

# Introduction to types and type classes in Haskell

Daniel Alonso

## Types and type signatures

### Type classes

Functor

Applicative

## Types and type signatures

# Haskell's three levels

Haskell code can be seen at three levels:

**Values** Booleans, numbers, lists, functions. . .

**Types** “Sets of values.”

**Kinds** Something else.

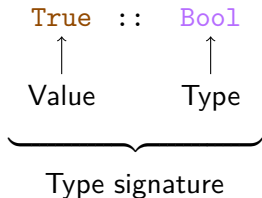
# Motivation for types

- Static typing is a (lightweight) formal verification method that guarantees the absence of certain classes of errors (e.g. `True + 'c'`).
- The static type of a function is a partial, machine checked specification (e.g. `fst :: (a,b) -> a`).
- Types are a design language.
- Types help with software maintenance.

(Stolen from SPJ's lecture "Adventure with types in Haskell")

# Type signatures

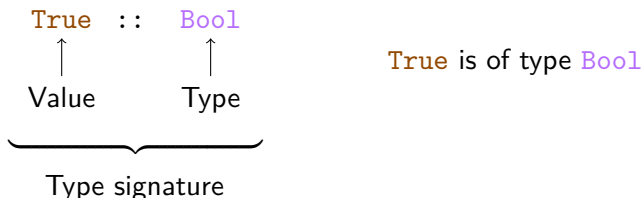
The `::` symbol is read as “is of type.”



`True` is of type `Bool`

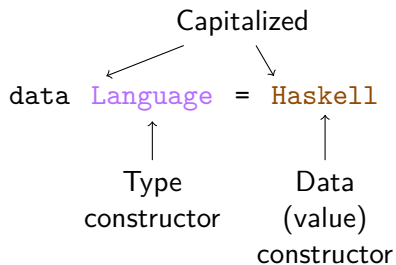
# Type signatures

The `::` symbol is read as “is of type.”



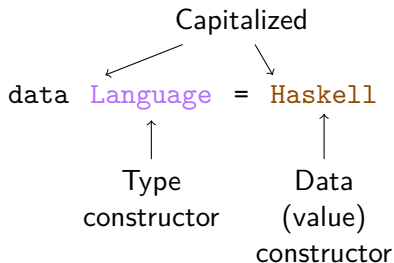
```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

# Defining types





# Defining types



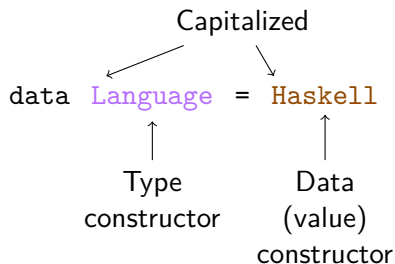
## In the interpreter

Ask for the type of values with `:t`.

```
Prelude> :t Haskell
```

```
Haskell :: Language
```

# Defining types



## In the interpreter

Ask for the type of values with `:t`.

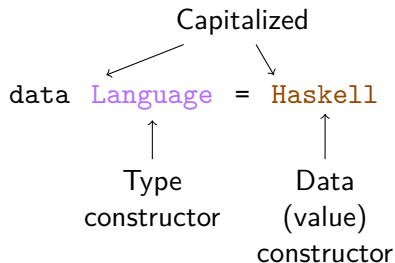
```
Prelude> :t Haskell
```

```
Haskell :: Language
```

```
Prelude> :t Language
```

```
<interactive>:1:1: error:  
Data constructor not in scope:  
Language
```

# Defining types



## In the interpreter

Ask for the type of values with `:t`.

