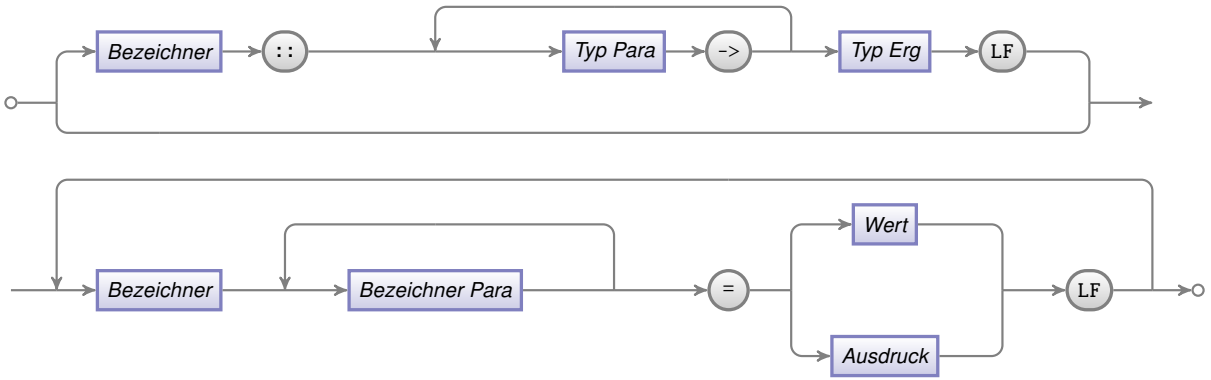
Konstante



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

Deklaration von Funktionen

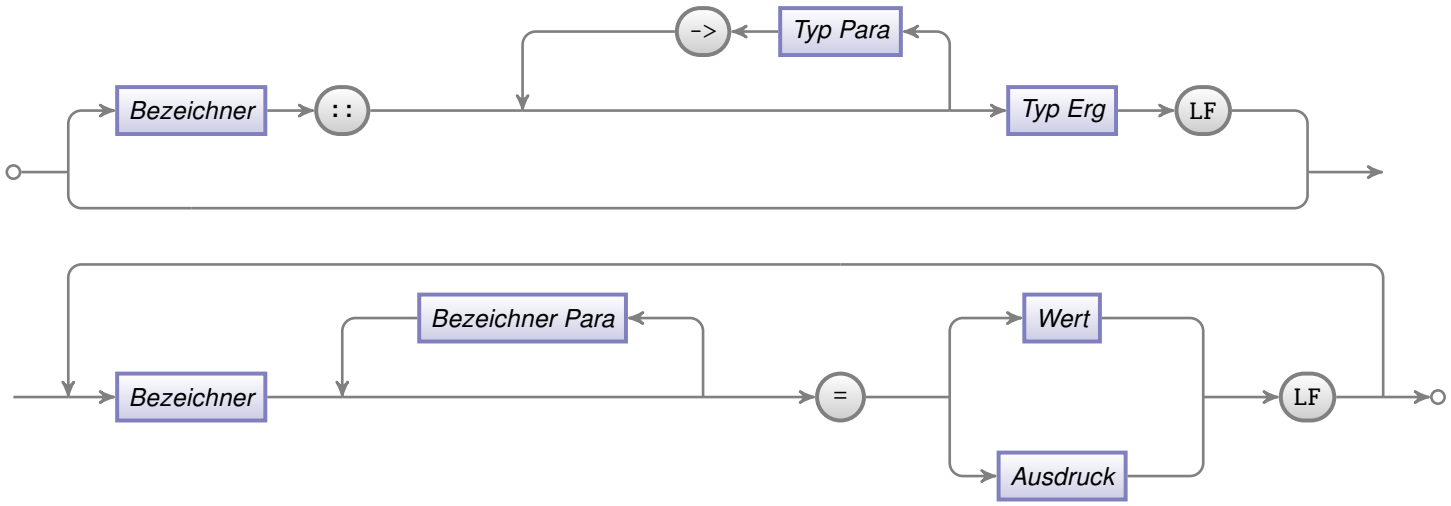
Funktion



