

# Funktionale Programmierung Mitschrieb

Finn Ickler

November 26, 2015

„Avoid success at all cost “

---

Simon Peyton Jones

## Contents

<b>Vorlesung 1</b>	<b>2</b>
Functional Programming (FP) . . . . .	3
Computational Model in FP : <i>Reduction</i> . . . . .	3
Haskell Ramp Up . . . . .	4
<b>Vorlesung 2</b>	<b>4</b>
Values and Types . . . . .	5
Types . . . . .	5
Type Constructors . . . . .	6
Currying . . . . .	7
<b>Vorlesung 3</b>	<b>8</b>
Defining Values (and thus: Functions) . . . . .	8
Lokale Definitionen . . . . .	9
Lists([a]) . . . . .	11
Type Synonyms . . . . .	13
<b>Vorlesung 4</b>	<b>13</b>
Pattern Matching . . . . .	13
Pattern matching in expressions (case) . . . . .	14
<b>Vorlesung 5</b>	<b>17</b>
Algebraic Data Types (Sum of Product Types) . . . . .	17

<b>Vorlesung 6</b>	<b>22</b>
Type Classes . . . . .	22
Class Constraints . . . . .	22
Class inheritance . . . . .	22
Class Instances . . . . .	25
Deriving Class Instances . . . . .	26
<b>Vorlesung 7</b>	<b>29</b>
Domain Specific Languages . . . . .	29
Modules . . . . .	29

## List of Listings

1	Hello World . . . . .	2
2	isPrime in C . . . . .	3
3	isPrime in Haskell . . . . .	4
4	Lazy Evaluation in der ghci REPL . . . . .	4
5	Verschiedene Schreibweise einer Applikation . . . . .	5
6	Eigener $\approx$ Operator . . . . .	5
7	fac in Haskell . . . . .	9
8	Power in Haskell . . . . .	9
9	sum in Haskell . . . . .	14
10	ageOf in Haskell . . . . .	15
11	take in Haskell . . . . .	15
12	merge in Haskell . . . . .	16
13	mergeSort in Haskell . . . . .	16
14	weekday.hs . . . . .	17
15	RockPaperScissors.hs . . . . .	18
16	sequence.hs . . . . .	19
17	cons.hs . . . . .	20
18	eval-compile-run.hs . . . . .	21
19	Default implementation of Show, Ord and Enum . . . . .	23
20	Rock paper Scissors with instances . . . . .	28
21	library-exposed.hs . . . . .	30

## Vorlesung 1

```
-- Hello World Haskell
main :: IO ()
main = putStrLn "Chewie, we're home"
```

Code example 1: Hello World

## Functional Programming (FP)

A programming language is a medium for expressive ideas (not to get a computer to perform operations ). Thus programs must be written for people to read, and only incidentally for machines.

### Computational Model in FP : *Reduction*

Replace expressions by their value.

IN FP, expressions are formed by applying functions to values.

1. Function as in maths:  $x = y \rightarrow f(x) = f(y)$
2. Functions are values like numbers or text

	FP	Imperative
construction	function application and composition	statement sequencing
execution	reduction (expression evaluation)	state changes
semantics	$\lambda$ -calculus	denotational

$n \in \mathbb{N}, n \geq 2$  is a prime number  $\Leftrightarrow$  the set of non-trivial factors of  $n$  is empty.

$n$  is prime  $\Leftrightarrow \{m \mid m \in \{2, \dots, n-1\}, n \bmod m = 0\} = \{\}$

```
int IsPrime(int n)
{
    int m;
    int found_factor;
    found_factor
    for (m = 2; m <= n - 1; m++)
    {
        if (n % m == 0)
        {
            found_factor = 1 ;
            break;
        }
    }
    return !found_factor;
}
```

Code example 2: isPrime in C

```
isPrime :: Integer -> Bool
isPrime n = factors n == []
  where
    factors :: Integer -> [Integer]
    factors n = [ m | m <- [2..n-1], mod n m == 0]

main :: IO ()
main = do
  let n = 42
  print (isPrime n)
```

Code example 3: isPrime in Haskell

```
let xs = [ x+1 | x <- [0..9] ]
:sprint xs = _
length xs
:sprint xs = [_,_,_,_,_,_,_,_,_,_]
```

Code example 4: Lazy Evaluation in der ghci REPL

## Haskell Ramp Up

Read  $\equiv$  as "denotes the same value as"

Apply f to value e:  $f \sqcup e$

(juxtaposition, "apply", binary operator  $\sqcup$ , Haskell speak:  $\text{infixL } 10 \sqcup$ ) =  $\sqcup$  has

max precedence (10):  $f \ e_1 + e_2 \equiv (f \ e_1) + e_2$   $\sqcup$  associates to the left  $g \sqcup f \sqcup e \equiv (g \sqcup f) \ e$  Function composition:

- $g \ (f \ e)$
- Operator "." ("after") :  $(g.f) \ e \ (. = \circ) = g(f \ (e))$
- Alternative "apply" operator \$ (lowest precedence, associates to the right),  
infix 0\$):  $f \$ e_1 + e_2 = f \ (e_1 + e_2)$

