MOD08: Functional Programming

Types and Typeclasses

Marco Gerards

01-05-2019

Organisation of the course

Block	Topic
1	- Introduction to Functional Programming
	- Higher order functions
2	- Types and type classes
	- Parsing (application)
3	- Advanced type classes
	- Parser combinators and ParSec (application)
4	- Code generation (application)
	- Motivational lecture (application)
5-7	- Project

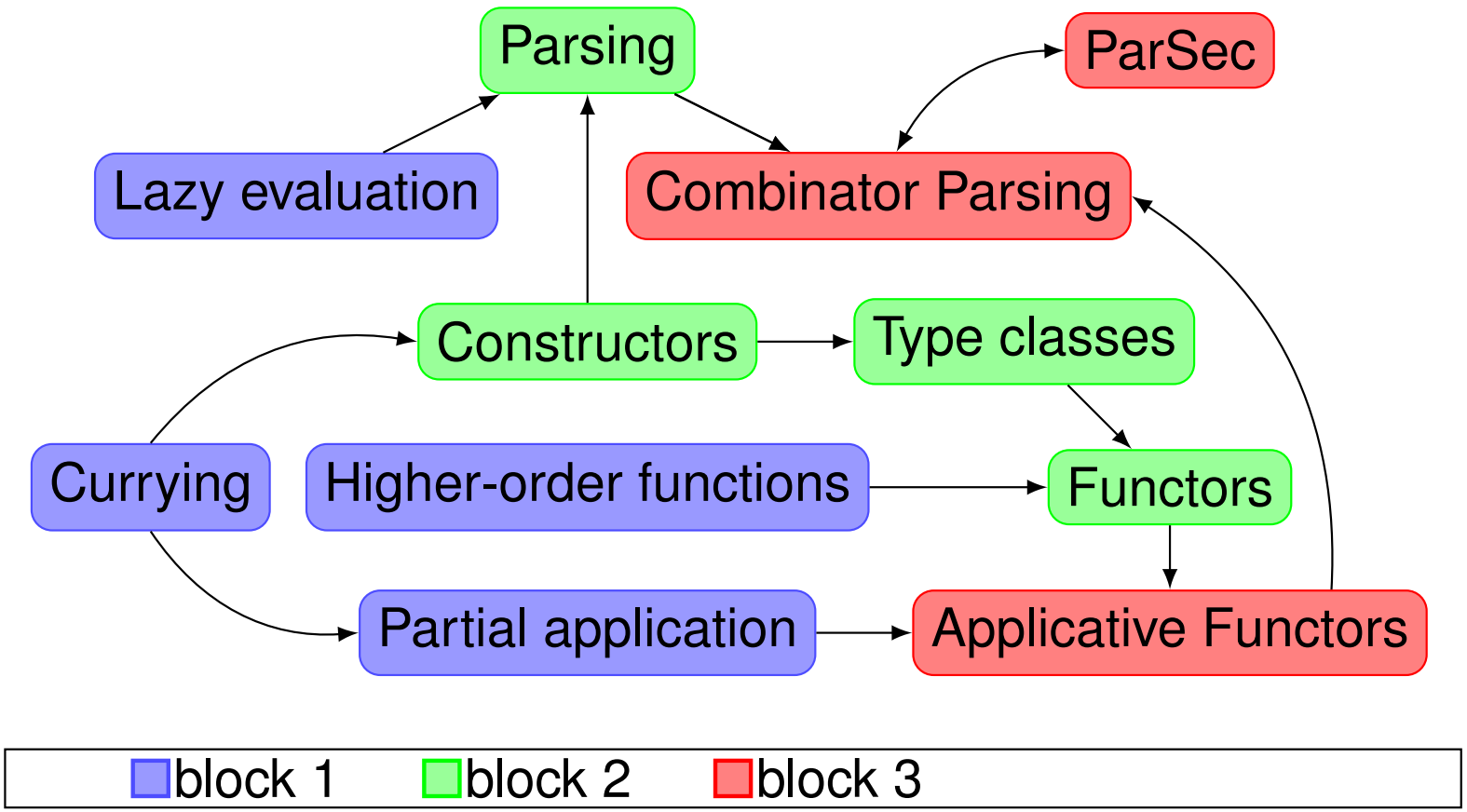
Organisation of the course

Block	Topic
1	- Introduction to Functional Programming
	- Higher order functions
2	- Types and type classes
	- Parsing (application)
3	- Advanced type classes
	- Parser combinators and ParSec (application)
4	- Code generation (application)
	- Motivational lecture (application)
5-7	- Project

This lecture: learning goals

- Explain and take advantage of the evaluation and execution mechanisms of FP
  - Sharing
- Explain and use the typical types and data structures in FP
  - Type constructors and data constructors
    - Lists, Tuples, Maybe and Either
  - Type classes
    - `Show`, `Eq`
    - `Functor`

Connection of some of the topics between blocks



Let expressions

```
f x y = let c = 42 in c * (c*x + y)
```

Let expressions

```
f x y = let c = 42 in c * (c*x + y)
```

Compare with:

```
f x y = c * (c*x + y)
 where c = 42
```

Let expressions

```
f x y = let c = 42 in c * (c*x + y)
```

Compare with:

```
f x y = c * (c*x + y)
 where c = 42
```

Differences:

- `where` declares a local scope (syntax)
- `let` is an expression
  - `g x = 1 + (let y = 2+x in y^2) + 3`
  - `h x = let p y = 2*y+32 in p x + p 5`

Sharing

- `slow (x-1)` is calculated twice:

```
slow 0 = 1
slow x = slow (x-1) + slow (x-1)
```

Sharing

- `slow (x-1)` is calculated twice:

```
slow 0 = 1
slow x = slow (x-1) + slow (x-1)
```

- `fast (x-1)` is calculated once (sharing):

```
fast 0 = 1
fast x = y + y
  where y = fast (x-1)
```

Sharing

- `slow (x-1)` is calculated twice:

```
slow 0 = 1
slow x = slow (x-1) + slow (x-1)
```

- `fast (x-1)` is calculated once (sharing):

```
fast 0 = 1
fast x = y + y
  where y = fast (x-1)
```

- `fast' (x-1)` is calculated once (sharing):

```
fast' 0 = 1
fast' x = let y = fast' (x-1) in y + y
```

List comprehension: let

```
f xs ys = [ x + y | x <- xs, y <- ys, x + y <= 1 ]
```

List comprehension: let

```
f xs ys = [ x + y | x <- xs, y <- ys, x + y <= 1 ]
```

To avoid calculating `x + y` twice:

```
f xs ys = [ z | x <- xs, y <- ys, let z = x + y, z <= 1 ]
```

List comprehension: pattern matching and “as” expressions

```
f :: [[Char]] -> [[Char]]
f xss = [ xs | xs <- xss, isUpper $ head xs ]
```