```
Prelude> :t Haskell
```

```
Haskell :: Language
```

```
Prelude> :t Language
```

```
<interactive>:1:1: error:
Data constructor not in scope:
Language
```

Same name for type and data constructors is OK:

```
data Hask = Hask
```

## More than one value

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

↙ ↘  
Different data  
constructors  
for Bool

In the interpreter

```
Prelude> :t False
```

```
False :: Bool
```

```
Prelude> :t True
```

```
True :: Bool
```

## More than one value

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

↙ ↘  
Different data  
constructors  
for Bool

In the interpreter

```
Prelude> :t False
```

```
False :: Bool
```

```
Prelude> :t True
```

```
True :: Bool
```

Any number of data constructors:

```
data Three = A | B | C
```

⋮

Conceptually:

```
data Int = ... | -1 | 0 | 1 | ...
```

## Data constructors with parameters

```
data Point = Point Double Double
```

                                  ↖      ↗  
                                  (Types of the)  
                                  parameters to the  
                                  data constructor

## Data constructors with parameters

```
data Point = Point Double Double
```

                                  ↖      ↗  
                                  (Types of the)  
                                  parameters to the  
                                  data constructor

### In the interpreter

```
Prelude> :t Point
```

```
Point :: Double -> Double -> Point
```

```
Prelude> :t Point 0.0 0.0
```

```
Point 0.0 0.0 :: Point
```

# Parametric polymorphism: motivation

Optional value:

```
data MaybeInt = NothingInt | JustInt Int
```

```
data MaybeString = NothingString | JustString String
```

⋮

Is there a better way?



# Parametric polymorphism

Function definition:

positive n = n > 0

Function name      parameter declaration (lowercase)      parameter use

# Parametric polymorphism

Function definition:

`positive n = n > 0`

Function name      parameter declaration (lowercase)      parameter use

Similarly, for types:

`data Maybe a = Nothing | Just a`

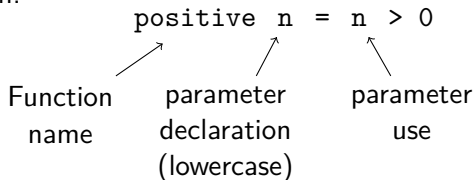
type parameter declaration      type parameter use

# Parametric polymorphism

Function definition:

`positive n = n > 0`

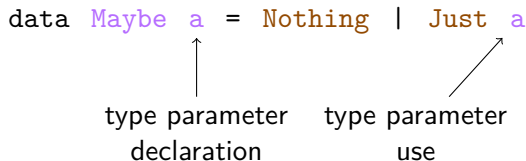
Function name      parameter declaration (lowercase)      parameter use



Similarly, for types:

`data Maybe a = Nothing | Just a`

type parameter declaration      type parameter use



In the interpreter