## Vorlesung 2

Prefix application of binary infix operator  $\oplus$

$(\oplus) \ e_1 \ e_2 \equiv e_1 \oplus e_2$

$(\&\&) \ \text{True} \ \text{False} \equiv \text{False}$

Infix application of binary function f:

$e_1 \ `f` \ e_2 \equiv f \ e_1 \ e_2$

$x \ `elem` \ xs \equiv x \in xs$

User defined operators with characters : !#%&\*+ / <=> ? @ \ ^ |

```
cos 2 * pi
cos (2 * pi)
cos $ 2 * pi
isLetter (head (reverse ("It's a " ++ "Trap")))
(isLetter . head . reverse) ("It's a " ++ "Trap")
isLetter $ head $ reverse $ "It's a " ++ "Trap"
```

Code example 5: Verschiedene Schreibweise einer Applikation

```
epsilon :: Double
epsilon = 0.00001
(==~) :: Double -> Double -> Bool
x ==~ y = abs (x - y) < epsilon
infix 4 ==~
```

Code example 6: Eigener  $\approx$  Operator

## Values and Types

Read `::` as "has type"

Any Haskell value `e` has a type `t` (`e :: t`) that is determined at compile time.

The `::` type assignment is either given explicitly or inferred by the computer

## Types

Type	Description	Value
Int	fixed precision integers ( $-2^{63} \dots 2^{63} - 1$ )	0, 1, 42
Integer	arbitrary Precision integers	0, $10^{100}$
Float, Double	Single/Double precision floating points	0.1, 1e03
Char	Unicode Character	'x', '\t', ' ', '\8710'
Bool	Booleans	True, False
()	Unit (single-value type)	()

```
2
it :: Integer
42 :: Int
it :: Int
'a'
it :: Char
True
it :: Bool
10^100
it :: Integer
10^100 :: Double
it :: Double
```

## Type Constructors

- Build new types from existing Types
- Let a,b denote arbitrary Types (type variables)

Type Constructor	Description	Values
(a,b)	pairs of values of types a and b	(1,True) :: (Int, Bool)
(a <sub>1</sub> , a <sub>2</sub> , ..., a <sub>n</sub> )	n-Types	2,False :: (Int, Bool)
[a]	list of values of type a	[] :: [a]
Maybe a	optional value of type a	Just 42 :: Maybe Integer Nothing :: Maybe a
Either a b	Choice between values of Type a and b	Left 'x' :: Either Char b Right pi :: Either a Double
IO a	I/O action that returns a value of type a (can have side effects)	print 42 :: IO ()
a -> b	function from type a to b	getChar :: IO Char isLetter :: Char -> Bool

```
(1, '1', 1.0)
it :: (Integer, Char, Double)
[1, '1', 1.0]
it :: Fehler
[0.1, 1.0, 0.01]
it :: [Double]
[]
it :: [t]
"Yoda"
it :: [Char]
['Y', 'o', 'd', 'a']
"Yoda"
[Just 0, Nothing, Just 2]
it :: [Maybe Integer]
[Left True, Right 'a']
it :: [Either Bool Char]
print 'x'
it :: ()
getChar
*
it :: Char
:t getChar
getChar :: IO Char
:t fst
fst :: (a,b) -> a
:t snd
snd :: (a,b) -> b
:t head
head :: [a] -> a
:t (++)
(++): :: [a] -> [a] -> [a]
```

## Currying

- Recall:
  - $e_1 ++ e_2 \equiv (++) e_1 e_2$
  - $++ e_1 e_2 \equiv ((++) e_1) e_2$
- Function application happens one argument at a time (currying, Haskell B. Curry)
- Type of n-ary function:  $: a_1 -> a_2 \dots -> a_n -> b$
- Type constructor  $->$  associates to the right thus read the type as:  
 $a_1 -> (a_2 -> a_3 (\dots -> (a_n -> b) \dots))$

- Enables partial application: "Give me a value of type  $a_1$ , I'll give you a (n-1)-ary function of type  $a_2 \rightarrow a_3 \rightarrow \dots \rightarrow a_n \rightarrow b$ "

```
"Chew" ++ "bacca"
"Chewbacca"
(++) "Chew" "bacca"
"Chewbacca"
((++) "Chew") "bacca"
"Chewbacca"
:t (++) "Chew"
"Chew" :: [Char] -> [Char]
let chew = (++) "Chew"
chew "bacca"
"Chewbacca"
let double (*) 2
double 21
42
```

## Vorlesung 3

### Defining Values (and thus: Functions)

- `=` binds names to values, names must not start with A-Z (Haskell style: camelCase)
- Define constant (0-ary) `c`, value of `c` is that of expression:  
`c = e`
- Define n-ary function, arguments  $x_i$  and `f` may occur in `e` (no "letrec" needed)  
`f x1 x2 ... xn = e`
- Hskell programm = set of top-level bindings (order immaterial, no rebinding)
- Good style: give type assignment for top-level bindings:  
`f :: a1 -> a2 -> b`  
`f x1 x2 = e`
- Guards (introduced by `|`).

```
f x1 x2 ... xn
  | q1 = e1
  | q2 = en
```

