Funktionale Programmierung Mitschrieb

Finn Ickler

January 28, 2016

„Avoid success at all cost “

Simon Peyton Jones

Contents

List of Listings

Vorlesung 1

```haskell
-- 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	λ-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 $\Leftrightarrow \{m \mid m \in m \in \{2, \ldots, n-1\}, n mod m = 0\} = \{\}$

```
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 ≡ as "denotes the same value as"
Apply f to value e: f ␣e
(juxtaposition, "apply", binary operator ␣, Haskell speak: infixL 10 ␣) = ␣has max precedence (10): f e_1 +e_2 ≡(f e_1) + e_2 ␣associates to the left g ␣f ␣e ≡ (g f) e Function composition:

- g (f e)
- Operator "." ("after") : (g.f) e (. = ∘) = 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

```
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

Prefix application of binary infix operator ⊕

$(\oplus) e_1 e_2 \equiv e_1 \oplus e_2$
(&&) `True False` ≡ `False`

Infix application of binary function f:

e_1 `` `f` `` e_2 ≡ `f` $e_1 e_2$
`` x `elem` xs ≡ x ∈ xs ``

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

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

Code example 6: Eigener ≈ Opperator

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)	`()`

```haskell
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)`
$(a_1, a_2, \ldots, a_n)$	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 habe side effects)	`print 42 :: IO ()`
		`getChar :: IO Char`
a `->` b	function from type a to b	`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:
  1. e_1 `++` $e_2 \equiv$ `(++)` $e_1 e_2$
  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 `...` `->` a_n `->` `b`
- Type constructor `->` associates to the right thus read the type as: a_1 `->` (a_2 `->` a_3 (`...` `->` (a_n `->` `b`)…))
- Enables partial application: ”Give me a value of type a_1, I'll give you a (n-1)-ary function of type a_2 `->` a_3 `->` `...` `->` a_n `->` `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 $x_1\ x_2 \ldots x_n = e$
- 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` x_1 x_2 `= e`

```
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

- Guards (introduced by |).

  `f` x_1 x_2 $\ldots$ x_n
  $|\, q_1$ `=` e_1
  $|\, q_2$ `=` e_n

```
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

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

Lokale Definitionen

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

$$
\begin{array}{l}
\texttt{f}\ \ x_1\ \ x_2\ \ \ldots\ \ x_n \\
\quad |\, q_1\ \ \texttt{=}\ \ e_1 \\
\quad |\, q_2\ \ \texttt{=}\ \ e_n \\
\ \ \texttt{where} \\
\qquad g_1\ \ \ldots\ \ \texttt{=}\ \ b_1 \\
\qquad g_i\ \ \ldots\ \ \texttt{=}\ \ b_i
\end{array}
$$

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

$$
\begin{array}{l}
\texttt{let}\ g_1\ \ \ldots\ \ \texttt{=}\ \ b_1 \\
\qquad\ g_2\ \ \ldots\ \ \texttt{=}\ \ b_1 \\
\texttt{in e}
\end{array}
$$

Haskells 2-dimensionale Syntax (Layout) (Forumbeitrag)

Hallo zusammen,
in der dritten Vorlesung hatte ich erwähnt, dass Haskells 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] -> [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 -> 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 ∀`xs`≠`[]`: `head xs : tail` = `xs`

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

Type Synonyms

- Introduce your own type synonyms. (type names : *U*ppercase) `type` t_1 `=` t_2

```
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]` ≡ `[x,x+1,x+2,...,y]`

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

- `x,s..y`≡ `[x,x+i,x+(2*i),...,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 ::`a_1 `->`...`->`a_k `-> b`
`f` $p_{11} \dots p_{1k} = e_1$
$\vdots \ \vdots \ \vdots \ \vdots$
`f` $p_{m1} \dots p_{nk} = e_n$
For all e_i `:: b` on a_i call `f` $x_1 x_2 \dots x_k$ each x_i is matched against patterns $p_{i1} \dots p_{in}$ in order. Result is e_r if the rth 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 $(p_1, \ldots, p_n)$	components of x_i match type component patterns	Those bound by the component patterns
[]	x_i == []	
$p_1 : p_2$	`head` x_1 matches p_1, `tail` 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)

```haskell
--(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` p_1 `|` q_{11} `->` e_{11}

$\vdots$

p_n `|` q_{n1} `->` e_{n1}

```haskell
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

```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

```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

```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:

$$\texttt{data T } a_1\ a_2\ \ldots\ a_n = K_1\ b11\ \ldots\ b_{1n_1}$$
$$|K_2\ b_{21}\ \ldots\ b_{2n_2}$$
$$\vdots \quad \vdots$$
$$|K_r\ b_{r1}\ \ldots\ b_{rnr}$$

- Defines *Type constructor* T and r *value constructor* with types
- $\texttt{K}_i :: b_{i1} \ldots b_{ini} \texttt{->}\ \texttt{T} a_1\ a_2 \ldots\ a_n$
- 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
`Eq`	equality (`==`,`/=`)
`Show`	printing (`show`)
`Ord`	ordering (`<`,`<=`,`max`)
`Enum`	enumeration (`[x..y]`)
`Bounded`	bounds (`minBound`,`maxBound`)

```haskell
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:

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

```haskell
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 = []
formList (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 $ formList $ filterList (> 3) $ mapList (+1) $ toList [1..5]
```

Code example 17: cons.hs

```haskell
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") whichi all *instances* of c must implement:

