Programmieren für Fortgeschrittene - eine Einführung in Haskell

Teil zwei - etwas mehr

Stephan Mielke, 01.12.2014

Technische Universität Braunschweig, IPS

Überblick

- Gültigkeitsbereiche
- Überladung und Auflösung von Namen
- Listen – ganz kurz
- Currying

Überblick

- **Gültigkeitsbereiche**
- Überladung und Auflösung von Namen
- Listen – ganz kurz
- Currying

Gültigkeitsbereiche - Block

- **Gültigkeitsbereiche**
  - Block
  - Module

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

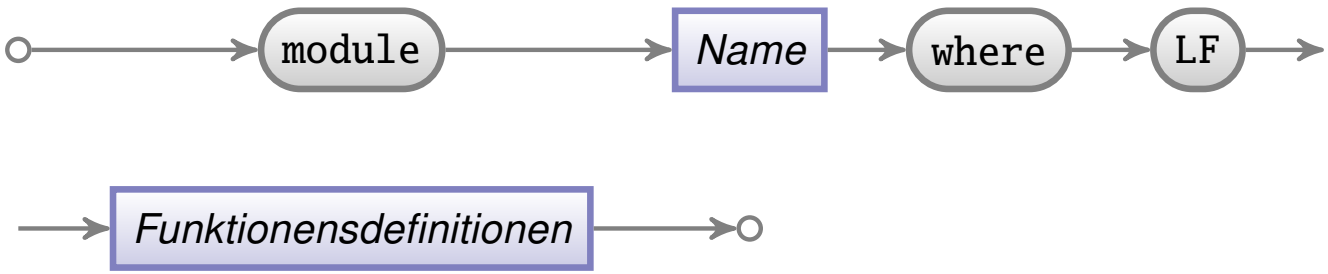
Gültigkeitsbereiche - Module

- **Gültigkeitsbereiche**
  - Block
  - 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