- $q_i$  (expressions of type `Bool`) evaluated top to bottom, first `True` guards "wins"
- $$\text{fac } n = \begin{cases} 1 & \text{if } n \geq 1 \\ n \cdot \text{fac}(n-1) & \text{else} \end{cases}$$



```
fac :: Integer -> Integer
fac n = if n <= 1 then 1 else n * fac (n - 1)

fac2 n | n <= 1    = 1
      | otherwise = n * fac2 (n - 1)

main :: IO ()
main = print $ fac 10
```

Code example 7: fac in Haskell

```
power :: Double -> Integer -> Double
power x k | k == 1 = x
          | even k = power (x * x) (halve k)
          | otherwise = x * power (x * x) (halve k)

where
  even :: Integer -> Bool -- Nicht typisch
  even n = n `mod` 2 == 0
  halve n = n `div` 2

main :: IO ()
main = print $ power 2 16
```

Code example 8: Power in Haskell

## Lokale Definitionen

1. **where** - binding : Local definitions visible in the entire right-hand-side (rhs) of a definition

```
f x1 x2 ... xn
  | q1 = e1
  | q2 = en
  where
    g1 ... = b1
    gi ... = bi
```

2. **let** - expression Local definitions visible inside an expression:

```
let g1 ... = b1
    g2 ... = b1
in e
```

## Haskells 2-dimensionale Syntax (Layout) (Forum-beitrag)

Hallo zusammen,

in der dritten Vorlesung hatte ich erwähnt, dass Haskell's Syntax darauf verzichtet, Blöcke (von Definitionen) mittels Sonderzeichen abzugrenzen und zu strukturieren. Andere Programmiersprachen bedienen sich hier typischerweise Zeichen wie `,` und `;`.

Haskell baut hingegen auf das sog. Layout, eine Art 2-dimensionaler Syntax. Wer schon einmal Python und seine Konventionen zur Einrückung von Blöcken hinter `for` und `if` kennengelernt hat, wird hier Parallelen sehen. Die Regelungen zu Layout lauten wie folgt und werden vom Haskell-Compiler während der Parsing-Phase angewandt:

- The first token **after** a `where/let` and the **first token of a top-level definition** define the upper-left corner of a box.
- The first token left of the box closes the box (offside rule).
- Insert a `{` before the box.
- Insert a `}` after the box.
- Insert a `;` before each line that starts at left box border.

Die Anwendung dieser Regeln auf dieses Beispielprogramm:

```
let y    = a * b
    f x  = (x + y) / y
in f c + f d
```

führt zur Identifikation der folgenden Box:

```
let { y    = a * b
     f x  = (x + y) / y
```

```
in f c + f d
```

Das Token `in` in der letzten Zeile steht links von der Boxgrenze im Abseits (siehe die offside rule). Der Parser führt nun die Zeichen `,` und `;` ein und verarbeitet das Programm so, als ob der Programmierer diese Zeichen explizit angegeben hätte. (Haskell kann alternativ übrigens auch in dieser sog. expliziten Syntax geschrieben werden — das ist aber sehr unüblich, hat negativen Einfluss aufs Karma und ist vor allem für den Einsatz in automatischen Programmgeneratoren gedacht.)

Die explizite Form des obigen Programmes lautet (nach den drei letzten Regeln):

```
let {y    = a * b
    ;f x = (x + y) / y}
in  f c + f d
```

Damit ist die Bedeutung des Programmes eindeutig und es ist klar, dass bspw. nicht das folgende gemeint war (in dieser alternativen Lesart ist das Token `f` aus der zweiten in die erste Zeile "gerutscht"):

```
let y = a * b f
    x = (x + y) / y
in  f c + f d
```

Aus diesen Layout-Regeln ergeben sich recht einfache Richtlinien für das Einrücken in Haskell-Programmen:

- Die Zeilen einer Definition auf dem Top-Level beginnen jeweils ganz links (Spalte 1) im Quelltext.
- Lokale `where` / `let`-Definitionen werden um mindestens ein Whitespace (typisch: 2 oder 4 Spaces oder 1 Tab) eingerückt.
- Es gibt in Haskell ein weiteres Keyword (`do`, wird später thematisiert), das den gleichen Regeln wie `where` / `let` folgt.

