Haskell

Funktionale Programmierung

Malte Voß

true = $\lambda x \rightarrow \lambda y \rightarrow x$

vossmalte.github.io

Grundlagen: Erste Schritte

```
z = 2
f x = 2*x + 3
meineListe = [1,1,2,3,5]
```

if-then-else

```
f x y = if x >= y then x else y
```

Was macht `f`?

if-then-else

```
f x y = if x >= y then x else y
```

Was macht `f`? Das Maximum von `x,y` berechnen.

switch-case

switch-case

```
max3 x y z
  | x >= y && x >= z = x
  | y >= x && y >= z = y
  |otherwise = z
```

switch-case

```
max3 x y z
  | x >= y && x >= z = x
  | y >= x && y >= z = y
  |otherwise = z
```

Der erste passende Fall wird ausgewertet.
Diese Struktur wird guards genannt.

Haskell ausführen

Datei anlegen: `programm.hs`

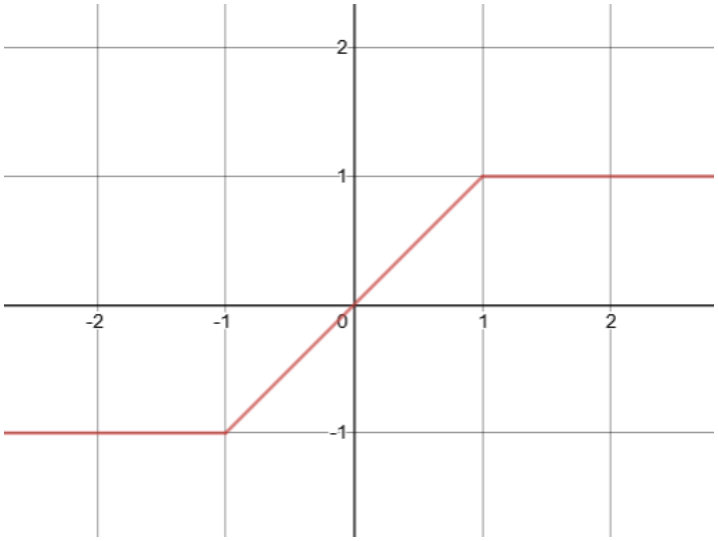
Haskell ausführen

Datei anlegen: `programm.hs`
Im Terminal:

- zur Datei navigieren: `cd directory`
- Interaktiven Compiler aufrufen: `ghci`
- Programm laden: `:l programm.hs`
- Programm neu laden: `:r`

Aufgabe: Funktion definieren

Graph von $f(x)$



Definiere diese Funktion in Haskell. Verwende if-then-else oder guards.

Basisfälle

```
lucky 7 = "LUCKY␣NUMBER␣SEVEN!"
lucky x = "Sorry,␣you're␣out␣of␣luck,␣pal!"
```

Basisfälle

```
lucky 7 = "LUCKY␣NUMBER␣SEVEN!"
lucky x = "Sorry,␣you're␣out␣of␣luck,␣pal!"
```

Das Konzept heißt Pattern-Matching. Mehr dazu bei Listen.

Rekursion

Rekursion allgemein: Basisfall und allgemeiner Fall.

Rekursion

Rekursion allgemein: Basisfall und allgemeiner Fall.

```
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

Prefix- / Infixnotation

Im Allgemeinen verwenden wir Prefix-Notation: `max 255 256`

Prefix- / Infixnotation

Im Allgemeinen verwenden wir Prefix-Notation: `max 255 256`
Besondere Funktionen werden infix notiert: `13+29`
Man kann sie aber auch jeweils andernorts verwenden:

```
7 `mod` 3
```

```
(+) 13 29
```

Grundlagen: Aufgaben

Schreibe eine Funktion...

- welche die ganzzahlige Potenz a^b berechnet
- die für a, b angibt, ob a durch b teilbar ist
- zur Berechnung des ggT mit Hilfe des euklidischen Algorithmus
- die prüft, ob eine Zahl prim ist
- welche die ganzzahlige Potenz geschickt berechnet: $a^{2b} = (a^2)^b$

Funktionale Programmierung

Alles ist Funktion*.

`x = 3` ist eine parameterlose Funktion.

Es gibt keine Nebeneffekte (außer IO)

Typen untersuchen