```
Prelude> :t Nothing
Nothing :: Maybe a
Prelude> :t Just
Just :: a -> Maybe a
Prelude> :t Just True
Just True :: Maybe Bool
```

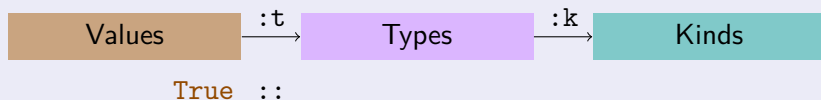
## Introducing kinds

- “Arity of type constructors:” how many type parameters do they have (and of what kinds)?
- Inhabited types (types with values) have kind  $*$  (`Bool`  $:: *$ ).
- Higher kinded type constructors (`Maybe`  $:: * \rightarrow *$ ) have no values.

# Introducing kinds

- “Arity of type constructors:” how many type parameters do they have (and of what kinds)?
- Inhabited types (types with values) have kind  $*$  (`Bool`  $:: *$ ).
- Higher kinded type constructors (`Maybe`  $:: * \rightarrow *$ ) have no values.

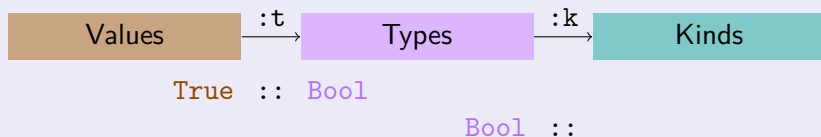
## In the interpreter



# Introducing kinds

- “Arity of type constructors:” how many type parameters do they have (and of what kinds)?
- Inhabited types (types with values) have kind  $*$  (`Bool`  $:: *$ ).
- Higher kinded type constructors (`Maybe`  $:: * \rightarrow *$ ) have no values.

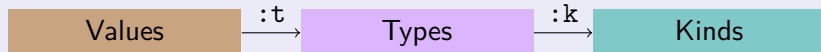
## In the interpreter



# Introducing kinds

- “Arity of type constructors:” how many type parameters do they have (and of what kinds)?
- Inhabited types (types with values) have kind  $*$  (`Bool`  $:: *$ ).
- Higher kinded type constructors (`Maybe`  $:: * \rightarrow *$ ) have no values.

## In the interpreter



`True`  $::$  `Bool`

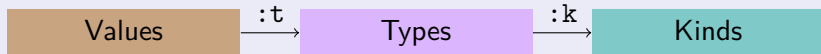
`Bool`  $::$   $*$

`Just`  $::$

# Introducing kinds

- “Arity of type constructors:” how many type parameters do they have (and of what kinds)?
- Inhabited types (types with values) have kind  $*$  (`Bool`  $:: *$ ).
- Higher kinded type constructors (`Maybe`  $:: * \rightarrow *$ ) have no values.

## In the interpreter



`True`  $::$  `Bool`

`Bool`  $::$   $*$

`Just`  $::$  `a`  $\rightarrow$  `Maybe a`

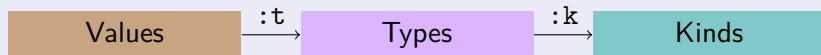
`Maybe`  $::$



# Introducing kinds

- “Arity of type constructors:” how many type parameters do they have (and of what kinds)?
- Inhabited types (types with values) have kind  $*$  (`Bool`  $:: *$ ).
- Higher kinded type constructors (`Maybe`  $:: * \rightarrow *$ ) have no values.

## In the interpreter



`True`  $::$  `Bool`

`Bool`  $::$   $*$

`Just`  $::$  `a`  $\rightarrow$  `Maybe a`

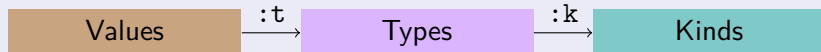
`Maybe`  $::$   $* \rightarrow *$

`Just True`  $::$

# Introducing kinds

- “Arity of type constructors:” how many type parameters do they have (and of what kinds)?
- Inhabited types (types with values) have kind  $*$  (`Bool`  $:: *$ ).
- Higher kinded type constructors (`Maybe`  $:: * \rightarrow *$ ) have no values.

## In the interpreter



`True`  $::$  `Bool`

`Bool`  $::$   $*$

`Just`  $::$  `a`  $\rightarrow$  `Maybe a`

`Maybe`  $::$   $* \rightarrow *$

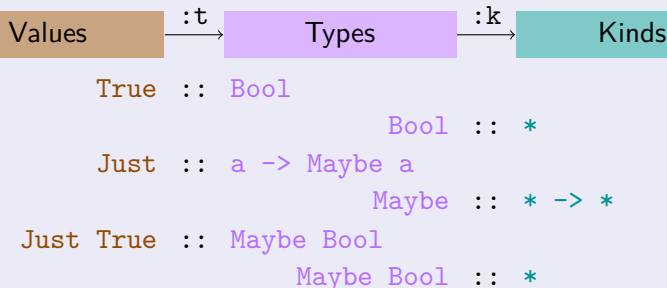
`Just True`  $::$  `Maybe Bool`

`Maybe Bool`  $::$

# Introducing kinds

- “Arity of type constructors:” how many type parameters do they have (and of what kinds)?
- Inhabited types (types with values) have kind  $*$  (`Bool`  $:: *$ ).
- Higher kinded type constructors (`Maybe`  $:: * \rightarrow *$ ) have no values.

## In the interpreter



# Exercises

Given the following types:

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

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

what are the types/kinds of the following values/types?

Left

Right

Either

Just (Left True)

# Exercises

Given the following types:

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

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

what are the types/kinds of the following values/types?