Beste Grüße,  
—Torsten Grust

## Lists([a])

- Recursive definition:
  1. `[]` ist a list (nil), type `[] :: [a]`
  2. `x : xs` (head, tail) is a list, if `x :: a`, and `xs :: [a]`.  
`cons :: (a -> [a] -> [a])` with `infixr : 5`
- Notation: `3:(2:1:[]) ≡ 3:2:1:[] ≡ [3,2,1]`

```
[]
it :: [t]
[1]
it :: [Integer]
[1,2,3]
it :: [Integer]
['z']
"z"
it :: [Char]
['z','x']
"zx"
it :: [Char]
[] == ""
True
it :: Bool
[[1],[2,3]]
it :: [[Integer]]
[[1],[2,3],[]]
[[1],[2,3]]
it :: [[Integer]]
False:[]
[False]
it :: [Bool]
(False:[]):[]
it :: [[Bool]]
:t [(<),(==),(>)]
[(<),(==),(>)] :: Ord a => [a -> Bool]
[(1,"one"),(2,"two"),(3,"three")]
it :: [(Integer,[Char])]
:t head
head :: [a] -> a
:t tail :: [a] -> [a]
head "It's a trap"
'I'
it :: Char
tail "It's a trap"
"t's a trap"
it :: [Char]
reverse "Never odd or even"
"neve ro ddo reveN"
it :: [Char]
```

- Law  $\forall xs \neq []: \text{head } xs : \text{tail } xs$

```
:i String
type String = [Char]
```

## Type Synonyms

- Introduce your own type synonyms. (type names : *Uppercase*) `type t1 = t2`

```
type Bits = [Integer]
```

```
type Predicate a = a -> Bool
```

```
bits :: Integer -> Bits
```

```
bits n | n == 0      = [0]
       | otherwise = (n `mod` 2) : bits (n `div` 2)
```

```
isEven :: Predicate Integer
```

```
isEven n = head (bits n) == 0
```

```
main :: IO ()
```

```
main = print $ isEven 35
```

Sequence (lists of enumerable elements)

- $[x..y] \equiv [x, x+1, x+2, \dots, y]$

```
['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

- $x, s..y \equiv [x, x+i, x+(2*i), \dots, y]$  where  $i = x-s$

```
[1,3..20]
[1,3,5,7,9,11,13,15,17,19]
[2,4..20]
[2,4,6,8,10,12,14,16,18,20]
```

- Infinite List `[1..]`

## Vorlesung 4

### Pattern Matching

The idiomatic way to define functions by cases: `f :: a1 -> ... -> ak -> b`

```
f p11 ... p1k = e1
```

```
⋮ ⋮ ⋮ ⋮
```

```
f pm1 ... pnk = en
```

For all  $e_i :: b$  on  $a_i$  call `f x1x2...xk` each  $x_i$  is matched against patterns  $p_{i1} \dots p_{in}$  in order. Result is  $e_r$  if the  $r$ th branch is the first in which all patterns match.

Pattern	Matches if...	Bindings in $e_r$
constant $c$	$x_1 == c$	
variable $v$	always	$v = x_i$
wildcard $\_$	always	
tuple $(p_1, \dots, p_n)$	components of $x_i$ match type component patterns	Those bound by the com- ponent patterns
$[]$	$x_i == []$	
$p_1 : p_2$	<b>head</b> $x_1$ matches $p_1$ , <b>tail</b> $x_i$ matches $p_2$	
$v@p$	$p$ matches	those bound by $p$ and $v = x_i$

Note: In a pattern, a variable may only occur once (linear patterns only)

```
--(1) if then else
sum' :: [Integer] -> Integer
sum' xs =
    if xs == [] then 0 else head xs + sum' (tail xs)
--(2) guards
sum'' :: [Integer] -> Integer
sum'' xs | xs == [] = 0
        | otherwise = head xs + sum'' (tail xs)
--(3) pattern matching
sum''' :: [Integer] -> Integer
sum''' [] = 0
sum''' (x:xs) = x + sum''' xs

main :: IO ()
main = do
    print $ sum' [1,2,3]
    print $ sum'' [1,2,3]
    print $ sum''' [1,2,3]
```

Code example 9: sum in Haskell

## Pattern matching in expressions (case)

```
case e of p1 | q11 -> e11
        :
        pn | qn1 -> en1
```

```
type Dictionary a b = [(a,b)]
type Person = String
type Age = Integer

people :: Dictionary Person Age
people = [("Darth", 46), ("Chewie",200), ("Yoda", 902)]

ageOf :: Dictionary Person Age -> Person -> Maybe Age
-- The old way
--ageOf pas p | fst (head pas) == p = snd (head pas)
--              | otherwise          = ageOf (tail pas) p
ageOf []      p'      = Nothing
ageOf ((p,a):pas) p' | p == p' = Just a
                    | otherwise = ageOf pas p'

main :: IO ()
main = do
    print $ ageOf people "Luke"
```

Code example 10: ageOf in Haskell

```
take' :: Integer -> [a] -> [a]
take' 0 _ = []
take' _ [] = []
take' n (x:xs) = x:take' (n-1) xs

main :: IO ()
main = print $ take' 20 [1,3..]
```

Code example 11: take in Haskell

```
-- Merge two sorted lists respecting their orderings
--
-- merge (<) [0,3,5] [1,2,4] = [0,1,2,3,4,5]

merge :: (a -> a -> Bool) -> [a] -> [a] -> [a]
merge _ [] ys = ys
merge _ xs [] = xs
merge (<<<) l1@(x: xs) l2@(y:ys) | x <<< y = x:merge (<<<) xs l2
                                | otherwise = y:merge (<<<) l1 ys

main :: IO ()
main = print $ merge (<) [1,3..19] [2,4..20]
```

Code example 12: merge in Haskell