List comprehension: pattern matching and “as” expressions

```
f :: [[Char]] -> [[Char]]
f xss = [ xs | xs <- xss, isUpper $ head xs ]
```

Pattern matching:

```
g xss = [ x:xs | (x:xs) <- xss, isUpper x ]
```

List comprehension: pattern matching and “as” expressions

```
f :: [[Char]] -> [[Char]]
f xss = [ xs | xs <- xss, isUpper $ head xs ]
```

Pattern matching:

```
g xss = [ x:xs | (x:xs) <- xss, isUpper x ]
```

As-patterns

```
h xss = [ r | r@(x:xs) <- xss, isUpper x ]
```

List comprehension: pattern matching and “as” expressions

```
f :: [[Char]] -> [[Char]]
f xss = [ xs | xs <- xss, isUpper $ head xs ]
```

Pattern matching:

```
g xss = [ x:xs | (x:xs) <- xss, isUpper x ]
```

As-patterns

```
h xss = [ r | r@(x:xs) <- xss, isUpper x ]
```

Are `f` and `h` equivalent?

List comprehension: pattern matching and “as” expressions

```
f :: [[Char]] -> [[Char]]
f xss = [ xs | xs <- xss, isUpper $ head xs ]
```

Pattern matching:

```
g xss = [ x:xs | (x:xs) <- xss, isUpper x ]
```

As-patterns

```
h xss = [ r | r@(x:xs) <- xss, isUpper x ]
```

Are `f` and `h` equivalent?

```
f ["John", "apple", ""]
h ["John", "apple", ""]
```

Reminder: Compound types

- Lists: `[a]`

Reminder: Compound types

- Lists: `[a]`
- Tuples:
  - `()`
  - `(a,b)`
  - `(a,b,c)`
  - `(a,b,c,d)`
  - ...

Type synonyms

- Give another name to an *existing* type

```
type String = [Char]
```

Type synonyms

- Give another name to an *existing* type

```
type String = [Char]
```

- `String` and `[Char]` are now interchangeable:

```
firstname :: [Char]
firstname = "Ann"

lastname :: String
lastname = "Steward"

fullname = firstname ++ " " ++ lastname
```

Algebraic datatypes

```
data Bool = True | False
```

Algebraic datatypes

```
data Bool = True | False

not :: Bool -> Bool
not True  = False
not False = True
```

Algebraic datatypes

```
data Bool = True | False

not :: Bool -> Bool
not True  = False
not False = True

data Day = Monday
         | Tuesday
         | Wednesday
        ...
       deriving (Show, Ord)
```

Algebraic datatypes

```
data Bool = True | False

not :: Bool -> Bool
not True  = False
not False = True

data Day = Monday
         | Tuesday
         | Wednesday
        ...
       deriving (Show, Ord)

isweekend :: Day -> Bool
isweekend Saturday = True
isweekend Sunday   = True
isweekend _        = False
```

Data constructors

```
data Shape = Rectangle Double Double -- width and height
           | Circle Double           -- radius

unitcircle :: Shape
unitcircle =  Circle  1.0
square     =  Rectangle  1.0 1.0

circumfence :: Shape -> Double
circumfence ( Rectangle  w h) = 2 * w + 2 * h
circumfence ( Circle  r)      = 2 * pi * r

area :: Shape -> Double
area ( Rectangle  w h) = w * h
area ( Circle  r)      = pi * r^2
```

Data constructors

```
data Shape = Rectangle Double Double -- width and height
           | Circle Double           -- radius

unitcircle :: Shape
unitcircle = Circle 1.0
square     = Rectangle 1.0 1.0

circumfence :: Shape -> Double
circumfence (Rectangle w h) = 2 * w + 2 * h
circumfence (Circle r)      = 2 * pi * r

area :: Shape -> Double
area (Rectangle w h) = w * h
area (Circle r)      = pi * r^2
```

Record syntax

- Data constructors:

```
data Person = Person String String Int
                deriving Show

jan :: Person
jan = Person "Jan" "Janssen" 42

firstname (Person x _ _) = x
lastname  (Person _ x _) = x
age       (Person _ _ x) = x
```

Record syntax

- Data constructors:

```
data Person = Person String String Int
              deriving Show

jan :: Person
jan = Person "Jan" "Janssen" 42

firstname (Person x _ _) = x
lastname  (Person _ x _) = x
age       (Person _ _ x) = x
```

- Record syntax:

```
data Person = Person { firstname:: String,
                       lastname :: String,
                       age:: Int
                     } deriving Show
jan = Person "Jan" "Janssen" 42
```

Record update

```
data Person = Person { firstname:: String,
                       lastname :: String,
                       age:: Int
                     } deriving Show

jan = Person "Jan" "Janssen" 42
jan' = Person {firstname="Jan",lastname="Janssen",age=42}
```

Record update

```
data Person = Person { firstname:: String,
                       lastname :: String,
                       age:: Int
                     } deriving Show

jan = Person "Jan" "Janssen" 42
jan' = Person {firstname="Jan",lastname="Janssen",age=42}

jan'' = jan { lastname = "Jansen" }
```

Record update

```
data Person = Person { firstname:: String,
                       lastname :: String,
                       age:: Int
                     } deriving Show

jan = Person "Jan" "Janssen" 42
jan' = Person {firstname="Jan",lastname="Janssen",age=42}

jan'' = jan { lastname = "Jansen" }
```

```
*Lec3> jan
Person {firstname = "Jan", lastname = "Janssen", age = 42}
*Lec3> lastname jan"
"Jansen"
```

Record update

```
data Person = Person { firstname:: String,
                       lastname :: String,
                       age:: Int
                     } deriving Show

jan = Person "Jan" "Janssen" 42
jan' = Person {firstname="Jan",lastname="Janssen",age=42}

jan'' = jan { lastname = "Jansen" }
```

```
*Lec3> jan
Person {firstname = "Jan", lastname = "Janssen", age = 42}
*Lec3> lastname jan"
"Jansen"
```

Does not change a record, but creates a new (updated) record

Type constructors

- `Maybe` type (Haskell Prelude):

```
data Maybe a = Nothing | Just a
```

Type constructors

- `Maybe` type (Haskell Prelude):

```
data Maybe a = Nothing | Just a
```

Type constructors

- `Maybe` type (Haskell Prelude):

```
data Maybe a = Nothing | Just a
```

- Examples:

```
Prelude Data.Maybe> :t Nothing
Nothing ::  Maybe a
Prelude Data.Maybe> fromJust (Just "abc")
"abc"
```

Type constructors

- `Maybe` type (Haskell Prelude):

```
data Maybe a = Nothing | Just a
```

- Examples:

```
Prelude Data.Maybe> :t Nothing
Nothing ::  Maybe a
Prelude Data.Maybe> fromJust (Just "abc")
"abc"
```