```
Left :: a -> Either a b
```

```
Right
```

```
Either
```

```
Just (Left True)
```

# Exercises

Given the following types:

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

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

what are the types/kinds of the following values/types?

```
Left :: a -> Either a b
```

```
Right :: b -> Either a b
```

```
Either
```

```
Just (Left True)
```

# Exercises

Given the following types:

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

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

what are the types/kinds of the following values/types?

```
Left :: a -> Either a b
```

```
Right :: b -> Either a b
```

```
Either :: * -> * -> *
```

```
Just (Left True)
```

# Exercises

Given the following types:

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

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

what are the types/kinds of the following values/types?

```
Left :: a -> Either a b
```

```
Right :: b -> Either a b
```

```
Either :: * -> * -> *
```

```
Just (Left True) :: Maybe (Either Bool b)
```



## Function type

Functions:

Start with	Notation	Example	Convert
letter	prefix	pAnd True False	True `pAnd` False
non-letter	infix	True && False	(&&) True False

## Function type

Functions:

Start with	Notation	Example	Convert
letter	prefix	pAnd True False	True `pAnd` False
non-letter	infix	True && False	(&&) True False

The same happens with type constructors!

Start with	Notation	Example	Convert
letter	prefix	Function Int Bool	—
non-letter	infix	Int -> Bool	(->) Int Bool

# Function type

Functions:

Start with	Notation	Example	Convert
letter	prefix	pAnd True False	True `pAnd` False
non-letter	infix	True && False	(&&) True False

The same happens with type constructors!

Start with	Notation	Example	Convert
letter	prefix	Function Int Bool	—
non-letter	infix	Int -> Bool	(->) Int Bool

In the interpreter

```
Prelude> :k (->)
```

```
(->) :: * -> * -> *
```

No data constructor for functions:

```
f, g :: Int -> Bool
```

```
f n = n > 0
```

```
g n = odd n
```

# Currying

- Functions only take one parameter (possibly returning other functions):

<code>f :: Int -&gt; Int -&gt; Bool</code>	<code>f' :: Int -&gt; (Int -&gt; Bool)</code>
<code>f m n = m &lt; n</code>	<code>f' m = g</code>
	<code>where g n = m &lt; n</code>

# Currying

- Functions only take one parameter (possibly returning other functions):

$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}$	$f' :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Bool})$
$f \ m \ n = m < n$	$f' \ m = g$
	where $g \ n = m < n$

- $(\rightarrow)$  is right associative:

$a \rightarrow b \rightarrow c \rightarrow d$  is the same as  $a \rightarrow (b \rightarrow (c \rightarrow d))$ .

# Currying

- Functions only take one parameter (possibly returning other functions):

$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}$	$f' :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Bool})$
$f \ m \ n = m < n$	$f' \ m = g$
	where $g \ n = m < n$

- $(\rightarrow)$  is right associative:  
 $a \rightarrow b \rightarrow c \rightarrow d$  is the same as  $a \rightarrow (b \rightarrow (c \rightarrow d))$ .
- $(a \rightarrow b) \rightarrow c$  means the first parameter of the function is a function of type  $a \rightarrow b$ .

# Currying

- Functions only take one parameter (possibly returning other functions):

$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}$	$f' :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Bool})$
$f \ m \ n = m < n$	$f' \ m = g$
	where $g \ n = m < n$

- $(\rightarrow)$  is right associative:  
 $a \rightarrow b \rightarrow c \rightarrow d$  is the same as  $a \rightarrow (b \rightarrow (c \rightarrow d))$ .
- $(a \rightarrow b) \rightarrow c$  means the first parameter of the function is a function of type  $a \rightarrow b$ .

## In the interpreter

```
Prelude> f :: Int -> Int -> Bool; f m n = m < n
```

```
Prelude> :t f
```

```
f :: Int -> Int -> Bool
```

```
Prelude> :t f 2
```

```
f 2 :: Int -> Bool
```

## More exercises

Given the functions

```
f :: Int -> Bool -> Int -> Int
```

```
f m add n = if add then m + n else m - n
```

```
g = f 3 True
```

```
g' = f 3 False
```

What is the type of `g`? And of `g'`? What is the value of `g 2`? And of `g' 2`?

Given the type

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

What is the kind of `Either Int`?



## More exercises

Given the functions

```
f :: Int -> Bool -> Int -> Int
```

```
f m add n = if add then m + n else m - n
```

```
g = f 3 True
```

```
g' = f 3 False
```

What is the type of `g`? And of `g'`? What is the value of `g 2`? And of `g' 2`?