```
--Sortes a list

mergeSort :: (a -> a -> Bool) -> [a] -> [a]
mergeSort _ [] = []
mergeSort _ [x] = [x]
mergeSort (<<<) xs = merge (<<<) (mergeSort (<<<) ls)
                           (mergeSort (<<<) rs)
  where
    (ls,rs) = splitAt (length xs `div` 2) xs
    merge :: (a -> a -> Bool) -> [a] -> [a] -> [a]
    merge _ [] ys = ys
    merge _ xs [] = xs
    merge (<<<) l1@(x: xs) l2@(y:ys)
      | x <<< y = x:merge (<<<) xs l2
      | otherwise = y:merge (<<<) l1 ys

main :: IO ()
main = print $ mergeSort (<) [1..100]
```

Code example 13: mergeSort in Haskell



## Vorlesung 5

### Algebraic Data Types (Sum of Product Types)

- Recall: `[]` and `(:)` are the *constructors* for Type `[a]`
- Can define entirely new Type `T` and its constructors  $K_i$ :

```
data T a1 a2 ... an = K1 b11 ... b1n1
                      | K2 b21 ... b2n2
                      | ...
                      | Kr br1 ... brnr
```

- Defines *Type constructor* `T` and *r value constructor* with types
- $K_i :: b_{i1} \dots b_{ini} \rightarrow T a_1 a_2 \dots a_n$
- $K_i$  identifier with uppercase first letter or symbol starting with `:`
- Example: `[weekday.hs]`
  - Sum (or enumeration, choice)

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Eq, Show, Ord, Enum, Bounded)
weekend :: Weekday -> Bool
weekend Sat = True
weekend Sun = True
weekend _   = False

main :: IO ()
main = do
  print $ weekend Mon
  print $ [Mon..Fri]
```

Code example 14: weekday.hs

```
Wed
No instance for (Show Weekday) arising from a use of print
Thu == Sun
No instance for (Eq Weekday) arising from a use of '=='
Mon > Sat
No instance for (Ord Weekday) arising form a use of '>'
```

- Add deriving (C, C, . . . , C) to data declaration to define canonical (intuitive) operations:

c (class)	operations
<b>Eq</b>	equality (==, /=)
<b>Show</b>	printing ( <b>show</b> )
<b>Ord</b>	ordering (<, <=, max)
<b>Enum</b>	enumeration ([x..y])
<b>Bounded</b>	bounds ( <b>minBound</b> , <b>maxBound</b> )

```
data Move = Rock | Paper | Scissor
  deriving (Eq)
```

```
data Outcome = Lose | Tie | Win
  deriving (Show)
```

```
outcome :: Move -> Move -> Outcome
outcome Rock Scissor = Win
outcome Paper Rock   = Win
outcome Scissor Paper = Win
outcome us      them
  | us == them = Tie
  | otherwise  = Lose
```

```
main :: IO ()
main = do
  print $ outcome Paper Scissor
```

Code example 15: RockPaperScissors.hs

- Product,  $r = 1$ ,  $n_1 = 2$  ()
- Sum of Products:
 

```
Maybe a = Nothing | Just a
data Either a b = Left a | Right a
List a = Nil
        | Cons a (List a)
```

```
data Sequence a = S Int [a]
    deriving (Eq, Show)

fromList :: [a] -> Sequence a
fromList xs = S (length xs) xs

(+++) :: Sequence a -> Sequence a -> Sequence a
S lx xs +++ S ly ys = S (lx + ly) (xs ++ ys)

len :: Sequence a -> Int
len (S lx _) = lx

main :: IO ()
main = do
    print $ fromList [0..9]
    print $ len (fromList ['a'..'z'])
```

Code example 16: sequence.hs

```
data List a = Nil
            | Cons a (List a)
            deriving (Show)

toList :: [a] -> List a
toList [] = Nil
toList (x:xs) = Cons x (toList xs)

fromList :: List a -> [a]
fromList Nil = []
fromList (Cons x xs) = x:fromList xs

mapList :: (a -> b) -> List a -> List b
mapList f Nil = Nil
mapList f (Cons x xs) = Cons (f x) (mapList f xs)

liftList f = toList . f . fromList

mapList' :: (a -> b) -> List a -> List b
mapList' f xs = liftList (map f) xs

filterList :: (a -> Bool) -> List a -> List a
filterList _ Nil = Nil
filterList p (Cons x xs) | p x = Cons x (filterList p xs)
                        | otherwise = filterList p xs

filterList' :: (a -> Bool) -> List a -> List a
filterList' p xs = liftList (filter p) xs

main :: IO()
main = do
    print $ mapList (+1) $ toList [1..5]
    print $ fromList $ filterList (> 3) $ mapList (+1) $ toList [1..5]
```