- Error handling:

```
Prelude Data.List> find even [1,3,7]
Nothing
Prelude Data.List> find even [1,3,22,7,200]
Just 22
```

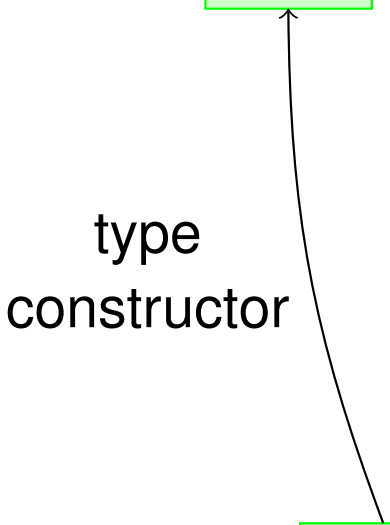
Type constructors and data constructors

```
data  Pair  a b =  Pair  a b
```

```
pfst ::  Pair  a b -> a
pfst   ( Pair  a b) = a
```

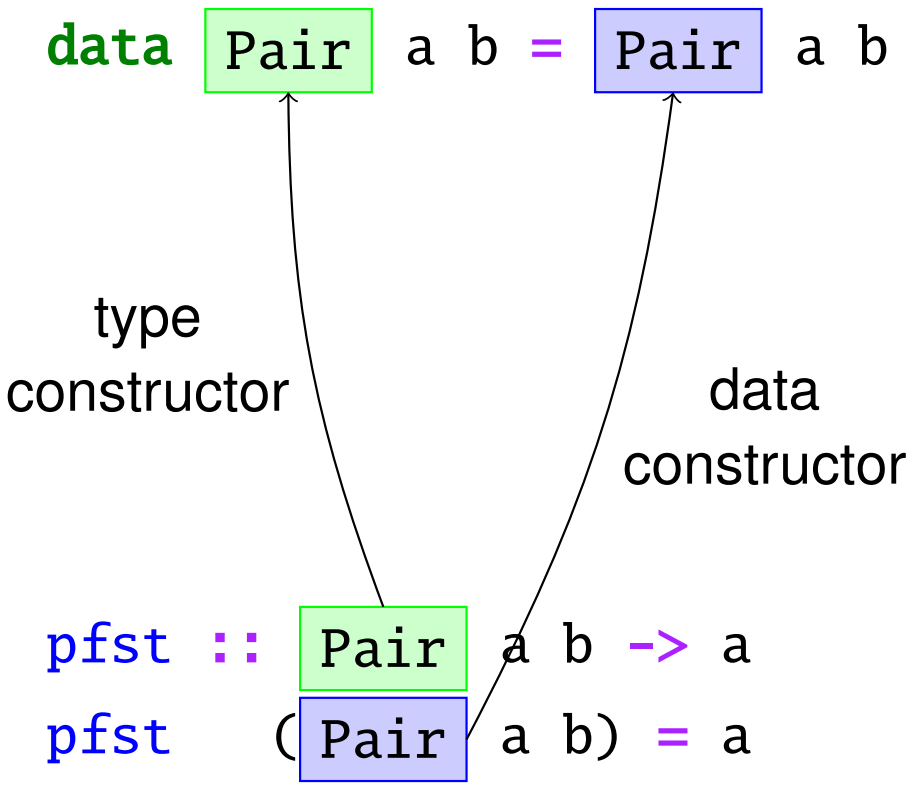
Type constructors and data constructors

```
data Pair a b = Pair a b
```




```
pfst :: Pair a b -> a
pfst   (Pair a b) = a
```

Type constructors and data constructors



Either

Either definition (“union-like type” - Haskell Prelude)

```
data Either a b = Left a | Right b
```

Either

Either definition (“union-like type” - Haskell Prelude)

```
data Either a b = Left a | Right b
```

- ```
  safeDiv :: Integral r => r -> r -> Either [Char] r
  safeDiv x 0 = Left "divide by zero"
  safeDiv x y = Right $ x `div` y
  ```
  ```

Either

Either definition (“union-like type” - Haskell Prelude)

```
data Either a b = Left a | Right b
```

- ```
  safeDiv :: Integral r => r -> r -> Either [Char] r
  safeDiv x 0 = Left "divide by zero"
  safeDiv x y = Right $ x `div` y
  ```
- Example `10 `safeDiv` 2 == Right 5`
  ```

Tuples revisited

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

Tuples revisited

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

- Type constructor `(,)`

  ```
  fst :: (,) a b -> a
  ```

  `(a,b)` is short hand notation for `(,) a b`

Tuples revisited

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

- Type constructor `(,)`

  ```
  fst :: (,) a b -> a
  ```

  `(a,b)` is short hand notation for `(,) a b`

- Data constructor `(,)`

  ```
  x = (,) 1 2
  ```

  `(1,2)` is short hand notation for `(,) 1 2`

Tuples revisited

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

- Type constructor `(,)`

  ```
  fst :: (,) a b -> a
  ```

  `(a,b)` is short hand notation for `(,) a b`

- Data constructor `(,)`

  ```
  x = (,) 1 2
  ```

  `(1,2)` is short hand notation for `(,) 1 2`

- Definition (type `:i (,)` in GHCi!)

  ```
  data (,) a b = (,) a b  -- Defined in `GHC.Tuple'
  ```

Tuples revisited

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

- Type constructor `(,)`

```
fst :: (,) a b -> a
```

`(a,b)` is short hand notation for `(,) a b`

- Data constructor `(,)`

```
x = (,) 1 2
```

`(1,2)` is short hand notation for `(,) 1 2`

- Definition (type `:i (,)` in GHCi!)

```
data (,) a b = (,) a b  -- Defined in `GHC.Tuple'
```

Lists revisited

```
head :: [a] -> a
head (x:xs) = x
```

Lists revisited

```
head :: [a] -> a
head (x:xs) = x
```

- Type constructor `[]`

```
head ::  []  a -> a
```

  - `[a]` is short hand notation for `[] a`
  - This `[]` is not the empty list!

Lists revisited

```
head :: [a] -> a
head (x:xs) = x
```

- Type constructor `[]`

```
head ::  []  a -> a
```

  - `[a]` is short hand notation for `[] a`
  - This `[]` is not the empty list!

- Data constructors: `[]` and `(:)`

```
(:)  1 ( (:)  2  [] ) == [1,2]
```

Infix notation `[1,2]` is short hand notation for `1:2:[]`

Lists revisited

```
head :: [a] -> a
head (x:xs) = x
```

- Type constructor `[]`

```
head ::  []  a -> a
```

  - `[a]` is short hand notation for `[] a`
  - This `[]` is not the empty list!