```
:t max
max :: Ord a =>a -> a -> a
```

Bedeutung?

Typen untersuchen

```
:t max
max :: Ord a =>a -> a -> a
```

Bedeutung? a ist ein Typ und hat eine **Ord**nung.

Typen untersuchen

```
:t max
max :: Ord a =>a -> a -> a
```

Bedeutung? a ist ein Typ und hat eine **Ord**nung.

`max` nimmt ein solches a und ein weiteres und gibt etwas vom Typ a aus.

Typen untersuchen 2

```
f x = x*x+1
:t f
f :: Num a => a -> a
```

Bedeutung?

Listen - Einstieg

```
[1,2,3,4,5]
1:2:3:4:5:[]
[1..5]
```

Listen - Einstieg

```
[1,2,3,4,5]
1:2:3:4:5:[]
[1..5]
```

Was ist der Typ von `(:)`?

Listen - Einstieg

```
[1,2,3,4,5]
1:2:3:4:5:[]
[1..5]
```

Was ist der Typ von `(:)`?

Auf ein Element der Liste zugreifen: `[1,1,2,3,5,8] !! 3`

List comprehension

```haskell
[x*x | x<-[1..30], mod x 2 == 0]
```

Pattern matching

```
summiere [] = 0
summiere (x:rest) = x + (summiere rest)
```

Zerteilung der Liste in Kopf und Rest.

Unendlichkeit

`[1..]`

Man kann unendliche Listen *definieren*,
man sollte sie aber lieber nicht ganz *aufrufen*.

```
take 11 [1..]
```

Operatoren auf Listen

`head`, `tail`, `last`, `init`, `take`, `drop`

Operatoren auf Listen

`head, tail, last, init, take, drop`

- **`map`**
- **`filter`**
- **`foldl`**
- **`zip`**
- **`zipWith`**

```
map odd [1..20]
```

Operatoren auf Listen

`head, tail, last, init, take, drop`

`map`
`filter`
`foldl`
`zip`
`zipWith`

```
map odd [1..20]
filter odd [1..20]
```

Operatoren auf Listen

`head, tail, last, init, take, drop`

`map`
`filter`
`foldl`
`zip`
`zipWith`

```
map odd [1..20]
filter odd [1..20]
foldl (+) 0 [1..10]
```

Operatoren auf Listen

`head, tail, last, init, take, drop`

`map`
`filter`
`foldl`
`zip`
`zipWith`

```
map odd [1..20]
filter odd [1..20]
foldl (+) 0 [1..10]
zip [1..4] ['a'..'d']
```

Operatoren auf Listen

head, tail, last, init, take, drop

map	`map odd [1..20]`
filter	`filter odd [1..20]`
foldl	`foldl (+) 0 [1..10]`
zip	`zip [1..4] ['a'..'d']`
zipWith	`zipWith (*) [1..4] [5..8]`

Listen: Aufgaben

Schreibe eine Funktion...

- zur Berechnung der Länge einer Liste
- welche die Liste aller gerade Zahlen erzeugt
- die prüft, ob ein gesuchtes Element in einer Liste enthalten ist
- die ein Element in eine sortierte Liste einfügt
- die zwei sortierte Listen zu einer zusammenfasst (merge)
- die eine Liste aller Quadratzahlen erzeugt
- die merge-sort implementiert
- die insertion-sort implementiert
- die eine Liste umdreht

Fibonacci und `zip`

Fibonacci und `zip`

```
fib = 1:zipWith (+) fib (tail fib)
```

Primzahlen

```
odds = filter odd [1..]
oddPrimes (p : ps) = p : (oddPrimes [q | q <- ps, q `mod` p /= 0])
primes = 2 : oddPrimes (tail odds)
```

lambda: λ

```
langweilig = \x -> x
linF m c = \x -> m*x+c
```

Lambdas sind anonyme Funktionen, sie haben keinen Namen.

lambda: λ

```
langweilig = \x -> x
linF m c = \x -> m*x+c
```

Lambdas sind anonyme Funktionen, sie haben keinen Namen.