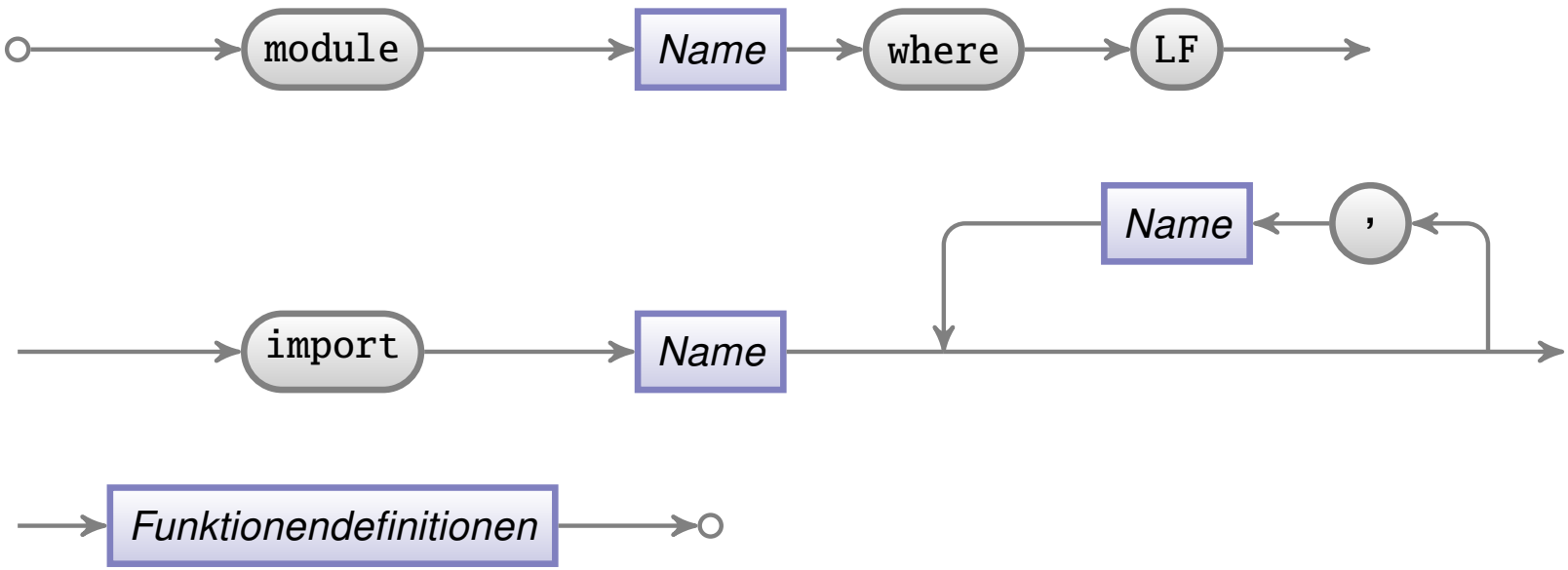
Module

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

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

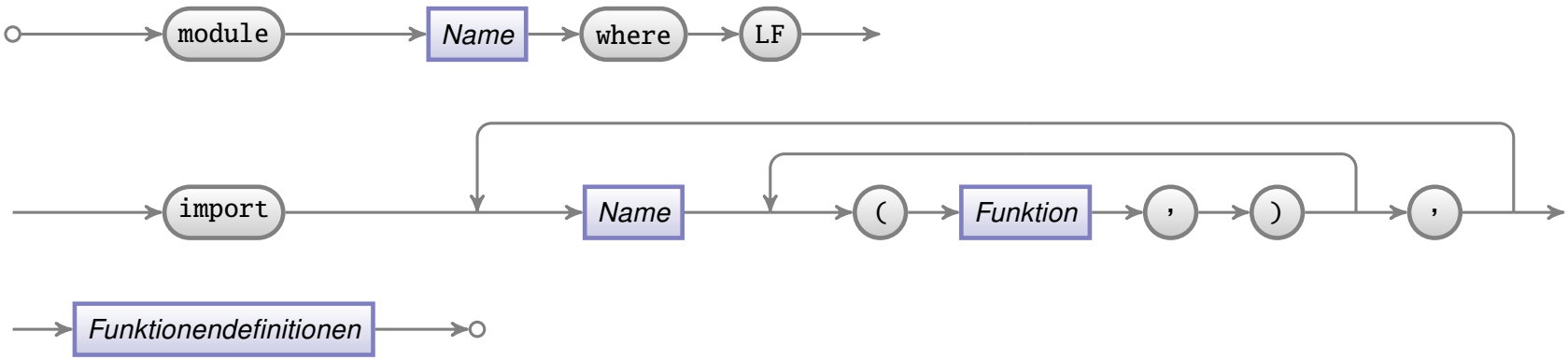
Module - Interfaces

Import



Module - Interfaces

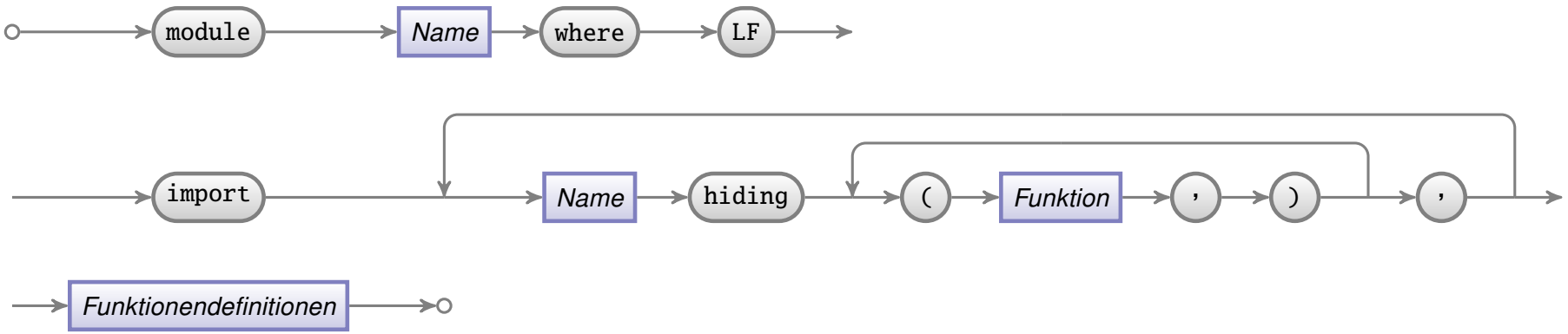
Selektiver Import



Am Ende steht natürlich kein Komma

Module - Interfaces

Negativ selektiver Import



Am Ende steht natürlich kein Komma

Module - Interfaces

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

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

Achtung

`square` ist für `bar` nicht definiert!