- 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

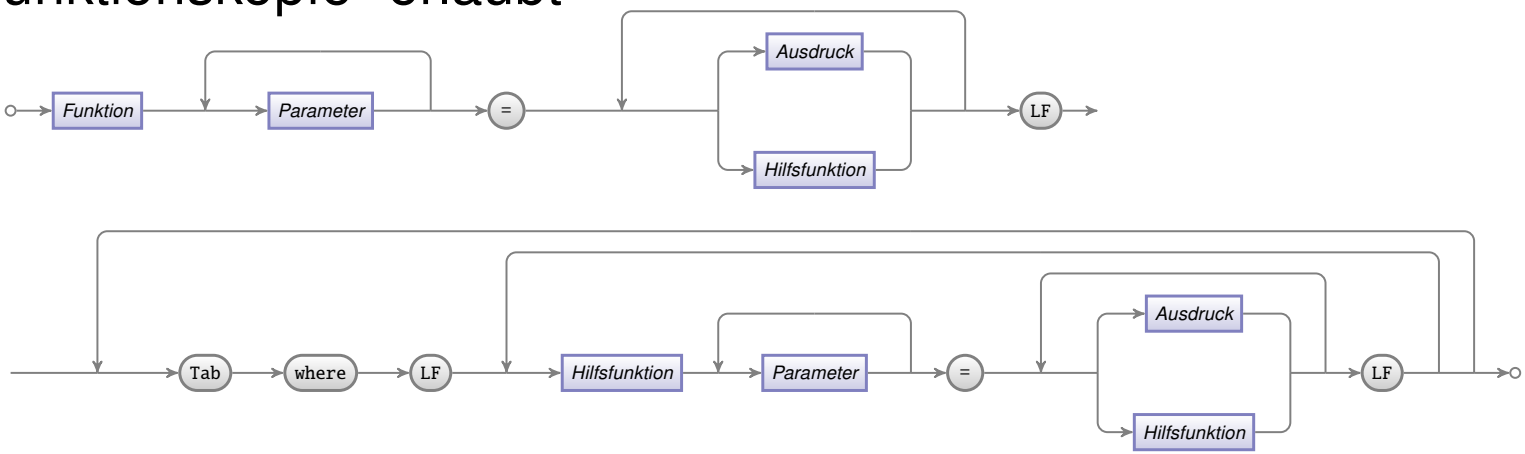
- **Semantische Grundbegriffe**
  - Namen und Attribute
  - Variablen und Konstanten
  - Ausdrücke
  - Funktionen
  - Blöcke