Code example 17: cons.hs

```
data Exp a = Lit a
           | Add (Exp a) (Exp a)
           | Sub (Exp a) (Exp a)
           | Mul (Exp a) (Exp a)
  deriving (Show)

ex1 :: Exp Integer
ex1 = Add (Mul (Lit 5) (Lit 8)) (Lit 2)

evaluate :: Num a => Exp a -> a
evaluate (Lit n)      = n
evaluate (Add e1 e2) = evaluate e1 + evaluate e2
evaluate (Mul e1 e2) = evaluate e1 * evaluate e2
evaluate (Sub e1 e2) = evaluate e1 - evaluate e2

main :: IO()
main = do
  print $ ex1
  print $ evaluate ex1
```

Code example 18: eval-compile-run.hs

## Vorlesung 6

### Type Classes

A Type class  $C$  defines a family of type signatures ("methods") which all *instances* of  $C$  must implement:

```
class C where
  f1 :: t1
  f2 :: t2
  ⋮
  fn :: tn
```

The  $t_i$  *must* mention  $C$ . For any  $f_i$ , the class may provide default definitions (that instances may overwrite).

- Example

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

### Class Constraints

A *class constraint*  $e$  ( $a \Rightarrow :: t$  (where  $t$  mentions  $a$ )) says that  $e$  has type  $t$  *only if*  $a$  is an instance of class  $C$ .

```
:t (+)
(+) :: Num a => a -> a -> a
:t print
print :: Show a => a -> IO ()
:hoogle +Data.List
Data.List sort :: Ord a => [a] -> [a]
:hoogle [(a,b)] -> a -> Maybe b
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```

### Class inheritance

Defining class  $(c_1 a, c_2 a, \dots) \Rightarrow (a \text{ where } \dots)$  makes type class  $C$  a *subclass* of the  $c_i$ .  $C$  inherits all methods of the  $c_i$ .

( $a \Rightarrow t$  implies  $(c_1 a, c_2 a, \dots, C a) \Rightarrow t$ )

```

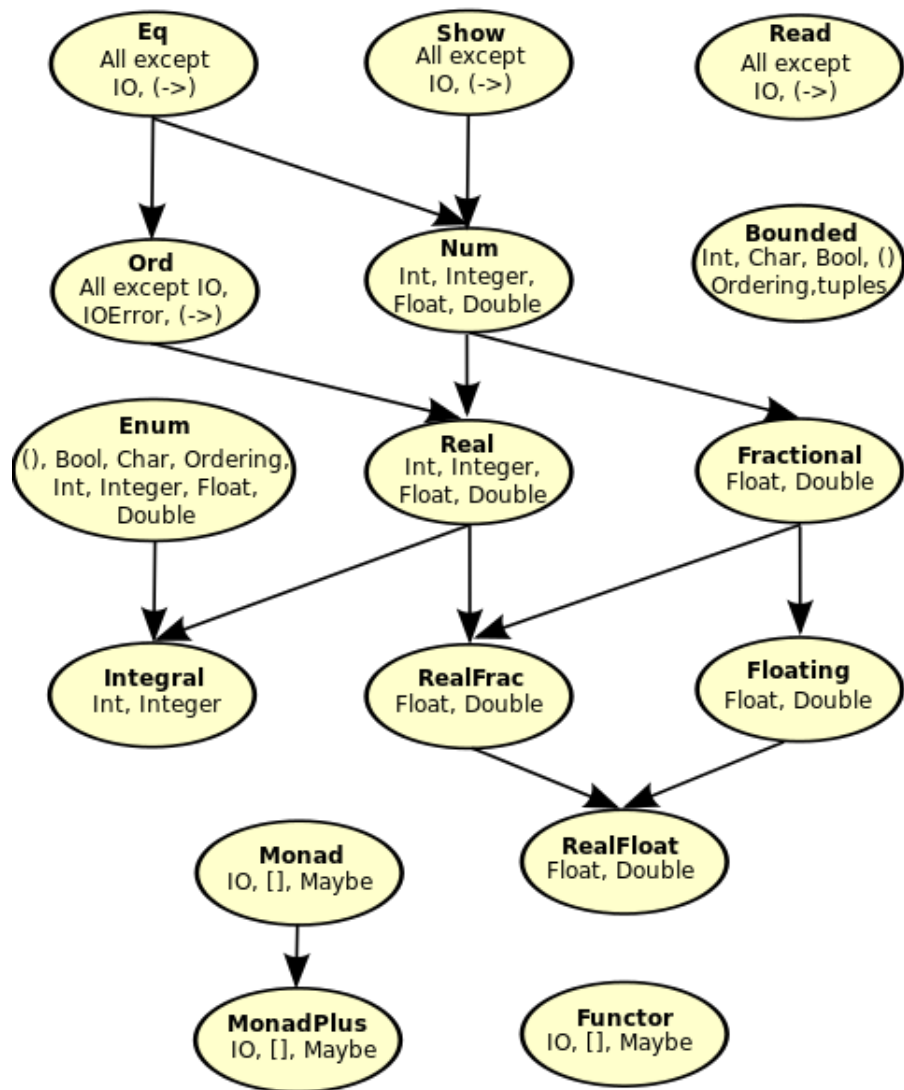
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
  --Minimal complete Definition enumfrom and toEnum
  succ = toEnum . (+1) . fromEnum
  pred = toEnum . (subtract 1) . fromEnum
  enumFrom x = map toEnum [fromEnum x ..]
  enumFromTo x y = map toEnum [fromEnum x .. fromEnum y]
  enumFromThenTo x y z = map toEnum [fromEnum x, fromEnum y .. fromEnum z]