```
filter (\x -> (mod x 3 == 0 || mod x 5 == 0)) [1..10]
```

lambda: λ

```
langweilig = \x -> x
linF m c = \x -> m*x+c
```

Lambdas sind anonyme Funktionen, sie haben keinen Namen.

```
filter (\x -> (mod x 3 == 0 || mod x 5 == 0)) [1..10]
```

Parameter stehen vor dem Pfeil, Vorschrift dahinter

Endrekursive Funktionen

```
pow a 0 = 1
pow a b = a * pow a (b-1)
```

Endrekursive Funktionen

```
pow a 0 = 1
pow a b = a * pow a (b-1)

xpow a b = powAcc a b 1
powAcc a b acc = a (b-1) (acc*a)
```

Typen in Haskell

```
type Polynom = [Double]
```

mit $f(x) = a_0 \cdot x^0 + a_1 \cdot x^1$ und den Koeffizienten a_i im Datentyp gespeichert.

Typen in Haskell

```
type Polynom = [Double]
```

mit $f(x) = a_0 \cdot x^0 + a_1 \cdot x^1$ und den Koeffizienten a_i im Datentyp gespeichert.

- Schreibe eine Funktion `add`, die zwei Polynome addiert
- Nutze das Hornerschema, um ein Polynom auszuwerten

Curry-ing

```
incr = 1 +
```

Curry-ing

```
incr = 1 +
square = flip (^) 2
```

“Dinge” im Lambda-Kalkül

```
TRUE = \x y -> x
FALSE = \x y -> y
IF_ELSE = \b d e -> b d e
```

“Dinge” im Lambda-Kalkül

```
TRUE = \x y -> x
FALSE = \x y -> y
IF_ELSE = \b d e -> b d e
```

Zum Testen: `IF_ELSE TRUE 0 1`

“Dinge” im Lambda-Kalkül

```
TRUE = \x y -> x
FALSE = \x y -> y
IF_ELSE = \b d e -> b d e
```

Zum Testen: `IF_ELSE TRUE 0 1`

Das Lambda-Kalkül ist eine Alternative zur Turing-Maschine.

Church-Zahlen

```
c0 = \s z -> z
c1 = \s z -> s z
c2 = \s z -> s (s z)
c3 = \s z -> s (s (s z))
```

Church-Zahlen

```
c0 = \s z -> z
c1 = \s z -> s z
c2 = \s z -> s (s z)
c3 = \s z -> s (s (s z))
```

Die Zahlen drücken aus, wie oft eine Funktion `s` angewandt wird.

Umrechnen: `c3 (1+) 0`

Church-Zahlen

```
c0 = \s z -> z
c1 = \s z -> s z
c2 = \s z -> s (s z)
c3 = \s z -> s (s (s z))
```

Die Zahlen drücken aus, wie oft eine Funktion `s` angewandt wird.

Umrechnen: `c3 (1+) 0`

Nachfolger:

```
succ = \n s z -> s (n s z)
c4 = succ c3
```

Lambda-Kalkül: Aufgaben

Nutze nur Lambdas und selbst definierte Ausdrücke

- definiere **`and`** – Tipp: if-else
- definiere **`or`**
- definiere `isZero`
- definiere `add`
- definiere `times`
- definiere **`exp`**

Lambda-Kalkül: Lösungen

```
and = \a b -> a b FALSE
or = \a b -> a TRUE b
isZero = \n -> n (\x -> FALSE) TRUE
add = \n m s z -> m s (n s z)
times = \n m s z -> n (m s) z
exp = \n m s z -> n m s z
```