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

The t_i *must* mention a 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 => :: 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 [(a,b)] -> Maybe b
```

Class inheritance

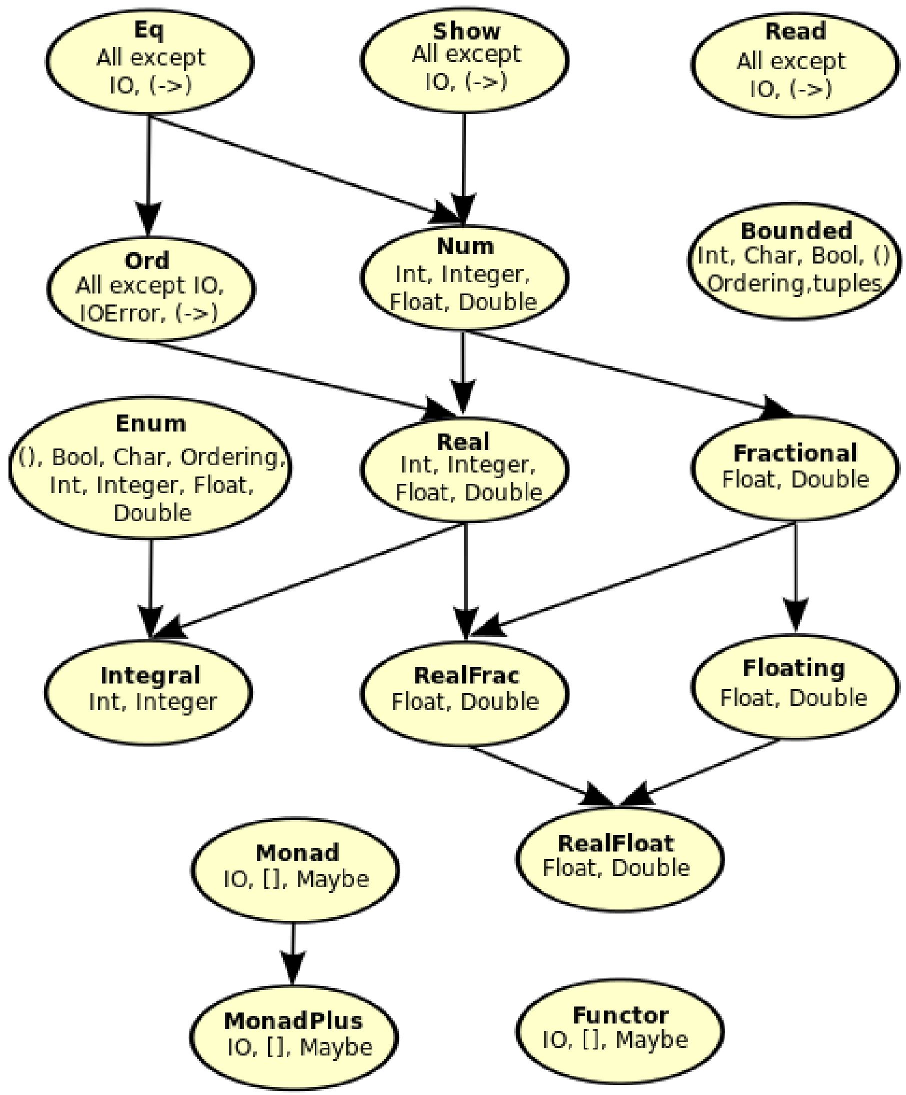
Defining class $(c_1 a, c_2 a, \dots)$ `=> (a where ...)` makes type class C a *subclass* of the c_1 C inherits all methods of the c_i.
(`a => t` implies $(c_1 a, c_2 a, \dots, Ca)$`=> t`)

```haskell
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


Eq
All except
IO, (->)
Show
All except
IO, (->)
Read
All except
IO, (->)
Ord
All except IO,
IOError, (->)
Num
Int, Integer,
Float, Double
Bounded
Int, Char, Bool, ()
Ordering,tuples
Enum
(), Bool, Char, Ordering,
Int, Integer, Float,
Double
Real
Int, Integer,
Float, Double
Fractional
Float, Double
Integral
Int, Integer
RealFrac
Float, Double
Floating
Float, Double
Monad
IO, [], Maybe
RealFloat
Float, Double
MonadPlus
IO, [], Maybe
Functor
IO, [], Maybe

Class Instances

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

```
instance C t where
f_1 = <def of f_1> --all f may be
        ⋮          --provided, minimal
f_n = <def of f_n> --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 ... :
`instance` $(c_1 a, c_2, c_3 b, \ldots)$ `=> (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 ...) intsances of classes $c_i \in$ {Eq, Ord, Enum,Bounded, Show, 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.
- Examples:

Domain	DSL
Os automation	Shell scripts
Typesetting	TEX, LATEX
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                                 } construct.
insert :: Integer    -> IntegerSet -> IntegerSet     }
delete :: Integer    -> IntegerSet -> IntegerSet     } observer
member :: Integer    -> IntegerSet -> Bool
member 3 (insert 1 (delete 3 (insert 2 (insert 3 empty))))
→ False
```

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
  ```

```haskell
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

  ```haskell
  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:

  ```haskell
  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
```

```haskell
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

Vorlesung 8

- Shallow DSL embedding : Semantiics of DSL operations directly expressed in terms of a host language value (e.g list or characteristic function).

  - constructors (`empty`,`insert`,`delete`) perform the work, harder to add

  - Observer (`member`) trivial

- *Deep* DSL embedding: DSL operations build an abstract syntax Tree (AST) that represents applications and arguments

  - constructors merely build the AST, very easy to add

  - observer: interpret (traverse) the AST and perform the work

```haskell
module SetLanguageDeep(IntegerSet(Empty,Insert,Delete),
    member,card) where

data IntegerSet = Empty
                  | Insert IntegerSet Integer
                  | Delete IntegerSet Integer
  deriving (Show)

member :: IntegerSet -> Integer -> Bool
member Empty          _ = False
member (Insert xs x) y = x == y || member xs y
member (Delete xs x) y = x /=y  && member xs y

card :: IntegerSet -> Integer
card Empty                        = 0
card (Insert xs x) | member xs x = card xs
                   | otherwise   = card xs + 1
card (Delete xs x) | member xs x = card xs - 1
                   | otherwise   = card xs
```