Der where-Block

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

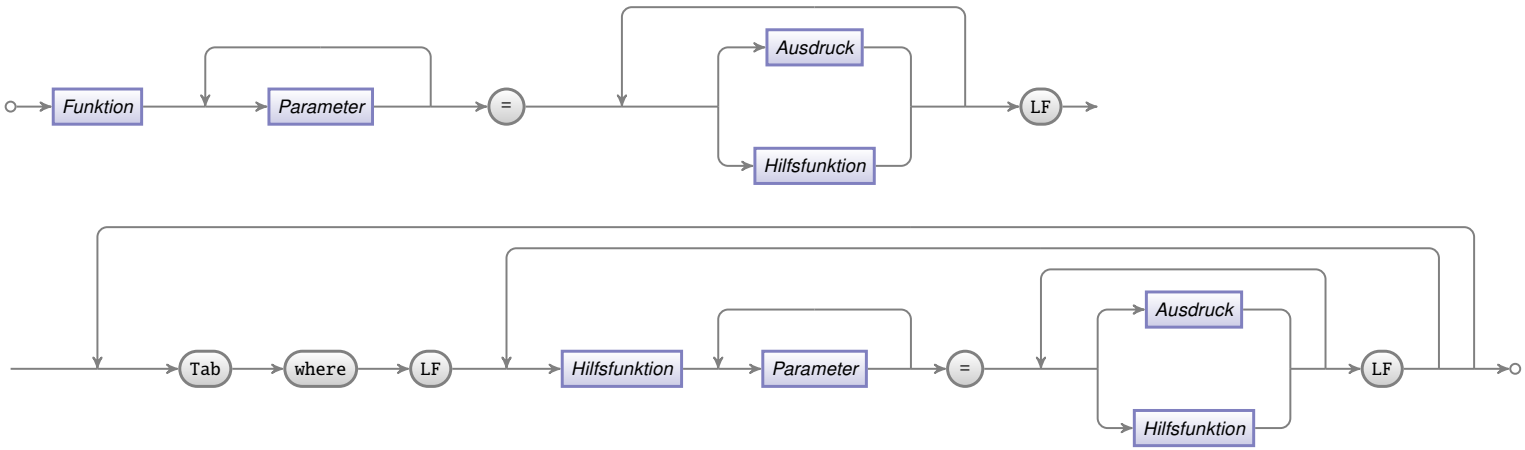
Der where-Block

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



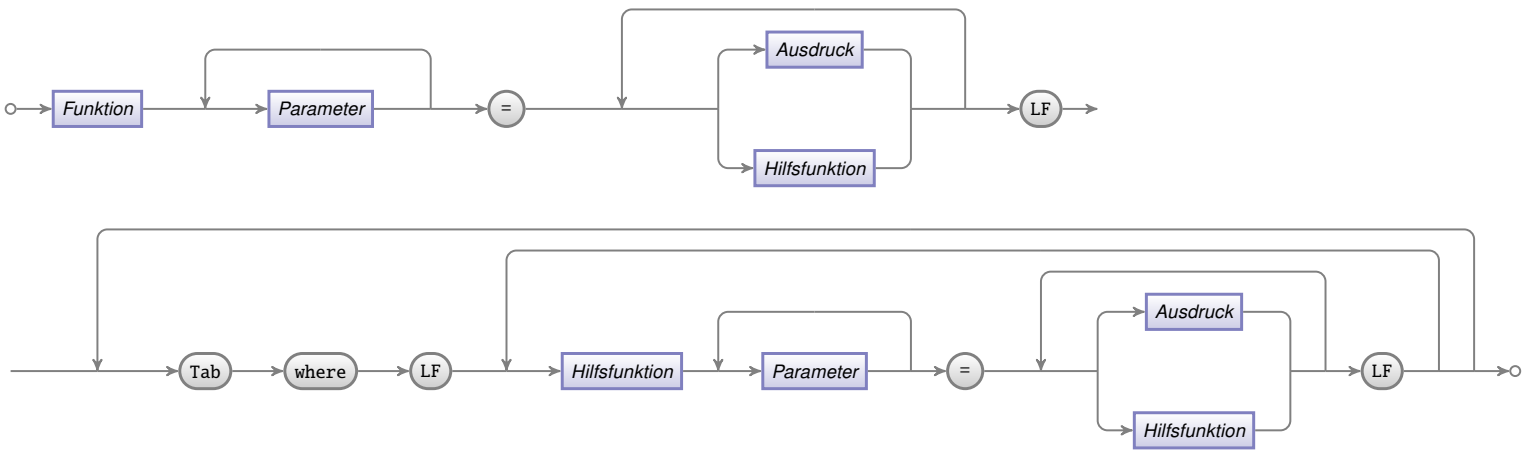
Der where-Block

```
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
```