- Data constructors: `[]` and `(:)`

```
(:)  1 ( (:)  2  [] ) == [1,2]
```

Infix notation `[1,2]` is short hand notation for `1:2:[]`

- Definition:

```
data  []  a =  []  | a  :  [a] -- Defined in `GHC.Types'
```

Lists revisited

```
head :: [a] -> a
head (x:xs) = x
```

- Type constructor `[]`

  ```
  head :: [] a -> a
  ```

  - `[a]` is short hand notation for `[] a`
  - This `[]` is not the empty list!

- Data constructors: `[]` and `(:)`

  `(:) 1 ((:) 2 []) == [1,2]`

  Infix notation `[1,2]` is short hand notation for `1:2:[]`

- Definition:

  ```
  data [] a = [] | a : [a] -- Defined in `GHC.Types'
  ```

Kinds

- Type constructor `[]` receives a type as argument

Kinds

- Type constructor `[]` receives a type as argument
- Make `[] a` concrete: supply a type for `a`
  - `[] Int`, `[Int]`, `[String]`, `[[Int]]`, `[(Int, Char)]`

Kinds

- Type constructor `[]` receives a type as argument
- Make `[] a` concrete: supply a type for `a`
  - `[] Int`, `[Int]`, `[String]`, `[[Int]]`, `[(Int, Char)]`
- Kinds: the level above types:
  - Values are of a Type
  - Types are of a Kind

Kinds

- Type constructor `[]` receives a type as argument
- Make `[] a` concrete: supply a type for `a`
  - `[] Int`, `[Int]`, `[String]`, `[[Int]]`, `[(Int, Char)]`
- Kinds: the level above types:
  - Values are of a Type
  - Types are of a Kind
- Kind for lists (`:k` gives the kind)

```
Prelude> :k []
[] ::  * -> *
Prelude> :k [] Int
[] Int ::  *
```

Kinds

- Type constructor `[]` receives a type as argument
- Make `[] a` concrete: supply a type for `a`
  - `[] Int`, `[Int]`, `[String]`, `[[Int]]`, `[(Int, Char)]`
- Kinds: the level above types:
  - Values are of a Type
  - Types are of a Kind
- Kind for lists (`:k` gives the kind)

```
Prelude> :k []
[] ::  * -> *
Prelude> :k [] Int
[] Int ::  *
```

- Nullary: * is any type constructor that takes no parameters
  - `Int, String, [Int], (Int, Char)`

Kinds

- Type constructor `[]` receives a type as argument
- Make `[] a` concrete: supply a type for `a`
  - `[] Int`, `[Int]`, `[String]`, `[[Int]]`, `[(Int, Char)]`
- Kinds: the level above types:
  - Values are of a Type
  - Types are of a Kind
- Kind for lists (`:k` gives the kind)

```
Prelude> :k []
[] ::  * -> *
Prelude> :k [] Int
[] Int ::  *
```

- Nullary: `*` is any type constructor that takes no parameters
  - `Int`, `String`, `[Int]`, `(Int, Char)`
- `* -> *`: kind that produces a nullary type from a nullary type

MyList

```
data MyList a = Nil | Cons a (MyList a)
```

MyList

```haskell
data MyList a = Nil | Cons a (MyList a)

myhead :: MyList a -> a
myhead (Cons x xs) = x

mytail :: MyList a -> MyList a
mytail (Cons x xs) = xs
```

MyList

```haskell
data MyList a = Nil | Cons a (MyList a)

myhead :: MyList a -> a
myhead (Cons x xs) = x

mytail :: MyList a -> MyList a
mytail (Cons x xs) = xs

mymap :: (a -> b) -> MyList a -> MyList b
mymap f Nil          = Nil
mymap f (Cons x xs) = (f x) `Cons` mymap f xs
```

MyList

```
data MyList a = Nil | Cons a (MyList a)

myhead :: MyList a -> a
myhead (Cons x xs) = x

mytail :: MyList a -> MyList a
mytail (Cons x xs) = xs

mymap :: (a -> b) -> MyList a -> MyList b
mymap f Nil          = Nil
mymap f (Cons x xs) = (f x) `Cons` mymap f xs
```

Example:

```
list   =

list'  =

mylist =
```

MyList

```
data MyList a = Nil | Cons a (MyList a)

myhead :: MyList a -> a
myhead (Cons x xs) = x

mytail :: MyList a -> MyList a
mytail (Cons x xs) = xs

mymap :: (a -> b) -> MyList a -> MyList b
mymap f Nil          = Nil
mymap f (Cons x xs) = (f x) `Cons` mymap f xs
```

Example:

```
list   = [                                    ]

list'  =                                      []

mylist =                                      Nil
```

MyList

```
data MyList a = Nil | Cons a (MyList a)

myhead :: MyList a -> a
myhead (Cons x xs) = x

mytail :: MyList a -> MyList a
mytail (Cons x xs) = xs

mymap :: (a -> b) -> MyList a -> MyList b
mymap f Nil          = Nil
mymap f (Cons x xs) = (f x) `Cons` mymap f xs
```

Example:

```
list   = [                    3          ]
list'  =                      3 :        []
mylist =                      3 `Cons`   Nil
```

MyList

```
data MyList a = Nil | Cons a (MyList a)

myhead :: MyList a -> a
myhead (Cons x xs) = x

mytail :: MyList a -> MyList a
mytail (Cons x xs) = xs

mymap :: (a -> b) -> MyList a -> MyList b
mymap f Nil          = Nil
mymap f (Cons x xs) = (f x) `Cons` mymap f xs
```

Example:

```
list   = [           2 ,        3        ]
list'  =             2 :        3 :      []
mylist =             2 `Cons` ( 3 `Cons` Nil)
```

MyList

```
data MyList a = Nil | Cons a (MyList a)

myhead :: MyList a -> a
myhead (Cons x xs) = x

mytail :: MyList a -> MyList a
mytail (Cons x xs) = xs

mymap :: (a -> b) -> MyList a -> MyList b
mymap f Nil          = Nil
mymap f (Cons x xs) = (f x) `Cons` mymap f xs
```

Example:

```
list   = [ 1 ,        2 ,        3        ]
list'  =   1 :        2 :        3 :      []
mylist =   1 `Cons` ( 2 `Cons` ( 3 `Cons` Nil))
```

Most important type/data constructors

Type	Type constructor	Data constructor(s)
Pair	`(,) a b`	`(,) :: a -> b -> (,) a b`

Most important type/data constructors

Type	Type constructor	Data constructor(s)
Pair	`(,) a b`	`(,) :: a -> b -> (,) a b`
Triple	`(,,) a b c`	`(,,) :: a -> b -> c -> („) a b c`