Code example 23: SetLanguageDeep.hs

```
:i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
          -- Defined in ‘GHC.Num’
instance Num Word -- Defined in ‘GHC.Num’
instance Num Integer -- Defined in ‘GHC.Num’
instance Num Int -- Defined in ‘GHC.Num’
instance Num Float -- Defined in ‘GHC.Float’
instance Num Double -- Defined in ‘GHC.Float’
:t 42
42 :: Num a => a
default ()
42
<interactive>:5:1:
    No instance for (Num a0) arising from a use of ‘it’
    The type variable ‘a0’ is ambiguous
    Note: there are several potential instances:
      instance Integral a => Num (GHC.Real.Ratio a)
        -- Defined in ‘GHC.Real’
      instance Num Integer -- Defined in ‘GHC.Num’
      instance Num Double -- Defined in ‘GHC.Float’
      ...plus three others
    In the first argument of ‘print’, namely ‘it’
    In a stmt of an interactive GHCi command: print it
default (Integer,Rational, Double)
42
42
42 / 3
14 % 1
42.1
421 % 10
default (Integer,Double)
```

```
module ExprDeepNum
    (Expr(..),
    eval
    ) where

data Expr =
   Val Integer
  |Add Expr Expr
  |Mul Expr Expr
  |Sub Expr Expr
  deriving(Show)

instance Num Expr where
  e1 + e2 = Add e1 e2
  e1 - e2 = Sub e1 e2
  e1 * e2 = Mul e1 e2
  fromInteger n = Val n
  abs    _ = undefined
  signum _ = undefined

eval :: Expr -> Integer
eval(Val n)     = n
eval(Add e1 e2)= eval e1 + eval e2
eval(Mul e1 e2)= eval e1 * eval e2
eval(Sub e1 e2)= eval e1 - eval e2
```

Code example 24: ExprDeepNum.hs

```haskell
module ExprDeep
    (Expr(..),
    eval
    ) where

data Expr =
   ValI Integer
   |ValB Bool
   |Add Expr Expr
   |And Expr Expr
   |EqZero Expr
   |If Expr Expr Expr

instance Show Expr where
  show (ValI n) = show n
  show (ValB b) = show b
  show (Add e1 e2) = show e1 ++ " + " ++ show e2
  show (And e1 e2) = show e1 ++ "∧" ++ show e2
  show (EqZero e)  = show e ++ "== 0"
  show (If p e1 e2) = "if " ++ show p ++ " then "
    ++ show e1 ++ " else " ++ show e2

eval :: Expr -> Either Integer Bool
eval (ValI n) = Left n
eval (ValB b) = Right b
eval (Add e1 e2) = case (eval e1, eval e2) of
                     (Left n1, Left n2) -> Left (n1 + n2)
eval (And e1 e2) = case (eval e1, eval e2) of
                      (Right n1, Right n2) -> Right (n1 && n2)
eval (EqZero e)    = case eval e of
                      Left n -> Right (n == 0)
                      Right b -> Right False
eval (If p e1 e2) = case eval p of
                      Right b -> if b then eval e1 else eval e2
```

Code example 25: ExprDeepNum.hs

Generalized Algebraic Datatypes

Idea :

- Encode the type of a DSL expression (here : Integer or Bool) in its *Haskell type*
- Use Haskell's type checker to ensure at *compile time* that only well-typed DSL expressions are built:

GADTs

- Language extensions: `{-## LANGUAGE GADTs ##-}`
- Define entirely new parameters `Type T`, its (value) constructors k_i and their type signatures

$$
\begin{array}{l}
\texttt{data T } a_1 \;\; a_2 \;\; \ldots \;\; a_n \;\; \texttt{where} \\
\quad k_1 \;\; \texttt{::} \;\; b_{11} \;\; \texttt{->} \;\; \ldots \;\; b_{1n_1} \;\; \texttt{->} \;\; \texttt{T} \;\; t_{11} \;\; t_{12} \ldots \;\; t_{1n} \\
\quad k_2 \;\; \texttt{::} \;\; b_{21} \;\; \texttt{->} \;\; \ldots \;\; b_{2n_2} \;\; \texttt{->} \;\; \texttt{T} \;\; t_{21} \;\; t_{22} \ldots \;\; t_{2n} \\
\ldots
\end{array}
$$

```haskell
{-# LANGUAGE GADTs #-}
module ExprDeep
    (Expr(..),
    eval
    ) where

data Expr a where
   ValI   :: Integer                          -> Expr Integer
   ValB   :: Bool                             -> Expr Bool
   Add    :: Expr Integer -> Expr Integer     -> Expr Integer
   And    :: Expr Bool -> Expr Bool           -> Expr Bool
   EqZero :: Expr Integer                     -> Expr Bool
   If     :: Expr Bool -> Expr a -> Expr   a -> Expr a

instance Show (Expr a) where
  show (ValI n) = show n
  show (ValB b) = show b
  show (Add e1 e2) = show e1 ++ " + " ++ show e2
  show (And e1 e2) = show e1 ++ "Δ" ++ show e2
  show (EqZero e)  = show e ++ "== 0"
  show (If p e1 e2) = "if " ++ show p ++ " then " ++ show e1 ++ " else " ++ show

eval :: Expr a -> a
eval (ValI n) = n
eval (ValB b) = b
eval (Add e1 e2) = eval e1 + eval e2
eval (And e1 e2) = eval e1 && eval e2
eval (EqZero e)  = eval e == 0
eval (If p e1 e2) = if eval p then eval e1 else eval e2
```