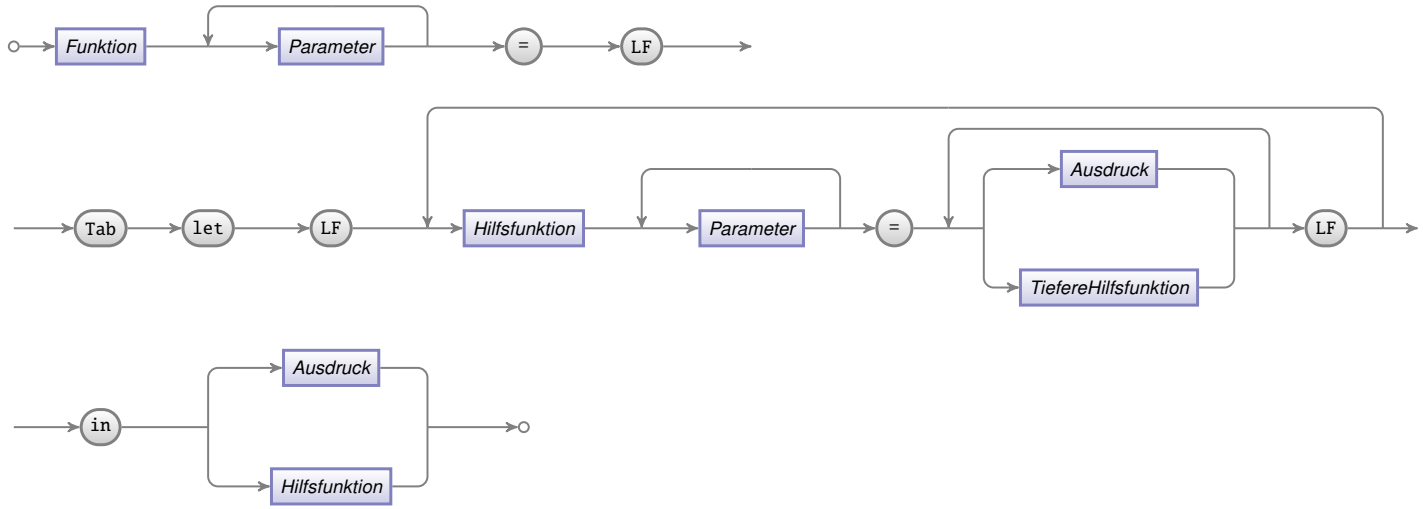
Ausgabe

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
```

Überblick

- Das funktionale Paradigma
- Haskell
- Semantische Grundbegriffe
- **Ausdrücke**
- Einfache Datentypen

Ausdrücke - Ausdrücke allgemein

- **Ausdrücke**
  - Ausdrücke allgemein
  - Lambda - Ausdruck
  - Strukturierte Anweisungen
  - Iterationsanweisungen
  - Sprunganweisungen

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
  - Variable 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 - Ausdruck

- **Ausdrücke**
  - Ausdrücke allgemein
  - Lambda - Ausdruck
  - Strukturierte Anweisungen
  - Iterationsanweisungen
  - Sprunganweisungen

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

- **Ausdrücke**
  - Ausdrücke allgemein
  - Lambda - Ausdruck
  - Strukturierte Anweisungen
  - Iterationsanweisungen
  - Sprunganweisungen

Strukturierte Anweisungen

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

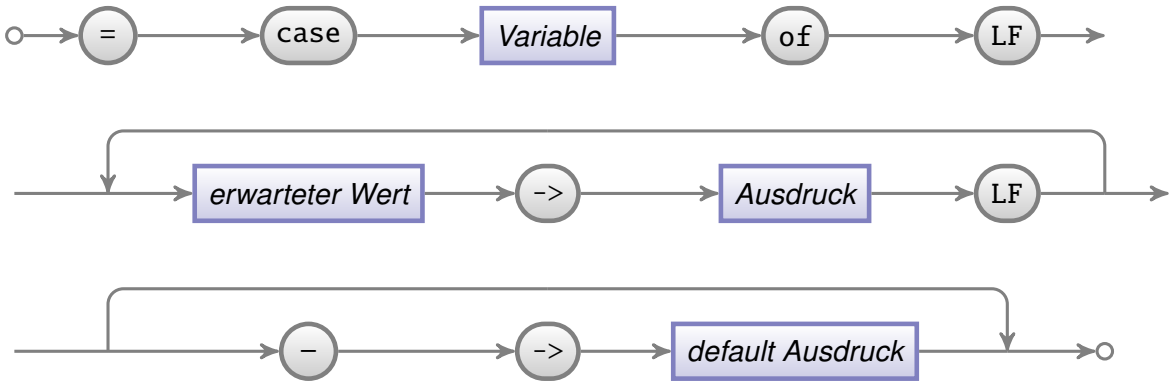
If-Then-Else

- 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 Datentyps
- Erster „Treffer“ gewinnt

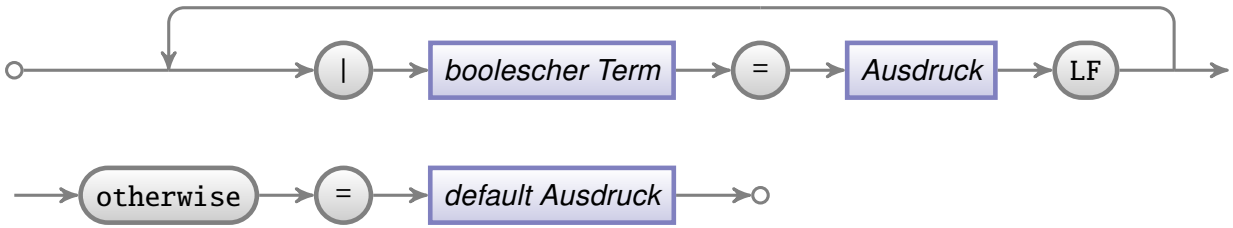
Pattern-Matching

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