Most important type/data constructors

Type	Type constructor	Data constructor(s)
Pair	`(,) a b`	`(,) :: a -> b -> (,) a b`
Triple	`(,,) a b c`	`(,,) :: a -> b -> c -> („) a b c`
List	`[] a`	`[] :: a -> [] a` *and* `(:) :: a -> [] a -> [] a`

Most important type/data constructors

Type	Type constructor	Data constructor(s)	
Pair	`(,) a b`	`(,) :: a -> b -> (,) a b`	
Triple	`(,,) a b c`	`(,,) :: a -> b -> c -> („) a b c`	
List	`[] a`	`[] :: a -> [] a` `(:) :: a -> [] a -> [] a`	*and*
Boolean	`Bool`	`True` `False`	*and*

Most important type/data constructors

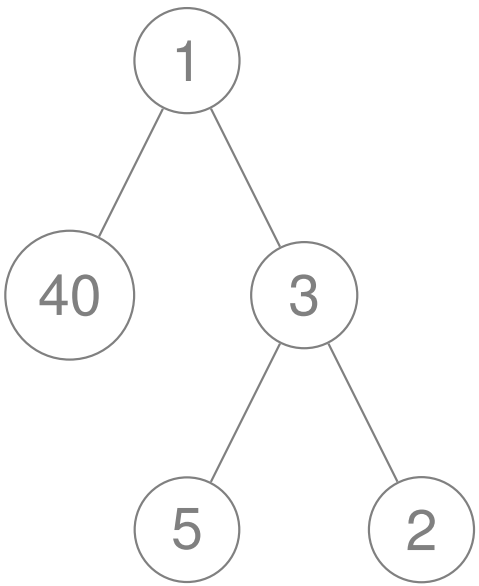
Type	Type constructor	Data constructor(s)	
Pair	`(,) a b`	`(,) :: a -> b -> (,) a b`	
Triple	`(,,) a b c`	`(,,) :: a -> b -> c -> („) a b c`	
List	`[] a`	`[] :: a -> [] a` `(:) :: a -> [] a -> [] a`	*and*
Boolean	`Bool`	`True` `False`	*and*
Maybe	`Maybe a`	`Just :: a -> Maybe a` `Nothing :: Maybe a`	*and*

Most important type/data constructors

Type	Type constructor	Data constructor(s)	
Pair	`(,) a b`	`(,) :: a -> b -> (,) a b`	
Triple	`(,,) a b c`	`(,,) :: a -> b -> c -> („) a b c`	
List	`[] a`	`[] :: a -> [] a` `(:) :: a -> [] a -> [] a`	*and*
Boolean	`Bool`	`True` `False`	*and*
Maybe	`Maybe a`	`Just :: a -> Maybe a` `Nothing :: Maybe a`	*and*
Either	`Either a b`	`Left :: a -> Either a b` `Right :: b -> Either a b`	*and*

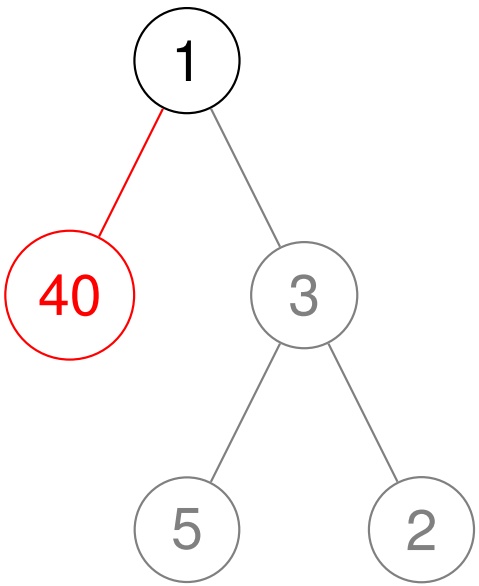
Trees

```
data IntTree = Leaf Int
             | Node Int IntTree IntTree
```



Trees

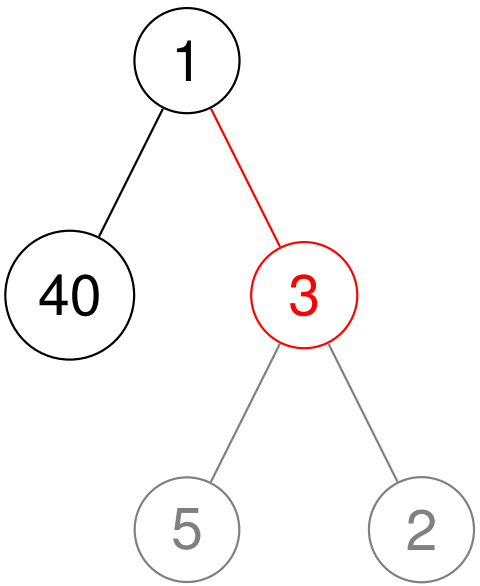
```
data IntTree = Leaf Int
             | Node Int IntTree IntTree
```




```
t :: IntTree
t = Node 1 (...)
           (...)
```

Trees

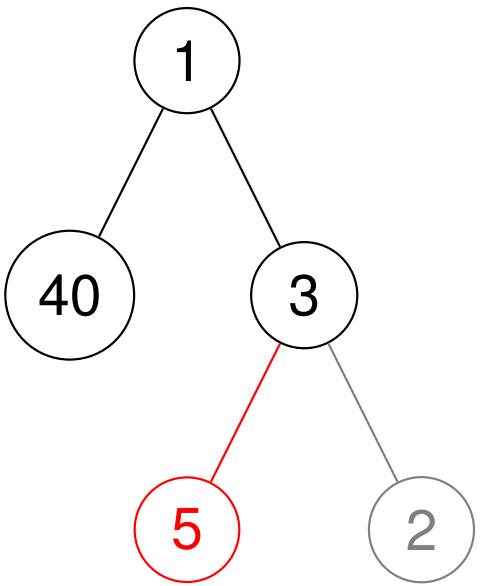
```
data IntTree = Leaf Int
             | Node Int IntTree IntTree
```




```
t :: IntTree
t = Node 1 (Leaf 40)
           (...)
```

Trees

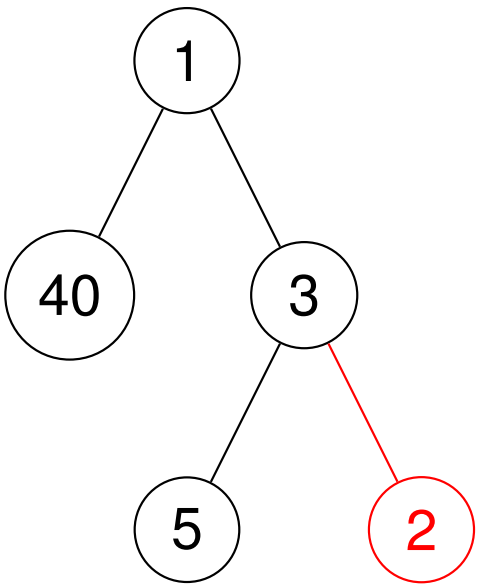
```
data IntTree = Leaf Int
             | Node Int IntTree IntTree
```




```
t :: IntTree
t = Node 1 (Leaf 40)
           (Node 3 (...)
                   (...))
```