Code example 26: ExprDeepTyped.hs

Vorlesung 9

```haskell
{-# LANGUAGE FlexibleInstances #-}
module ExprEmbedding (Expr, Env, val, add, var,
bnd,AST (..)) where
class Expr a where
  val :: Integer          -> a
  add :: a           -> a -> a
  var :: String           -> a
  bnd :: (String,a) -> a -> a


type Env = [(String,Integer)]

-- Shallow Ebedding #1
instance Expr (Env -> Integer) where
  val n         = \_ -> n
  add e1 e2     = \e -> e1 e + e2 e
  var v         = \e -> case lookup v e of
                    Just n -> n
                    Nothing -> error (v ++ " is unknown")
  bnd (v,e1) e2 = \e -> e2 ((v,e1 e):e)
-- Shallow Embedding #2

instance Expr String where
  val n = show n
  add e1 e2 = e1 ++ " + " ++ e2
  var v = v
  bnd (v,e1) e2 = "let " ++ v ++ " = " ++ e1 ++ " in (" ++ e2 ++ ")"

data AST a = Val a
           | Add (AST a) (AST a)
           | Var String
           | Let String (AST a) (AST a)
           deriving (Eq, Show)

instance Expr (AST Integer) where
  val n      =Val n
  add e1 e2  =Add e1 e2
  var v      =Var v
  bnd (v,e1) e2 = Let v e1 e2
```

Code example 27: ExprEmbedding.hs

```
import           ExprEmbedding

prog :: Expr a => a
prog = bnd ("x", val 3) (add (bnd ("x", val 2) (var "x")) (var "x"))
simplify :: AST Integer -> AST Integer
simplify e = repeat rewrite e
  where
    repeat :: Eq a => (a -> a) -> a -> a
    repeat f = until (\x -> f x == x) f
    rewrite :: AST Integer -> AST Integer
    rewrite (Add (Val 0) e2)                = rewrite e2
    rewrite (Add e1 (Val 0))                = rewrite e1
    rewrite (Add e1 e2)                     = Add (rewrite e1) (rewrite e2)
    rewrite (Let _ _ e2@(Val _))           = rewrite e2
    rewrite (Let v e1 (Val v'))   | v == show v' = rewrite e1
    rewrite (Let v e1 e2 )                  = Let v (rewrite e1) (rewrite e2)
    rewrite e                                = e

main :: IO()
main = print (prog :: String)
```

Code example 28: expr-embeddings.hs

Shallow Embedding of a String Matching DSL

- Pattern:

  1. Given a string, a pattern returns a *list of matches*. Match failure? return the *empty list* (of matches)
  2. A match consists of a value (e.g the match of characters, tokens parse tree) and the residual string to match
     Thus: `type Pattern a = String -> [(a,String)]`

Replace failure by a list of successes

A pattern of things is list of things and strings

Torsten Grust, 10.12.2015

- DSL design:

Pattern	DSL Function `Char -> String ([Char,String])`
match literal	`lit :: Char -> Pattern Char`
match empty string	`empty :: a -> Pattern a`
fail always	`fail :: Pattern a`
alternative	`alt :: Pattern a -> Pattern a -> Pattern a`
sequence	`seq ::(a -> b -> c)-> Pattern a -> Pattern b -> Pattern c`
repetition	`Pattern a -> Pattern [a]`

```haskell
module PatternMatching (Pattern,
                        module Prelude,
                        lit, empty, fail,
                        alt, seq, rep, rep1,
                        alts, seqs, lits, app) where

import           Prelude hiding (fail, seq)

-- Given a string, a pattern returns the (possibly empty) list of
-- possible matches.  A match consists of a match value (e.g., matched
-- the matched character(s), token, or parse tree) and the residual string
-- left to match:

type Pattern a = String -> [(a, String)]

-- BASIC PATTERNS

-- match character c, returning the matched character
lit ::  Char -> Pattern Char
lit _c []                  = []
lit c  (x:xs) | c == x     = [(c, xs)]
              | otherwise = []

-- match the empty string, return v
empty :: a -> Pattern a
empty v xs = [(v, xs)]

-- fail always (yields empty list of matches)
fail :: Pattern a
fail _ = []

-- COMBINE PATTERNS

-- match p or q
alt :: Pattern a -> Pattern a -> Pattern a
alt p q xs = p xs ++ q xs

-- match p and q in sequence (use f to combine match values)
seq :: (a -> b -> c) -> Pattern a -> Pattern b -> Pattern c
seq f p q xs = concat (map (\(v1, xs1) ->
                         map (\(v2, xs2) -> (f v1 v2, xs2))
                             (q xs1))
                         (p xs))

-- An alternative (more consise and readable) implementation of seq
-- based on list comprehension syntax:
--
-- seq f p q xs = [ (f v1 v2, xs2) | (v1, xs1) <- p xs, (v2, xs2) <- q xs1 ]

-- match p repeatedly (including 0 times)
rep :: Pattern a -> Pattern [a]
rep p = alt (seq (:) p (rep p)) (empty [])

-- match p repeatedly, but at least once
rep1 :: Pattern a -> Pattern [a]
rep1 p = seq (:) p (rep p)
```

```haskell
import           Prelude         hiding (fail, seq)

import           PatternMatching

-- Make use of the fact that the pattern matching DSL is *embedded*
-- into Haskell: define new functions (abstractions) that combine
-- simple patterns

-- Example:
--
-- Match a fully parenthesized arithmetic expression over integers,
-- e.g. ((4*10)+2)


-----------------------------------------------------------------------
-- Variant 1: return list of matched characters

digit :: Pattern Char
digit = alts [ lit d | d <- ['0'..'9'] ]

number :: Pattern String
number = rep1 digit

op :: Pattern String
op = alts [ lits o | o <- ["+", "-", "*", "/"] ]

expr :: Pattern String
expr = alts [ number, app concat (seqs [lits "(", expr, op, expr, lits ")"]) ]


-----------------------------------------------------------------------
-- Variant 2: return a simple AST for the matched expression

data Expr a =
    Num a
  | Op (Expr a) String (Expr a)
  deriving (Show)

number' :: Pattern (Expr Integer)
number' = app (Num . read) (rep1 digit)

expr' :: Pattern (Expr Integer)
expr' = alts [ number', seq (\_ (e1,(o,(e2,_))) -> Op e1 o e2)
                            (lit '(') (seq (,)
                                      expr' (seq (,)
                                            op (seq (,)
                                               expr' (lit ')'))))
             ]

-----------------------------------------------------------------------

main :: IO ()
main = do
  print $ rep1 digit "1234.56"
  print $ lits "abc" "abcdef"
  print $ expr "((4*10)+2)"
```