Module - Interfaces

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

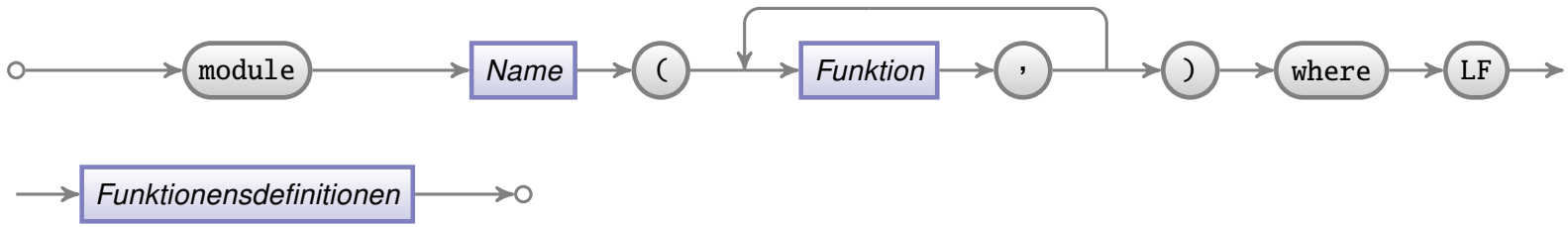
```haskell
module Foo where
import Wurf hiding (weite)
foo ... = ... (weite v w) ...
bar ... = ... (square a) ...
```

Achtung

`weite` ist für `foo` nicht definiert!

Module - Sichtbarkeit

Module können festlegen was importiert werden darf



Am Ende steht natürlich kein Komma

Module - Sichtbarkeit

```haskell
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
```

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

Achtung

In `Wurf` ist nur `weite` sichtbar

Überblick

- Gültigkeitsbereiche
- **Überladung und Auflösung von Namen**
- Listen – ganz kurz
- Currying

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

```haskell
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

Überblick

- Gültigkeitsbereiche
- Überladung und Auflösung von Namen
- **Listen – ganz kurz**
- Currying

Listen

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

- Listen sind Folgen von Elementen gleichen Typs
- `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 Typs
- `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

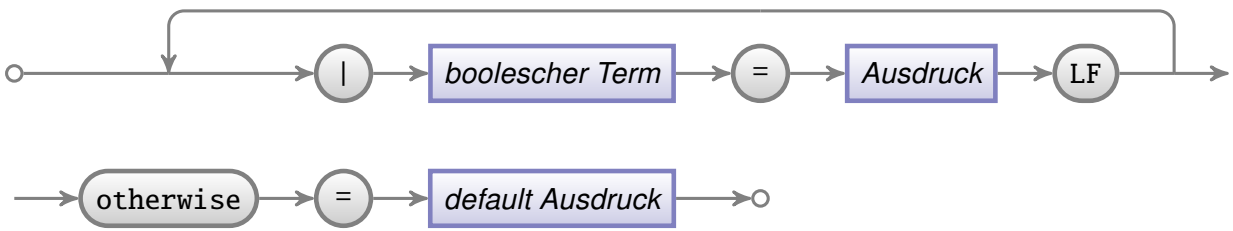
```haskell
length :: [Int] -> Int
length []     = 0
length (_:xs) = 1 + length xs