Trees

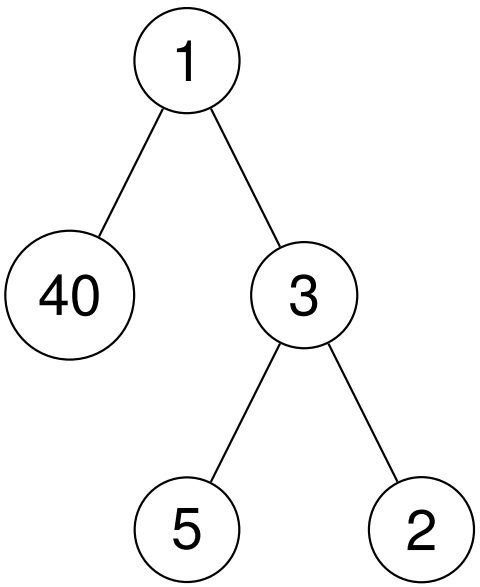
```
data IntTree = Leaf Int
             | Node Int IntTree IntTree
```




```
t :: IntTree
t = Node 1 (Leaf 40)
           (Node 3 (Leaf 5)
                   (...))
```

Trees

```
data IntTree = Leaf Int
             | Node Int IntTree IntTree
```




```
t :: IntTree
t = Node 1 (Leaf 40)
           (Node 3 (Leaf 5)
                   (Leaf 2))
```

Overloading

```
data Binary = Zero | One

show Zero = "0"
show One  = "1"
```

Overloading

```
data Binary = Zero | One

show Zero = "0"
show One  = "1"
```

Not allowed: `show` is already defined

Typeclasses

Definition of a typeclass:

```
class Show a where      -- from GHC.Show
  show :: a -> String
  ...
```

Typeclasses

Definition of a typeclass:

```
class Show a where          -- from GHC.Show
  show :: a -> String
  ...
```

Instance of a typeclass:

```
data Binary = Zero | One

instance Show Binary where
  show Zero = "0"
  show One  = "1"
```

Typeclasses

Definition of a typeclass:

```
class Show a where          -- from GHC.Show
  show :: a -> String
  ...
```

Instance of a typeclass:

```
data Binary = Zero | One

instance Show Binary where
  show Zero = "0"
  show One  = "1"

show [Zero, Zero, One, Zero] == "0010"
```

Overloading equality (1/2)

Define equality for `MyList`:

```
Cons x l1 === Cons y l2  = x == y && l1 === l2
Nil       === Nil        = True
_         === _          = False
```

Overloading equality (1/2)

Define equality for `MyList`:

```
Cons x l1 === Cons y l2  = x == y && l1 === l2
Nil       === Nil        = True
_         === _          = False
```

Nicer, but not allowed ((==) is already defined):

```
Nil       == Nil         = True
Cons x l1 == Cons y l2   = x == y && l1 == l2
_         == _           = False
```

Overloading equality (1/2)

Define equality for `MyList`:

```
Cons x l1 === Cons y l2  = x == y && l1 === l2
Nil       === Nil        = True
_         === _          = False
```

Nicer, but not allowed ((==) is already defined):

```
Nil       == Nil         = True
Cons x l1 == Cons y l2   = x == y && l1 == l2
_         == _           = False
```

Overloading equality (2/2)

- Definition of typeclass `Eq`

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

Overloading equality (2/2)

- Definition of typeclass `Eq`

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

- Instances must be of kind: `a :: *`

Overloading equality (2/2)

- Definition of typeclass `Eq`

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

- Instances must be of kind: `a :: *`
- Instance for `Mylist a` of `Eq`:

```
instance Eq a => Eq (MyList a) where
  Nil        == Nil         = True
  Cons x l1 == Cons y l2    = x == y && l1 == l2
  _          == _           = False
```

Overloading equality (2/2)

- Definition of typeclass `Eq`

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

- Instances must be of kind: `a :: *`
- Instance for `Mylist a` of `Eq`:

```
instance Eq a => Eq (MyList a) where
  Nil        == Nil         = True
  Cons x l1 == Cons y l2    = x == y && l1 == l2
  _          == _           = False
```

- Kind: `MyList a :: *`

Kind errors

Incorrect code:

```
instance Eq MyList where
  Nil        == Nil      = True
  Cons x l1 == Cons y l2  = l1 == l2
  _          == _        = False
```

Kind errors

Incorrect code:

```
instance Eq MyList where
  Nil        == Nil      = True
  Cons x l1 == Cons y l2  = l1 == l2
  _          == _        = False
```

```
Lec3.hs:xx:yy:
Expecting one more argument to ‘MyList’
The first argument of ‘Eq’ should have kind ‘*’,
but ‘MyList’ has kind ‘* -> *’
In the instance declaration for ‘Eq (MyList)’
Failed, modules loaded:  none.
```

Kind errors

Incorrect code:

```
instance Eq MyList where
  Nil        == Nil      = True
  Cons x l1 == Cons y l2  = l1 == l2
  _          == _        = False
```

```
Lec3.hs:xx:yy:
Expecting one more argument to ‘MyList’
The first argument of ‘Eq’ should have kind ‘*’,
but ‘MyList’ has kind ‘* -> *’
In the instance declaration for ‘Eq (MyList)’
Failed, modules loaded:  none.
```

Kind errors

Incorrect code:

```
instance Eq MyList where
  Nil        == Nil      = True
  Cons x l1 == Cons y l2  = l1 == l2
  _          == _        = False
```

```
Lec3.hs:xx:yy:
Expecting one more argument to ‘MyList’
The first argument of ‘Eq’ should have kind ‘*’,
but ‘MyList’ has kind ‘* -> *’
In the instance declaration for ‘Eq (MyList)’
Failed, modules loaded:  none.
```

Generalizing map: Functors

- Class definition of functor:

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

Generalizing map: Functors

- Class definition of functor:

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

- `f` is a type constructor of kind: `* -> *`
- Use `:i Functor` in GHCi to see the kind

Generalizing map: Functors

- Class definition of functor:

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

- `f` is a type constructor of kind: `* -> *`
- Use `:i Functor` in GHCi to see the kind
- Exampes of types with kind `* -> *`

Type	**Type constructor (`f`)**	Concrete type (**`f b`**)
Lists	`[]`	`[] b` (or equivalent `[b]`)
Maybe	`Maybe`	`Maybe b`
Either	`Either a`	`Either a b`
Tuple	`(,) a`	`(,) a b` (or equivalent `(a,b)`)

Generalizing map: Functors

- Class definition of functor:

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

- `f` is a type constructor of kind: `* -> *`
- Use `:i Functor` in GHCi to see the kind
- Exampes of types with kind `* -> *`

Type	**Type constructor (`f`)**	Concrete type (**`f b`**)
Lists	`[]`	`[] b` (or equivalent `[b]`)
Maybe	`Maybe`	`Maybe b`
Either	`Either a`	`Either a b`
Tuple	`(,) a`	`(,) a b` (or equivalent `(a,b)`)

- Intuition: “boxes” you can put *any* `b` type in

List functor

- Class definition of functor:

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

List functor

- Class definition of functor:

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

- What is the type constructor `f` for lists?