class (Eq a) => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
  -- Minimal complete Definition compare
  compare x y | x == y    = EQ
               | x <= y    = LT
               | otherwise = GT
  x <= y = compare x y /= GT
  x < y  = compare x y == LT
  x >= y = compare x y /= LT
  x > y  = compare x y == GT

class Show a where
  showsPre :: Int -> a -> ShowS
  show     :: a -> String
  showList :: [a] -> ShowS
  --Minimal complete definition: show or showsPrec
  showsPrec _ x s = show x ++ s
  show x          = showsPrec 0 x ""

```

Code example 19: Default implementation of Show, Ord and Enum





## Class Instances

If type  $t$  implements the method of class  $C$ ,  $t$  becomes an *instance* of  $c$ :

```
instance C t where
  f1 = <def of f1> --all f may be
      :             --provided, minimal
  fn = <def of fn> --complete definition
      --must be provided
```

- Example:

```
instance Eq Bool where
  x == y = (x && y) || (not x && not y)

instance (Eq a, Eq b) => Eq (a,b) where
  (x1,y1) == (x2,y2) = x1 == x2 && y1 == y2

instance Eq a => Eq [a] where
  [] == [] = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _ == _ = False
```

- An instance definition for type constructor  $t$  may formulate type constraints for its argument types:  $a, b \dots$  :

```
instance (c1a, c2, c3b, ...) => (t a b) where
```

```
:i Enum
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
    -- Defined in 'GHC.Enum'
instance Enum Word -- Defined in 'GHC.Enum'
instance Enum Ordering -- Defined in 'GHC.Enum'
instance Enum Integer -- Defined in 'GHC.Enum'
instance Enum Int -- Defined in 'GHC.Enum'
instance Enum Char -- Defined in 'GHC.Enum'
instance Enum Bool -- Defined in 'GHC.Enum'
instance Enum () -- Defined in 'GHC.Enum'
instance Enum Float -- Defined in 'GHC.Float'
instance Enum Double -- Defined in 'GHC.Float'
fromEnum 'A'
65
fromEnum 'B'
66
toEnum 65
Exception: Prelude.Enum.().toEnum: bad argument
:t toEnum 65
toEnum 65 :: Enum a => a
toEnum 65 :: Char
'A'
toEnum 0 :: Bool
False
toEnum 20 :: Double
20.0
```

## Deriving Class Instances

- Automatically made user-defined data (data ...) instances of classes  $c_i \in \{\text{Eq}, \text{Ord}, \text{Enum}, \text{Bounded}, \text{Show}, \text{Read}\}$

```
data T a1 a2 ... an = ...
    |
    deriving (c1 .. , cn)
```

```
import Data.Maybe
import Data.Tuple
data Outcome = Lose | Tie | Win
```

```
deriving (Eq, Ord, Enum, Bounded, Show)

data Move = Rock | Paper | Scissor
  deriving (Eq)
instance Ord Move where
  Rock <= Rock      = True
  Rock <= Paper     = True
  Paper <= Paper    = True
  Paper <= Scissor  = True
  Scissor <= Scissor = True
  Scissor <= Rock   = True
  _ <= _            = False
instance Show Move where
  show Scissor = ""
  show Rock    = ""
  show Paper   = ""

table :: [(Move, Int)]
table = [(Rock, 0), (Paper, 1), (Scissor, 2)]
instance Enum Move where
  fromEnum o = fromJust $ lookup o table
  toEnum n   = fromJust $ lookup n $ map swap table

outcome :: Move -> Move -> Outcome
outcome Rock    Scissor = Win
outcome Paper   Rock    = Win
outcome Scissor Paper   = Win
outcome us      them
  | us == them = Tie
  | otherwise  = Lose

main :: IO ()
main = do
  print $ outcome Paper Scissor
```

```

import Data.Maybe
import Data.Tuple
data Outcome = Lose | Tie | Win

instance Eq Outcome where
    Lose == Lose = True
    Tie == Tie = True
    Win == Win = True
    _ == _ = False
instance Enum Outcome where
    fromEnum Lose = 0
    fromEnum Tie = 1
    fromEnum Win = 2
    toEnum 0 = Lose
    toEnum 1 = Tie
    toEnum 2 = Win
instance Show Outcome where
    show Lose = "Lose"
    show Tie = "Tie"
    show Win = "Win"

instance Ord Outcome where
    Lose <= Lose = True
    Lose <= Tie = True
    Lose <= Win = True
    Tie <= Tie = True
    Tie <= Win = True
    Win <= Win = True
    _ <= _ = False