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
```

Überblick

- Gültigkeitsbereiche
- Überladung und Auflösung von Namen
- Listen – ganz kurz
- **Currying**

Currying

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

Deklaration von Funktionen in λ Notation

Lambda Currying

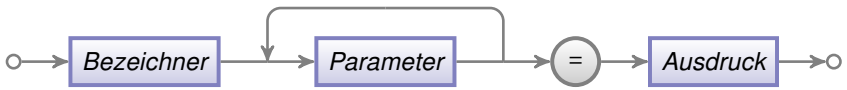
$$f = \lambda x_1 \rightarrow \lambda x_2 \rightarrow \cdots \rightarrow \lambda x_n \rightarrow e$$

Deklaration von Funktionen in λ Notation

Lambda Currying

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

Funktions Currying

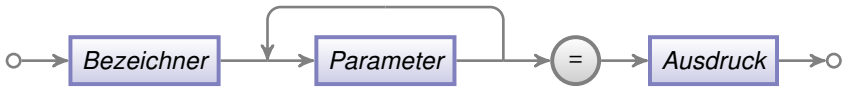


Deklaration von Funktionen in λ Notation

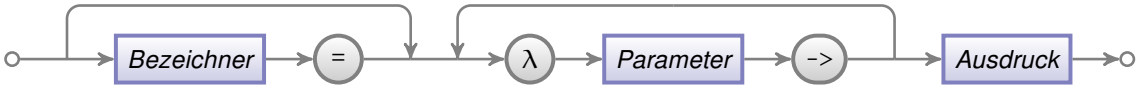
Lambda Currying

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

Funktions Currying



Lambda Currying



Deklaration von Funktionen in λ Notation

Lambda Uncurrying

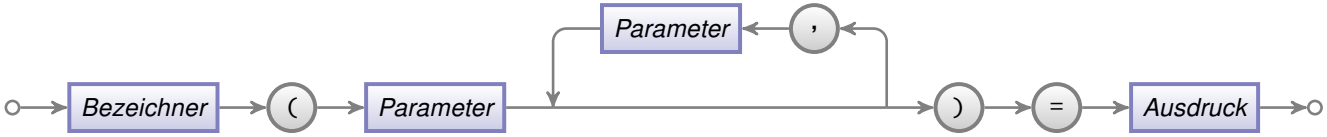
$$f = \lambda(x_1, x_2, \ldots, x_n) \rightarrow e$$

Deklaration von Funktionen in λ Notation

Lambda Uncurrying

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

Funkitons Uncurrying



ABER

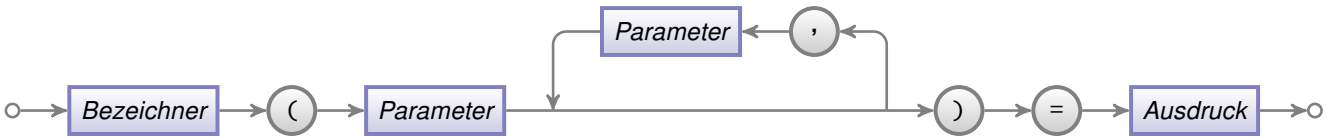
Das Tupel $(x_1, x_2, \ldots, x_n)$ ist ein eigener Datentyp
Deswegen nie Funktionsargumente klammern und mit Kommata trennen!

Deklaration von Funktionen in λ Notation

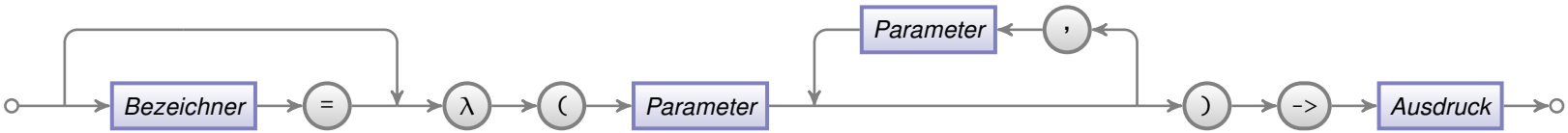
Lambda Uncurrying

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

Funkitons Uncurrying



Lambda Uncurrying



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)True
```

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 $\leqslant 1$ sein
- `f :: Int -> Int -> Int`
- Hierfür kommen Tupel ins Spiel

Currying in Haskell

- Soll Currying unterbunden werden, so muss die Anzahl der Argumente von Anfang an $\leqslant 1$ sein
- `f :: Int -> Int -> Int`
- Hierfür 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 $\leqslant 1$ sein
- `f :: Int -> Int -> Int`
- Hierfür 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)
```

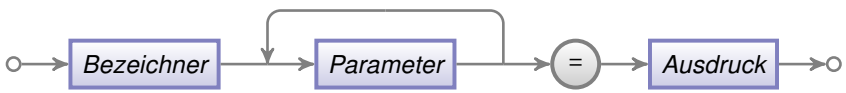
uncurry

```
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`

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

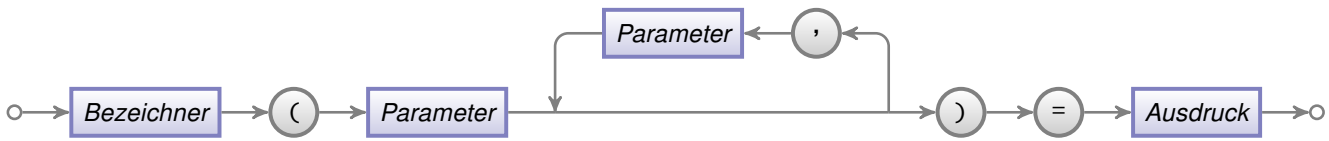
```
plus 6 7
```

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
```

Danke

Vielen Dank für die Aufmerksamkeit und das Interesse!