List functor

- Class definition of functor:

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

- What is the type constructor `f` for lists?
  - `[]` is of kind `* -> *`
  - Example: `[] Int` is `[Int]`

List functor

- Class definition of functor:

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

- What is the type constructor `f` for lists?
  - `[]` is of kind `* -> *`
  - Example: `[] Int` is `[Int]`
  - Intuition: replace `f` by `[]` in the `fmap` type:

```
fmap :: (a -> b) -> [a] -> [b]
```

List functor

- Class definition of functor:

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

- What is the type constructor `f` for lists?
  - `[]` is of kind `* -> *`
  - Example: `[] Int` is `[Int]`
  - Intuition: replace `f` by `[]` in the `fmap` type:

```
fmap :: (a -> b) -> [a] -> [b]
```

- Instance:

```
instance Functor [] where
  fmap = map
```

Tuple functor

- Class definition of functor:

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

Tuple functor

- Class definition of functor:

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

- What is the type constructor `f` for tuples?

Tuple functor

- Class definition of functor:

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

- What is the type constructor `f` for tuples?
  - `(,)` is of kind `* -> * -> *`

Tuple functor

- Class definition of functor:

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

- What is the type constructor `f` for tuples?
  - `(,)` is of kind `* -> * -> *`
  - `(,) a` is of kind `* -> *`

Tuple functor

- Class definition of functor:

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

- What is the type constructor `f` for tuples?
  - `(,)` is of kind `* -> * -> *`
  - `(,) a` is of kind `* -> *`
- Functor instance:

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

Tuple functor

- Class definition of functor:

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

- What is the type constructor `f` for tuples?
  - `(,)` is of kind `* -> * -> *`
  - `(,) a` is of kind `* -> *`
- Functor instance:

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

- Why is the following instance wrong?

```
instance Functor ((,) a) where
  fmap f (a, b) = (f a, f b)
```

IO in Haskell (1/2)

- IO Actions
  - `IO a` : IO Action that results in type `a`
  - IO Actions describe the IO steps to be taken
    - Think of it as a datastructure that describes the actions
  - *pure* Haskell functions can generate IO actions

IO in Haskell (1/2)

- IO Actions
  - `IO a` : IO Action that results in type `a`
  - IO Actions describe the IO steps to be taken
    - Think of it as a datastructure that describes the actions
  - *pure* Haskell functions can generate IO actions
- Executing IO actions
  - Interpreter: executes IO Actions, shows result
  - Compiler: define `main :: IO ()` in the module `Main`

IO in Haskell (1/2)

- IO Actions
  - `IO a` : IO Action that results in type `a`
  - IO Actions describe the IO steps to be taken
    - Think of it as a datastructure that describes the actions
  - *pure* Haskell functions can generate IO actions
- Executing IO actions
  - Interpreter: executes IO Actions, shows result
  - Compiler: define `main :: IO ()` in the module `Main`
- Example `putStrLn :: String -> IO ()`
  - *pure* function that maps a `String` to an IO Action that shows the `String`
  - Example:

```
Prelude> hello = putStrLn "Hello, world!"
Prelude> :t hello
hello ::  IO ()
Prelude> hello
Hello, world!
```

IO in Haskell (2/2)

- Reading input with `getLine :: IO String`

```
Prelude> x = getLine
Prelude> :t x
x ::  IO String
Prelude> x
abc
"abc"
```

IO in Haskell (2/2)

- Reading input with `getLine :: IO String`

```
Prelude> x = getLine
Prelude> :t x
x ::  IO String
Prelude> x
abc
"abc"
```

- IO Functor: map over the IO result

```
Prelude> y = fmap length x
Prelude> :t y
y ::  IO Int
Prelude> y
abc
3
```

IO in Haskell (2/2)

- Reading input with `getLine :: IO String`

```
Prelude> x = getLine
Prelude> :t x
x ::  IO String
Prelude> x
abc
"abc"
```

- IO Functor: map over the IO result

```
Prelude> y = fmap length x
Prelude> :t y
y ::  IO Int
Prelude> y
abc
3
```

- The Functor transforms the IO action!

Functor laws

- `fmap id = id`
  Intuition: mapping the `id` function (`id x = x`) over a Functor ("box") does not change anything to the Functor ("box").

Functor laws

- `fmap id = id`
  Intuition: mapping the `id` function (`id x = x`) over a Functor ("box") does not change anything to the Functor ("box").
- `fmap (f . g) = fmap f . fmap g`
  Intuition: mapping the combined function `h x = f (g x)` over a Functor gives the same as mapping the individual functions

Functor laws

- `fmap id = id`
  Intuition: mapping the `id` function (`id x = x`) over a Functor ("box") does not change anything to the Functor ("box").
- `fmap (f . g) = fmap f . fmap g`
  Intuition: mapping the combined function `h x = f (g x)` over a Functor gives the same as mapping the individual functions

- Why is the following instance wrong?

```
instance Functor [] where
  fmap f xs = (reverse . map f) xs
```

Proving the laws: tuples

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

Proving the laws: tuples

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

First law: Prove `fmap id = id`

Proving the laws: tuples

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

First law: Prove `fmap id = id`

This means: `fmap id (a,b) = id (a,b)`

Proving the laws: tuples

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

First law: Prove `fmap id = id`

This means: `fmap id (a,b) = id (a,b)`

```
fmap id (a, b) = (a, id b) -- use: fmap
```

Proving the laws: tuples

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

First law: Prove `fmap id = id`

This means: `fmap id (a,b) = id (a,b)`

```
fmap id (a, b) = (a, id b)  -- use: fmap
               = (a, b)     -- use: id x = x
```

Proving the laws: tuples

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

First law: Prove `fmap id = id`

This means: `fmap id (a,b) = id (a,b)`

```
fmap id (a, b) = (a, id b)  -- use: fmap
               = (a, b)     -- use: id x = x
               = id (a, b)  -- use: id x = x
```

Proving the laws: tuples

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

First law: Prove `fmap id = id`

This means: `fmap id (a,b) = id (a,b)`

```
fmap id (a, b) = (a, id b)  -- use: fmap
               = (a, b)     -- use: id x = x
               = id (a, b)  -- use: id x = x
```