Vorlesung 10

Lazy Evaluation

To execute a programm, Haskell *reduces* expression to values. Haskell uses *normal order reduction* to select the next expression to reduce:

- The *outermost* reducable expression (redex) is reduced first.
- ⇒ Function application are reduced first *before* their arguments.
- If no further redex is found, the expression is in *normal form.* and reduction terminates.

```
fst :: (a,b) -> a
fst (x,y) = x
sqr :: Num a => a -> a
sqr x = x * x
----------------
fst (sqr (1 + 3), sqr 2) → sqr (1 + 3)         [fst]
                         → (1 + 3) * (1 + 3)   [sqr]
                         → 4 * 4               [+/+]
                         → 16                  [*]
```

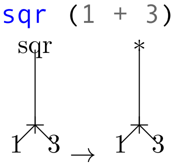
```
(define-racket-procedures pair
  make-pair
  pair?
  (pair-fst)
  (pair-snd))

(define fst
  (lambda (p)
    (pair-fst p)))

(define sqr
  (lambda (x) (* x x)))

;Racket uses applicative order reduction (innermost first)
```

Haskell avoids the duplication of work through *graph reduction*: Expression are shared (referenced more than once) instead of duplicated. Reduction of `sqr (1 + 3)`:



Lazy evaluation: normal order reduction + sharing + WHNF
thunks

```haskell
import           Prelude hiding (fst)

fst :: (a,b) -> a
fst (x,y) = x

sqr :: Num a => a -> a
sqr a = a * a
bomb :: [String]
bomb = "" : bomb
main = print $ fst (sqr (1 + 3),bomb)
```

Code example 29: This Programm compiles in Haskell, but not in Racket

WHNF

An expression e ist in *weak head normal form* (WHNF) if it is of the following form:

① `v` (where `v` is an *atomic* value `Integer`, `Char`, `Bool`, ...)

② $c\ e_1\ e_2\ \ldots e_n$ (where c is an n-ary constructor (like `(:)`))

③ $f\ e_1\ e_2\ \ldots e_m$ (where f is a n-ary function, $m < n$)

Haskell reduces values to WHNF only (stop criteria for reducion) unless we request reduction to normal fprm (e.g when printing result)

Example expressions in WHNF

```haskell
42 -- ①
(sqr 2,sqr 4) -- ②
f x = map f xs -- ② (:)
Just (40 + 2) -- ② Just
(* (40 + 2)) --  ③ * binary
(\x -> 40 + 2) --  ③ * binary
```

```haskell
(1 + 3) : []
[4]
it :: [Integer]
let xs = (1 + 3) : []
xs :: [Integer]
:sprint xs
xs = [_]
```

Lazy Evaluation and Bottom (⊥)

Some Haskell expressions have the value *bottom* (*perp*) Examples: `error ".."`, `undefined`, `bomb`. Lazy evaluation admits functions that return a non-bottom value even if they receive ⊥ as an argument (also:*non-Strict functions*). N-ary function is *strict* in its i-th argument, if $f\ x_1\ \ldots\ x_{i-1} \perp x_{i+1} \ldots x_n = \perp$
Examples:

- `const :: a -> b -> a` strict in first, non-strict in second argument
- `(&&) :: Bool -> Bool -> Bool`

⚠ If a function *pattern matches* an argument, Haskell semantics define it to be strict in that argument.
Example:

```
data T = T Int
f :: T -> Int
f (T x) = 42
f undefined     → undefined
f (T undefined) → 42
```

```
min [8,6,1,7,5] → (head . isort (<)) [8,6,1,7,5]   [min]
→ head (isort (<) [8,6,1,7,5])                      [(.)]
→ head (ins 8 (ins 6 (ins 1 (ins 7 (ins 5 []))))) [isort.2*]
→ head (ins 8 (ins 6 (ins 1(ins 7 [8]))))          [ins.1]
→ head (ins 8 (ins 6 (ins 1 (5 : ins 7 []))))      [ins.3]
→ head (ins 8 (1 : ins 6 ( 5 : ins 7 [])))         [ins.2]
→ (1 : ins 8 (ins 6 (5 : ins 7 [])))               [ins.3]
→ 1
```

```
min [1..1000000]
1
it :: Integer
(1.50 secs, 521,738,256 bytes)
min [1..10000000]
1
it :: Integer
(15.43 secs, 5,164,721,896 bytes)
```

```haskell
import           Debug.Trace
data T1 = T1 Int

f :: T1 -> Int
f (T1 x) = 42

g :: Int -> Int
g x = 42

a :: T1
a = trace "a has been evaluated" (T1 0)

b :: Int
b = trace "b has been evaluated" 0
{-
newtype T2 = T2 Int
h :: T2 -> Int
h (T2 x) = 42
-}
main :: IO ()
main = do
  print $ f a
  print $ g undefined
  print $ g b
```

Code example 30: Bottom type

```haskell
import           Prelude hiding (min)

min :: Ord a => [a] -> a
min = head . isort (<)

isort :: Ord a => (a -> a -> Bool) -> [a] -> [a]
isort (<<<) []       = []                        --[isort.1]
isort (<<<) (x:xs)   = ins x (isort (<<<) xs)    --[isort.2]
  where
    ins x []           = [x]                     --[isort.1]
    ins x (y:ys)       | x <<< y = x:y:ys        --[isort.2]
                       | otherwise = y:ins x ys --[isort.3]
main :: IO ()
main = do
  print $ isort (<) [8,6,1,7,5]
  print $ min [8,6,1,7]
```