```
g :: Int -> Int
```

```
g' :: Int -> Int
```

Given the type

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

What is the kind of `Either Int`?

## More exercises

Given the functions

```
f :: Int -> Bool -> Int -> Int
```

```
f m add n = if add then m + n else m - n
```

```
g = f 3 True
```

```
g' = f 3 False
```

What is the type of `g`? And of `g'`? What is the value of `g 2`? And of `g' 2`?

```
g :: Int -> Int
```

```
g 2 == 5
```

```
g' :: Int -> Int
```

```
g' 2 == 1
```

Given the type

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

What is the kind of `Either Int`?

## More exercises

Given the functions

```
f :: Int -> Bool -> Int -> Int
```

```
f m add n = if add then m + n else m - n
```

```
g = f 3 True
```

```
g' = f 3 False
```

What is the type of `g`? And of `g'`? What is the value of `g 2`? And of `g' 2`?

```
g :: Int -> Int
```

```
g 2 == 5
```

```
g' :: Int -> Int
```

```
g' 2 == 1
```

Given the type

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

What is the kind of `Either Int`?

```
Either Int :: * -> *
```

## What to remember

- Type definitions consist of type constructors (functions on types) and data constructors (functions on values).
- Types can be seen as sets of values, kinds as the arity of type functions.
- The function type  $(\rightarrow)$  takes two type parameters: the domain (“input”) and the codomain (“output”) types.
- All functions take exactly one parameter, and can return other functions.

Type classes

## Type classes: motivation

- Want to test equality with `v1 == v2` for different types: `Bools`, `Ints`, `Strings`... But not for others: `f1 == f2` (equality between functions?).
- `n1 + n2` should work for `Ints`, `Doubles`, `Complex numbers`...
- The executed code will be different for each type.

## Type classes: ad hoc polymorphism

- Type classes are sets of types that support certain operations. Think of **OOP interfaces, not OOP classes**.

```
class Eq a where
```

```
    (==) :: a -> a -> Bool
```

```
    ∴
```

```
instance Eq Bool where
```

```
    (==) = implementation
```

```
    ∴
```

## Type classes: ad hoc polymorphism

- Type classes are sets of types that support certain operations. Think of **OOP interfaces, not OOP classes**.

```
class Eq a where
  (==) :: a -> a -> Bool
  ...
```

```
instance Eq Bool where
  (==) = implementation
  ...
```

- Subclasses are subsets that support additional operations:

```
class Eq a => Ord a where
  (<=) :: a -> a -> Bool
  ...
```

```
instance Ord Bool where
  (<=) = implementation
  ...
```



# Type classes: ad hoc polymorphism

- Type classes are sets of types that support certain operations. Think of **OOP interfaces, not OOP classes**.

<pre>class Eq a where   (==) :: a -&gt; a -&gt; Bool   ...</pre>	<pre>instance Eq Bool where   (==) = implementation   ...</pre>
--	---

- Subclasses are subsets that support additional operations:

<pre>class Eq a =&gt; Ord a where   (&lt;=) :: a -&gt; a -&gt; Bool   ...</pre>	<pre>instance Ord Bool where   (&lt;=) = implementation   ...</pre>
---	---

- The compiler can autogenerate instances for some type classes:

```
data Language = Lisp | Haskell
  deriving (Eq, Ord, Show)
```

# How to use type classes

In the interpreter