Second law: Prove `fmap (f . g) = fmap f . fmap g`

Proving the laws: tuples

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

First law: Prove `fmap id = id`

This means: `fmap id (a,b) = id (a,b)`

```
fmap id (a, b) = (a, id b)  -- use: fmap
               = (a, b)     -- use: id x = x
               = id (a, b)  -- use: id x = x
```

Second law: Prove `fmap (f . g) = fmap f . fmap g`

```
fmap (f.g) (a,b) = (a, (f . g) b)          -- use fmap
```

Proving the laws: tuples

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

First law: Prove `fmap id = id`

This means: `fmap id (a,b) = id (a,b)`

```
fmap id (a, b) = (a, id b)  -- use: fmap
               = (a, b)     -- use: id x = x
               = id (a, b)  -- use: id x = x
```

Second law: Prove `fmap (f . g) = fmap f . fmap g`

```
fmap (f.g) (a,b) = (a, (f . g) b)        -- use fmap
                 = (a, f (g b))          -- (f.g) x = f (g x)
```

Proving the laws: tuples

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

First law: Prove `fmap id = id`

This means: `fmap id (a,b) = id (a,b)`

```
 fmap id (a, b) = (a, id b)  -- use: fmap
                = (a, b)     -- use: id x = x
                = id (a, b)  -- use: id x = x
```

Second law: Prove `fmap (f . g) = fmap f . fmap g`

```
fmap (f.g) (a,b) = (a, (f . g) b)          -- use fmap
                 = (a, f (g b))            -- (f.g) x = f (g x)
                 = fmap f (a, g b)         -- use fmap
```

Proving the laws: tuples

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

First law: Prove `fmap id = id`

This means: `fmap id (a,b) = id (a,b)`

```
fmap id (a, b) = (a, id b)  -- use: fmap
               = (a, b)     -- use: id x = x
               = id (a, b)  -- use: id x = x
```

Second law: Prove `fmap (f . g) = fmap f . fmap g`

```
fmap (f.g) (a,b) = (a, (f . g) b)          -- use fmap
                 = (a, f (g b))            -- (f.g) x = f (g x)
                 = fmap f (a, g b)         -- use fmap
                 = fmap f (fmap g (a, b))  -- use fmap
```

Proving the laws: tuples

```
instance Functor ((,) a) where
  fmap f (a, b) = (a, f b)
```

First law: Prove `fmap id = id`

This means: `fmap id (a,b) = id (a,b)`

```
fmap id (a, b) = (a, id b)  -- use: fmap
               = (a, b)     -- use: id x = x
               = id (a, b)  -- use: id x = x
```

Second law: Prove `fmap (f . g) = fmap f . fmap g`

```
fmap (f.g) (a,b) = (a, (f . g) b)          -- use fmap
                 = (a, f (g b))            -- (f.g) x = f (g x)
                 = fmap f (a, g b)         -- use fmap
                 = fmap f (fmap g (a, b))  -- use fmap
                 = (fmap f . fmap g) (a, b) -- (f.g) x = f (g x)
```

Examples of functors

Type	Type constructor	Expression	Result
pairs	`(,) a b`	fmap (+10) ("abc", 1)	("abc", 11)

Examples of functors

Type	Type constructor	Expression	Result
pairs	`(,) a b`	fmap (+10) ("abc", 1)	("abc", 11)
lists	`[] b`	fmap (+10) [1,2,3]	[11,12,13]

Examples of functors

Type	Type constructor	Expression	Result
pairs	`(,) a b`	fmap (+10) ("abc", 1)	("abc", 11)
lists	`[] b`	fmap (+10) [1,2,3]	[11,12,13]
Maybe	`Maybe b`	fmap (+10) (Just 1)	Just 11
		fmap (+10) Nothing	Nothing

Examples of functors

Type	Type constructor	Expression	Result
pairs	`(,) a b`	fmap (+10) ("abc", 1)	("abc", 11)
lists	`[] b`	fmap (+10) [1,2,3]	[11,12,13]
Maybe	`Maybe b`	fmap (+10) (Just 1)	Just 11
		fmap (+10) Nothing	Nothing
Either	`Either a b`	fmap (+10) (Left 1)	Left 1
		fmap (+10) (Right 1)	Right 11

Examples of functors

Type	Type constructor	Expression	Result
pairs	`(,) a b`	fmap (+10) ("abc", 1)	("abc", 11)
lists	`[] b`	fmap (+10) [1,2,3]	[11,12,13]
Maybe	`Maybe b`	fmap (+10) (Just 1)	Just 11
		fmap (+10) Nothing	Nothing
Either	`Either a b`	fmap (+10) (Left 1)	Left 1
		fmap (+10) (Right 1)	Right 11

- Infix notation: `fmap reverse xss == reverse <$> xss`

Examples of functors

Type	Type constructor	Expression	Result
pairs	`(,) a b`	fmap (+10) ("abc", 1)	("abc", 11)
lists	`[] b`	fmap (+10) [1,2,3]	[11,12,13]
Maybe	`Maybe b`	fmap (+10) (Just 1)	Just 11
		fmap (+10) Nothing	Nothing
Either	`Either a b`	fmap (+10) (Left 1)	Left 1
		fmap (+10) (Right 1)	Right 11

- Infix notation: `fmap reverse xss == reverse <$> xss`
- Practical session: instances for `MyList` and trees
- Practical session: test functor laws using QuickCheck
- We assume that you studied Appendix A-2 on QuickCheck

Suggestions for more practice

- Define Paeno arithmetic using the `data` keyword. Make it a member of the `Eq` typeclass.
- Define the function `lefts :: [Either a b] -> [a]`
  Extracts all the `Left` elements of a list, in the same order
  (hint: use list comprehension)
- Define a Functor for triples
- Prove the Functor laws for this triple Functor
- Define a Functor for that does not satisfy the Functor laws
- Define a Functor for `MyList`
- Define QuickCheck tests that tests if the Functor laws hold for your Functors

Self study

- This lecture: “Types and Type Classes”
  - Learn you a Haskell (Lipovaca): Chapter 8
  - Programming in Haskell (Hutton): Chapter 8, Chapter 12.1

Self study

- This lecture: “Types and Type Classes”
  - Learn you a Haskell (Lipovaca): Chapter 8
  - Programming in Haskell (Hutton): Chapter 8, Chapter 12.1
- Lecture Friday (03-05): “Parsing in Haskell”
  - Knowledge on parsing is assumed (top-down, (E)BNF, recursive descent)
  - MOD08 students: part of compiler construction
  - Other students: self-study; suggestions:
    - “CS143 Notes: Parsing”, David L. Dill
    - Compiler construction video lectures (Canvas)