Code example 31: Finding the minimum by sorting the list

Vorlesung 11

Infinite Lists (Data Structures)

One consequence of lazy evaluation, programs can handle *infinite Lists* as long as any run will inspect onlt a finite prefix of such a list.
Enables a modular programming style:

1. **generator functions** produce an infinite number of solutions/approximations
2. **test functions** select one (or finite number of) solutions from this infinie list)

Example: Newton-Raphson square root approximation Iteratively approximate the square root of x

1. $a_0 = x/2$
2. $a_{i+1} = (a_i + x/a_i)/2$ $\qquad a = (a + x/a/2) \Leftrightarrow a = \sqrt{x}$

```
-- Demonstrate modular program construction through laziness:
-- value generation (here: iterate) and consume/test (here: within)
-- can be implemented separately.
--
-- Can replace test (within → relative) without modifying the generator.
--
-- See John Hughes, "Why Functional Programming Matters", Section 4.1


import Prelude hiding (iterate)

-- [x, f x, f (f x), f (f (f x)), ...]
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

-- Consume list until two adjacent elements are
-- 1. within eps of each other
-- 2. differ by a factor less than eps
within :: (Ord a, Num a) => a -> [a] -> a
within eps (x1:x2:xs) | abs (x1 - x2) <= eps = x2
                      | otherwise             = within eps (x2:xs)

relative :: (Ord a, Fractional a) => a -> [a] -> a
relative eps (x1:x2:xs) | abs (x1/x2 - 1) <= eps = x2
                        | otherwise               = relative eps (x2:xs)
```

```haskell
-- Square root of x using the Newton-Raphson algorithm:
--
--    a   = x / 2
--    a+1 = (a + x / a) / 2
--
-- Why does this work?  If the approximations a converge to some
-- limit a, then:
--
--    a = (a + x / a) / 2
--   2a = a + x / a
--    a = x / a
--   a² = x
--    a = √x

sqroot :: Double -> Double -> Double
sqroot eps x = within eps (iterate next a0)
--                       ↑
--              relative
  where
    -- initial approximation
    a0 :: Double
    a0 = x / 2
    -- find next a+1, given a
    next :: Double -> Double
    next a = (a + x / a) / 2


main :: IO ()
main = print $ sqroot 0.001 81
```

Example (Tic-Tac-Toe game tree):
Build the (potentially huge) *tree of possible moevs* for the Tic-Tac-Toe Board game. Evaluate promise of game position.
Plan:

① Find representation of game position (board + player next up)

1	2	3
4	5	6
X	O	X

② provide pretty-printing for game

③ Define initial position and possible moves : `moves :: Position -> [Position]`

④ Evaluate a given position: `static :: Position -> Int`

⑤ Build a game tree of positions:
`gameTree :: Position -> Tree Position`

⑥ Pattern than simple statci evaluate now evaluate portions based on possible game futures:
gameTree,position evaluate bottom up