```haskell
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

If-Then-Else

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

Sturkturierte Anweisungen Beispiel ggT

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

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

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

- **Ausdrücke**
  - Ausdrücke allgemein
  - Lambda - Ausdruck
  - Strukturierte Anweisungen
  - Iterationsanweisungen
  - Sprunganweisungen

Iterationsanweisungen

- In Haskell existieren keine Schleifen wie
  `while`, `do while`, `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

- **Ausdrücke**
  - Ausdrücke allgemein
  - Lambda - Ausdruck
  - Strukturierte Anweisungen
  - Iterationsanweisungen
  - Sprunganweisungen

Sprunganweisungen

Es existieren keine Sprunganweisungen

Überblick

- Das funktionale Paradigma
- Haskell
- Semantische Grundbegriffe
- Ausdrücke
- **Einfache Datentypen**

Einfache Datentypen

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

Einfache Datentypen - Wahrheitswerte

- **Einfache Datentypen**
  - Wahrheitswerte
  - Ganzzahlen
  - Gleitkommazahl
  - Zeichen

Bool

- Einfacher Wahrheitswert
- `True` oder `False`

Bezeichner	Typ	Bedeutung
`not`	`Bool -> Bool`	Verneinung
`(||)`	`Bool -> Bool -> Bool`	Oder
`or`	`[Bool] -> Bool`	Oder auf Listen
`(&&)`	`Bool -> Bool -> Bool`	Und
`and`	`[Bool] -> Bool`	Und auf Listen
`(==)`	`Bool -> Bool -> Bool`	Gleich
`(/=)`	`Bool -> Bool -> Bool`	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

- **Einfache Datentypen**
  - Wahrheitswerte
  - Ganzzahlen
  - Gleitkommazahl
  - Zeichen

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 unendlichem 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

```haskell
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

- **Einfache Datentypen**
  - Wahrheitswerte
  - Ganzzahlen
  - Gleitkommazahl
  - Zeichen

Float - Double

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

Einfache Datentypen - Zeichen

- **Einfache Datentypen**
  - Wahrheitswerte
  - Ganzzahlen
  - Gleitkommazahl
  - Zeichen

Char

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

Danke

Vielen Dank für die Aufmerksamkeit und das Interesse!