data Move = Rock | Paper | Scissor
instance Eq Move where
    Rock == Rock = True
    Paper == Paper = True
    Scissor == Scissor = True
    _ == _ = False

table :: [(Move,Int)]
table = [(Rock, 0), (Paper, 1), (Scissor, 2)]
instance Enum Move where
    fromEnum o = fromJust $ lookup o table
    toEnum n = fromJust $ lookup n $ map swap table

outcome :: Move -> Move -> Outcome
outcome Rock Scissor = Win
outcome Paper Rock = Win
outcome Scissor Paper = Win
outcome us them
    | us == them = Tie
    | otherwise = Lose

main :: IO ()
main = do
    print $ outcome Paper Scissor

```

## Vorlesung 7

### Domain Specific Languages

- "small languages" designed to easily and directly express the concepts/idioms of a given domain. *Not* Turing-complete in general.

	Domain	DSL
• Examples:	Os automation	Shell scripts
	Typesetting	T <sub>E</sub> X, L <sub>A</sub> T <sub>E</sub> X
	Queries	SQL
	Game Scripting	UnrealScript, Lua
	Parsing	Bison, ANTLR

- Functional Languages are good hosts for Embedded DSLs:
  - algebraic data types (e.g model abstract syntax trees)
  - higher-order functions (e.g control constructs)
  - lightweight syntax (layout/whitespace, non-alphabetic identifiers)

Example: An embedded DSL for finite sets of integers:

```
type IntegerSet = ...
empty :: IntegerSet
insert :: Integer -> IntegerSet -> IntegerSet
delete :: Integer -> IntegerSet -> IntegerSet
member :: Integer -> IntegerSet -> Bool
member 3 (insert 1 (delete 3 (insert 2 (insert 3 empty))))
→ False
```

} construct.  
} observer

DSL: ① Library of functions, implementaion details exposed

### Modules

Group related definitions (names, types) in a single file (named `M.hs`)

```
module M where
type Predicate a = a -> Bool
id :: a -> a
id = \x -> x
```

Hierarchy : module A.B.C.M in file `A/B/C/M.hs`

- definitions in other module M:

```
import M
```

- Explicit export Lists hode all other definitions

```
module M (id) where ...
--type Predicate a not exported
```

```
import Data.List (nub)

type IntegerSet = [Integer]

s1,s2 :: IntegerSet
s1 = insert 1 (insert 2 (insert 3 empty))
s2 = foldr insert empty [1..10]

empty :: IntegerSet
empty = []

insert :: Integer -> IntegerSet -> IntegerSet
insert x xs = x:xs

delete :: Integer -> IntegerSet -> IntegerSet
delete x xs = filter (/= x) xs

(|) :: Integer -> IntegerSet -> Bool
x | xs = elem x xs

(|) :: IntegerSet -> IntegerSet -> Bool
xs | ys = all (\x -> x | ys) xs

card :: IntegerSet -> Int
card xs = length (nub xs)

main :: IO ()
main = print $ 1 | s2
```

Code example 21: library-exposed.hs

- Abstract data types: export algebraic datatypes, but *not* its constructor functions

```
module M (Rose, leaf) where
data Rose a = Node a [Rose a] --constructor Node not exported
leaf :: a -> Rose a
leaf x = Node x []
```

- Export constructors:

```
module M (Rose (Node), leaf) where ...
module M (Rose (...), leaf) where ...
```

- Qualified imports to partition space:

```
import qualified M [as Nickname]
t :: M.Rose Char
t = M.leaf 'x'
```

```
:t fromJust
Not in scope: 'fromJust'
import Data.Maybe
:t fromJust
fromJust :: Maybe a -> a

import qualified Data.Maybe
:t Data.Maybe.fromJust
Data.Maybe.fromJust :: Maybe a -> a

import qualified Data.Maybe as DM
:t DM.fromJust
DM.fromJust :: Maybe a -> a
```

- Partially import module:

```
import Data.List (nub,maybe)
import Prelude hiding (otherwise)
otherwise :: Bool
otherwise = False
```

```
module SetLanguage
  (IntegerSet,
   empty,
   insert,
   delete,
   member
  ) where

data IntegerSet = IS [Integer]

empty :: IntegerSet
empty = IS []

insert :: IntegerSet -> Integer -> IntegerSet
insert (IS xs) x = IS (x:xs)

delete :: IntegerSet -> Integer -> IntegerSet
delete (IS xs) x = IS (filter (/= x) xs)

member :: IntegerSet -> Integer -> Bool
member (IS xs) x = elem x xs

module SetLanguage
  (IntegerSet,
   empty,
   insert,
   delete,
   member
  ) where

data IntegerSet = IS (Integer -> Bool)

empty :: IntegerSet
empty = IS (\_ -> False)

insert :: IntegerSet -> Integer -> IntegerSet
insert (IS f) x = IS (\y -> x == y || f y)

delete :: IntegerSet -> Integer -> IntegerSet
delete (IS f) x = IS (\y -> y /= x && f y)

member :: IntegerSet -> Integer -> Bool
member (IS f) x = f x
```

Code example 22: Two implementations of the SetLanguage module