⑦ Optimization ($\alpha - \beta$-algorithm)

```haskell
import           Data.List              (intersperse, transpose)
import           Text.PrettyPrint.Boxes

data Player = X | O
  deriving (Eq, Show)

type Square = Either Int Player
type Board = [[Square]]

data Position = Position Board Player

showSquare :: Square -> String
showSquare = either show show

showBoard :: Board -> [String]
showBoard = frame "┌─┬─┬─┐" "├─┼─┼─┤" "└─┴─┴─┘" .
            map (concat . frame "│" "│" "│" . map showSquare)
  where
    frame :: a -> a -> a -> [a] -> [a]
    frame l m r xs = [l] ++ intersperse m xs ++ [r]

instance Show Position where
  show (Position b _) = unlines (showBoard b)

initial :: Position
initial = Position (map (map Left) [[1,2,3],[4,5,6],[7,8,9]]) O
moves :: Position -> [Position]
moves pos@(Position b _) = map (move pos) (openSquares b)
  where
    openSquares :: Board -> [Square]
    openSquares b = [ Left sq | Left sq <- concat b ]
    move :: Position -> Square -> Position
    move (Position b p) sq = Position (map (map (place sq p)) b) (next p)
    place :: Square -> Player -> Square -> Square
    place sq p sq' | sq == sq' = Right p
    place _  _ sq'             = sq'

    next :: Player -> Player
```

```
    next X = O
    next O = X

-- Static evaluation of position p: has computer (X) won the game?
--  1: X won the game
-- -1: O won the game
--  0: game still undecided
static :: Position -> Int
static (Position b p) = if won b then case p of
                                         X -> -1
                                         O -> 1
                                  else 0
  where
    won :: Board -> Bool
    won b = any full (diagonals b ++ rows b ++ cols b)
    full :: [Square] -> Bool
    full [Right p1, Right p2, Right p3] = p1 == p2 && p2 == p3
    full _                              = False
    diagonals, rows, cols :: Board -> [[Square]]
    diagonals [[a1, _,a3],
               [ _,b2, _],
               [c1, _,c3]] = [[a1,b2,c3], [a3,b2,c1]]
    rows = id
    cols = transpose

data Tree a = Node a [Tree a]
  deriving Show

class Boxable a where
  box :: a -> Box

instance Boxable Int where
  box x = text (show x)

instance Boxable Position where
  box (Position b _) = vcat top (map text (showBoard b))

instance (Boxable a, Boxable b) => Boxable (a,b) where
  box (x,y) = box x <+> box y

instance Boxable a => Boxable (Tree a) where
  --
  -- ├----
  -- │
  -- └----
  box (Node x []) = box x
```

```haskell
  box (Node x ts) =
    vcat top ([box x]
                ++
              map (\b -> stem (rows b) <> b)
                (init branches)
                ++
              [char '└' <> last branches])
    where
      branches :: [Box]
      branches = map (\t -> text "----" <> box t) ts
      stem :: Int -> Box
      stem n = vcat top (char '├' : replicate n (char '│'))

repTree :: (a -> [a]) -> a -> Tree a
repTree f x = Node x (map (repTree f) (f x))

-- The complete game tree (if explored fully)
gameTree :: Position -> Tree Position
gameTree p = repTree moves p

-- Cut off game subtrees whenever a winning/losing position
-- has been reached
cutOff :: Tree Int -> Tree Int
cutOff (Node 0 ts) = Node 0 (map cutOff ts)
cutOff (Node x _)  = Node x []

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Node x ts) = Node (f x) (map (mapTree f) ts)

-- prune n t: cut off tree t at depth n
prune :: Int -> Tree a -> Tree a
prune 0 (Node x _)  = Node x []
prune n (Node x ts) = Node x (map (prune (n - 1)) ts)

maximize, minimize :: Ord a => Tree a -> a
maximize (Node x []) = x
maximize (Node _ ts) = maximum (map minimize ts)
--                          |             |
--          we will take the       assume opponent (O) will take his best move
--          best move possible     (= the worst move from our perspective)

minimize (Node x []) = x
minimize (Node _ ts) = minimum (map maximize ts)

-- evaluate a position from the viewpoint of the computer (X)
--
```

```haskell
-- modular specification thanks to lazy evaluation
evaluate :: Position -> Int
evaluate = maximize . cutOff . mapTree static . gameTree

-----------------------------------------------------------------------
-- Optional optimization
-- (alpha-beta algorithm)
--
--                  ℓ
--                  ↓
--     max
--     ├----min
--     |     └----1
--     |     |
--     |     └----1
--     |
--     └----min
--           ├----0
--           |
--           ├----?   ← We're looking for the maximal minimum which will
--           |           at least be 1 (see ).  Due to 0 (see ), this minimum
--           └----?   ← will be 0 or less  need not know value of the '?'
--
--   - Need a formulation that can see all lists in layer ℓ at once
--     (list of lists, see `[[a]] in `mapmin`/`mapmax` below)

maximize', minimize' :: Ord a => Tree a -> [a]
maximize' (Node x []) = [x]
maximize' (Node _ ts) = mapmin (map minimize' ts)
  where
    mapmin :: Ord a => [[a]] -> [a]
    mapmin (xs:xss) = minimum xs : omit (minimum xs) xss
    --                                    |
    --                     largest minimum found so far (pot)

    -- we're only interested in the maximal minimum, so
    -- omit the minimum computation for those lists xs in which we find
    -- any element that does not exceed pot (the largest minimum found so far);
    -- finding such an element (with `any') does not need to inspect all
    -- elements in xs => savings thanks to lazy evaluation
    omit :: Ord a => a -> [[a]] -> [a]
    omit pot []                          = []
    omit pot (xs:xss) | any (<= pot) xs = omit pot xss
                      | otherwise       = minimum xs : omit (minimum xs) xss

minimize' (Node x []) = [x]
```

```
minimize' (Node _ ts) = mapmax (map maximize' ts)
  where
    mapmax :: Ord a => [[a]] -> [a]
    mapmax (xs:xss) = maximum xs : omit (maximum xs) xss

    omit pot []                            = []
    omit pot (xs:xss) | any (>= pot) xs = omit pot xss
                      | otherwise       = maximum xs : omit (maximum xs) xss

-- evaluate a position from the viewpoint of the computer (X),
-- NB: no pruning necessary, can evaluate full game tree
evaluate' :: Position -> Int
evaluate' = maximum . maximize' . cutOff . mapTree static . gameTree

-----------------------------------------------------------------------


main :: IO ()
main = do
  print initial
```

Vorlesung 12

Functor

Type class Functor embodies the application of a functor to the elements (or: inside) if a structure. while leaving structure (or: outside) alone.
Examples:

```
map :: (a -> b) -> [a] -> [b]
mapTree :: (a -> b) -> Tree a -> Tree b
```

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

- Note: f is a *type constructor* that receives exactly one argument (Functor is also called a *constructor class*)
  Examples:

  ```
  instance Functor [] where
    fmap = map

  instance Functor Tree where
    fmap = mapTree

  instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing  = Nothing
  ```

Type constructors can be partially applied, Uses prefix notation:
`a -> b` ≡ `(->) a b`
`(a,b)` ≡ `(,) a b`
`[a]` ≡ `[] a b`
Examamples (deffine type constructors):

```
Type Flagged = (,) Bool
Type Indexed = (->) Int
Type MayFail e = Either e
```

```
instance Functor (Either e) where
  fmap f (Left e)  = Left e
  fmap f (Right e) = Right (f x)
```

```
import           Text.Read (readEither)
readRoundToNearest :: Integral a => a -> String -> Either String a
readRoundToNearest n = fmap toNearest . readEither
  where
    toNearest x = n * round (x / fromIntegral n)

main :: IO ()
```

```
main = do
  print $ readRoundToNearest 10 "42"
  print $ readRoundToNearest 10 "BB-8"

instance Functor Flagged where
  fmap f (b,x)) = (b,f x)
--fmap :: (a -> b) -> (,) Bool a -> (,) Bool b

instance Functor Indexed where
  fmap f g = f . g
--fmap :: (a->b) -> (Int -> a) -> (Int -> b)
```

kinds ("Types of Types")

kind	describes	example
*	types	`Int,Bool,(Int,Bool),[Char]`
* -> *	unary Type constuctors	`Maybe,[]`
x -> x -> x	binary Type constructors	`Either, (,), (->)`

```
:k Int
Int :: *
:k (Float,Bool)
(Float,Bool) :: *
:k Maybe
Maybe :: * -> *
data Tree a = Node a [Tree a]
data Tree a = Node a [Tree a]
:k Tree
Tree :: * -> *
:k (->)
(->) :: * -> * -> *
:k (,)
(,) :: * -> * -> *
data Z c e = Z (c e)
type role Z representational nominal
data Z (c :: * -> *) e = Z (c e)
:k Z
Z :: (* -> *) -> * -> *
```