```
Prelude> :k Ord
```

```
Ord :: * -> Constraint
```

Constraint



```
sort :: Ord a => [a] -> [a]
```

```
sort = ...
```

We can use `sort` for lists of any type `a` that is an instance of the `Ord` class.

Multiple constraints possible:

```
f :: (Ord a, Num a) => ...
```

```
g :: (Ord a, Ord b) => ...
```

Functor

## Functor: motivation

- Lists are polymorphic over the type of its elements (`[Bool]`, `[Int]`...). `map` applies a function to all the elements of a list.

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

## Functor: motivation

- Lists are polymorphic over the type of its elements (`[Bool]`, `[Int]`...). `map` applies a function to all the elements of a list.

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

- Can we have polymorphism over the type of the container? A `map` that works for `Maybe a`, `[a]`, `Tree a`...

## Functor: motivation

- Lists are polymorphic over the type of its elements (`[Bool]`, `[Int]`...). `map` applies a function to all the elements of a list.  
`map :: (a -> b) -> [a] -> [b]`
- Can we have polymorphism over the type of the container? A `map` that works for `Maybe a`, `[a]`, `Tree a`...
- Not only for containers.

# Functor

Function application (space or (\$)) has type:

$(\$)$  ::  $(a \rightarrow b) \rightarrow a \rightarrow b$

$f \$ x = f x$

# Functor

Function application (space or (\$)) has type:

$(\$)$  ::  $(a \rightarrow b) \rightarrow a \rightarrow b$

$f \$ x = f x$

class **Functor** f where

**fmap** ::  $(a \rightarrow b) \rightarrow f a \rightarrow f b$



# Functor

Function application (space or (\$)) has type:

$(\$)$  ::  $(a \rightarrow b) \rightarrow a \rightarrow b$

$f \$ x = f x$

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

In the interpreter

```
Prelude> :k Functor
```

```
Functor :: (* -> *) -> Constraint
```

# Functor

Function application (space or (\$)) has type:

$(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b$

$f \$ x = f x$

class **Functor** f where

**fmap** :: (a -> b) -> f a -> f b

In the interpreter

Prelude> :k Functor

**Functor** :: (\* -> \*) -> Constraint

**Maybe** has kind \* -> \*. Let's see its **Functor** instance:

instance **Functor** **Maybe** where

**fmap** :: (a -> b) -> **Maybe** a -> **Maybe** b

**fmap** \_ **Nothing** = **Nothing**

**fmap** g (**Just** x) = **Just** (g x)

## Functor laws

`fmap id = id`

*If the fmapped function does not change the values of type `a`,  
fmap does not change them, either.*

`fmap (g . h) = (fmap g) . (fmap h)`

*Fmapping the composition of functions is the same as composing  
the fmapping of functions.*

## Functor laws

`fmap id = id`

*If the fmapped function does not change the values of type `a`,  
fmap does not change them, either.*

`fmap (g . h) = (fmap g) . (fmap h)`

*Fmapping the composition of functions is the same as composing  
the fmapping of functions.*

Let's see two rogue instances:

`instance Functor Maybe` where

`fmap _ Nothing = Nothing`

`fmap g (Just x) = Nothing`

## Functor laws

`fmap id = id`

*If the fmapped function does not change the values of type `a`,  
fmap does not change them, either.*

`fmap (g . h) = (fmap g) . (fmap h)`

*Fmapping the composition of functions is the same as composing  
the fmapping of functions.*

Let's see two rogue instances:

instance `Functor Maybe` where

`fmap _ Nothing = Nothing`

`fmap g (Just x) = Nothing`

instance `Functor []` where

`fmap _ [] = []`

`fmap g (x:xs) = g x : g x : fmap g xs`

# Functor laws intuition

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

# Functor laws intuition

U Can't Touch This!

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



# Exercises

Write the `Functor` instance for the following types:

```
data EqPair a =  
    EqPair a a
```

```
data DiffPair a b =  
    DiffPair a b
```

```
data Validated e a =  
    Error e | OK a
```

```
data Phantom a =  
    Phantom
```



# Exercises

Write the `Functor` instance for the following types:

```
data EqPair a =  
    EqPair a a  
instance Functor EqPair where  
    fmap g (EqPair x y) =  
        EqPair (g x) (g y)
```

```
data DiffPair a b =  
    DiffPair a b
```

```
data Validated e a =  
    Error e | OK a
```

```
data Phantom a =  
    Phantom
```

# Exercises

Write the `Functor` instance for the following types:

```
data EqPair a =           instance Functor EqPair where
    EqPair a a             fmap g (EqPair x y) =
                           EqPair (g x) (g y)
```

```
data DiffPair a b =       instance Functor (DiffPair a) where
    DiffPair a b
```

```
data Validated e a =
    Error e | OK a
```

```
data Phantom a =
    Phantom
```

# Exercises

Write the `Functor` instance for the following types:

```
data EqPair a =  
    EqPair a a  
instance Functor EqPair where  
    fmap g (EqPair x y) =  
        EqPair (g x) (g y)
```

```
data DiffPair a b =  
    DiffPair a b  
instance Functor (DiffPair a) where  
    fmap g (DiffPair x y) =  
        DiffPair x (g y)
```

```
data Validated e a =  
    Error e | OK a
```

```
data Phantom a =  
    Phantom
```

## Exercises

Write the `Functor` instance for the following types:

```
data EqPair a =           instance Functor EqPair where
    EqPair a a             fmap g (EqPair x y) =
                           EqPair (g x) (g y)
```

```
data DiffPair a b =       instance Functor (DiffPair a) where
    DiffPair a b           fmap g (DiffPair x y) =
                           DiffPair x (g y)
```

```
data Validated e a =      instance Functor (Validated e) where
    Error e | OK a         fmap _ (Error x) = Error x
                           fmap g (OK y) = OK (g y)
```

```
data Phantom a =
    Phantom
```

## Exercises

Write the `Functor` instance for the following types:

```
data EqPair a =           instance Functor EqPair where
    EqPair a a             fmap g (EqPair x y) =
                           EqPair (g x) (g y)

data DiffPair a b =       instance Functor (DiffPair a) where
    DiffPair a b          fmap g (DiffPair x y) =
                           DiffPair x (g y)

data Validated e a =       instance Functor (Validated e) where
    Error e | OK a         fmap _ (Error x) = Error x
                           fmap g (OK y) = OK (g y)

data Phantom a =           instance Functor Phantom where
    Phantom                fmap _ Phantom = Phantom
```

## Exercises

Write the `Functor` instance for the following types:

```
data EqPair a =  
    EqPair a a  
    deriving (Functor)
```

```
data DiffPair a b =  
    DiffPair a b  
    deriving (Functor)      {-# LANGUAGE DeriveFunctor #-}
```

```
data Validated e a =  
    Error e | OK a  
    deriving (Functor)
```

```
data Phantom a =  
    Phantom  
    deriving (Functor)
```

## What to remember

- The **Functor** type class allows to apply a function through an outer context, without knowing what this context is.
- The context will not change.
- Not only for containers:

In the interpreter

```
capitalize :: String -> String
getLine    :: IO String
Prelude> fmap capitalize getLine
"haskell" -- User input
"HASKELL"
```

Applicative



## Applicative: motivation (i)

```
data Contact = Contact Phone Email
valPhone :: String -> Maybe Phone
valEmail :: String -> Maybe Email
```

## Applicative: motivation (i)

```
data Contact = Contact Phone Email
valPhone :: String -> Maybe Phone
valEmail  :: String -> Maybe Email

valContact :: String -> String -> Maybe Contact
valContact sPhone sEmail = case valPhone sPhone of
    Nothing -> Nothing
    Just phone-> case valEmail sEmail of
        Nothing -> Nothing
        Just email -> Just (Contact phone email)
```

Imagine a function with more parameters!

## Applicative: motivation (ii)

Why not `fmap` the `Contact` data constructor into a `Maybe Phone`?

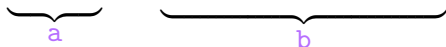
## Applicative: motivation (ii)

Why not fmap the **Contact** data constructor into a **Maybe Phone**?

Remember currying?

**Contact** :: **Phone** -> **Email** -> **Contact**

**Contact** :: **Phone** -> (**Email** -> **Contact**)

  
a                      b

## Applicative: motivation (ii)

Why not `fmap` the `Contact` data constructor into a `Maybe Phone`?

Remember currying?