Functor Laws

1. `fmap id = id`
2. `fmap f . fmap g = fmap (f . g)`

```
data Pred i a = T |
                F |
          Var i a | And (Pred i a)  (Pred i a) | Or (Pred i a)  (Pred i a)
```

```
  deriving (Eq,Show)


eval :: [Bool] -> Pred Int a -> Bool
eval _ T = True
eval _ F = False
eval env (Var n _) = env !! n
eval env (And p1 p2) = eval env p1 && eval env p2
eval env (Or p1 p2) = eval env p1 || eval env p2

instance Functor (Pred i) where
  fmap _ T = T
  fmap _ F = F
  fmap f (Var n v) = Var n (f v)
  fmap f (And p1 p2) = And (fmap f p1) (fmap f p2)
  fmap f (Or p1 p2) = Or (fmap f p1) (fmap f p2)

name :: Show a => String -> a -> String
name n v = n ++ show v

quote :: String -> String
quote v = v ++ "'''"

expr :: Pred Int Int
expr = And (Var 0 0) (Or (Var 1 1) F)

main :: IO ()
main = do
  print $ eval [True,False] expr
  print $ fmap (quote . name "v") expr
  --Test the Functor Laws
  print $ fmap id expr == id expr
  print $ fmap (quote . name "v") expr == (fmap quote . fmap (name "v")) expr
```

Vorlesung 13

Applicative

compare:

```
($) :: (a -> b) -> a -> b
```

```
<$> :: Functor f => (a -> b) -> f a -> f b
```

```
(<*>) Applicative f => f (a -> b) -> (f a) -> f b
```

Read (<x>) as apply "Tie Figher"

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>):: f (a -> b) -> f a -> f b
```

Make any Applicative f a Functor

```
class Functor f a where
  fmap g x = pure g <x> x
```

```
pure 42 :: [Int]
[42]
it :: [Int]
pure 42 :: Either a Int
Right 42
it :: Either a Int
pure 42 :: ([a],Int)
([],42)
it :: ([a], Int)
pure 42 :: (Bool,Int)
No instance for (Monoid Bool) arising from a use of ‘pure’
In the expression: pure 42 :: (Bool, Int)
In an equation for ‘it’: it = pure 42 :: (Bool, Int)
```

Applicative embodies:

1. function application on the level of (constrained) values, and
2. combination of the various structures

```
                    preserve
                    ___=____
                    |      |
fmap :: (a -> b) -> f a -> f b

(<*>) :: f (a -> b) -> f a -> f b
         |             |      ↑
         |_____________|      |
                |             |
                |_____________|
```

```
[(+ 1), (* 10)] <*> [1,2,3]
[2,3,4,10,20,30]
it :: [Integer]
:i (,)
data (,) a b = (,) a b          -- Defined in ‘GHC.Tuple’
instance (Bounded a, Bounded b) => Bounded (a, b)
  -- Defined in ‘GHC.Enum’
instance (Eq a, Eq b) => Eq (a, b) -- Defined in ‘GHC.Classes’
instance Functor ((,) a) -- Defined in ‘GHC.Base’
instance (Ord a, Ord b) => Ord (a, b) -- Defined in ‘GHC.Classes’
instance (Read a, Read b) => Read (a, b) -- Defined in ‘GHC.Read’
instance (Show a, Show b) => Show (a, b) -- Defined in ‘GHC.Show’
instance Monoid a => Applicative ((,) a) -- Defined in ‘GHC.Base’
instance Foldable ((,) a) -- Defined in ‘Data.Foldable’
instance Traversable ((,) a) -- Defined in ‘Data.Traversable’
instance (Monoid a, Monoid b) => Monoid (a, b)
  -- Defined in ‘GHC.Base’
```

Interlude: Monoid

Type class Monoid a represents combinable values of type a:

```
class Monoid a where
  mempty   :: a           --empty, neutral element
  mappend :: a -> a -> a --combination
  mconcat :: [a] -> a    --is implemented by default
```

Examples :

- $(\varnothing, t)$ $(\text{true}, \wedge)$ $(\text{false}, \vee)$ $([\,], (++))$

```
mempty :: Sum Int
Sum {getSum = 0}
it :: Sum Int
mempty :: Product Int
Product {getProduct = 1}
it :: Product Int
:i Product
newtype Product a = Product {getProduct :: a}
          -- Defined in ‘Data.Monoid’
instance Bounded a => Bounded (Product a)
  -- Defined in ‘Data.Monoid’
instance Eq a => Eq (Product a) -- Defined in ‘Data.Monoid’
instance Num a => Num (Product a) -- Defined in ‘Data.Monoid’
instance Ord a => Ord (Product a) -- Defined in ‘Data.Monoid’
instance Read a => Read (Product a) -- Defined in ‘Data.Monoid’
instance Show a => Show (Product a) -- Defined in ‘Data.Monoid’
instance Num a => Monoid (Product a) -- Defined in ‘Data.Monoid’
2 `mappend` 21 :: Product Int
Product {getProduct = 42}
it :: Product Int
mempty :: All
All {getAll = True}
it :: All
mempty :: Any
Any {getAny = False}
it :: Any
[1..10] `mappend` [1..20]
[1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
it :: [Integer]
```

Monoid Laws

```
mempty `mappend` xs = xs
xs `mappend` mempty = xs
xs `mappend` (ys `mappend` zs) = (xs `mappend` ys) `mappend` zs
```

Applicative Instances

```
instance Applicative Maybe where
pure x = Just x
Just f <*> Just x = Just (f x)
_      <*> _      = Nothing

instance Monoid c => Applicative ((,),c) where
  pure x = (mempty c, x)
  (c1,f)  <*> (c2,x) = (c1 `mappend` c2 ,f x)
```

```
instance Applicative [] where
  pure x     = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
```