```
Contact :: Phone -> Email -> Contact
```

```
Contact :: Phone -> (Email -> Contact)
```



`fmap` specialized to `Maybe` has this signature:

```
fmap :: (a -> b)  
      -> Maybe a  
      -> Maybe b
```

## Applicative: motivation (ii)

Why not fmap the **Contact** data constructor into a **Maybe Phone**?

Remember currying?

**Contact** :: **Phone** -> **Email** -> **Contact**

**Contact** :: **Phone** -> (**Email** -> **Contact**)



fmap specialized to **Maybe** has this signature:

fmap :: (a -> b)  
     -> Maybe a  
     -> Maybe b

Let's substitute accordingly:

fmap :: (**Phone** -> (**Email** -> **Contact**))  
     -> Maybe **Phone**  
     -> Maybe (**Email** -> **Contact**)

# Applicative

```
( $\$$ ) :: (a -> b) -> a -> b  
fmap :: (a -> b) -> f a -> f b  
?? :: f (a -> b) -> f a -> f b
```

# Applicative

```
( $\$$ ) :: (a -> b) -> a -> b  
fmap :: (a -> b) -> f a -> f b  
?? :: f (a -> b) -> f a -> f b
```

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



# Applicative

```
( $\$$ ) :: (a -> b) -> a -> b  
fmap :: (a -> b) -> f a -> f b  
?? :: f (a -> b) -> f a -> f b
```

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

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

## Applicative valContact

```
valContact :: String -> String -> Maybe Contact
```

```
valContact p e = fmap Contact (valPhone p) <*> (valEmail e)
```

## Applicative valContact

```
valContact :: String -> String -> Maybe Contact
```

```
valContact p e = fmap Contact (valPhone p) <*> (valEmail e)
```

Or better yet, using <\$> (infix version of fmap):

```
valContact p e = Contact <$> valPhone p <*> valEmail e
```

## Applicative valContact

```
valContact :: String -> String -> Maybe Contact
```

```
valContact p e = fmap Contact (valPhone p) <*> (valEmail e)
```

Or better yet, using <\$> (infix version of fmap):

```
valContact p e = Contact <$> valPhone p <*> valEmail e
```

This can scale to any number of arguments:

```
valContact p e a ... = Contact <$> valPhone p
                        <*> valEmail e
                        <*> valAddress a
                        ⋮
```

# Applicative laws

They exist.

## Exercises

Write the `Applicative` instance for the following types:

```
data Validated e a = Error e | OK a
```

```
data DiffPair a b = DiffPair a b
```

## Exercises

Write the `Applicative` instance for the following types:

```
data Validated e a = Error e | OK a

instance Applicative (Validated e) where
  (OK f) <*> (OK x) = OK (f x)
  (Error y) <*> (OK _) = Error y
  (OK _) <*> (Error y) = Error y
  (Error y1) <*> (Error y2) = Error y1
  pure x = OK x

data DiffPair a b = DiffPair a b
```

## Exercises

Write the `Applicative` instance for the following types:

```
data Validated e a = Error e | OK a

instance Applicative (Validated e) where
  (OK f) <*> (OK x) = OK (f x)
  (Error y) <*> (OK _) = Error y
  (OK _) <*> (Error y) = Error y
  (Error y1) <*> (Error y2) = Error y1
  pure x = OK x
```

```
data DiffPair a b = DiffPair a b

instance Applicative (DiffPair a) where
  (DiffPair x1 f) <*> (DiffPair x2 y) = DiffPair x1 (f y)
  pure x = DiffPair ?? x
```



## Example

**Applicative** works for contexts other than containers, too. Each of the arguments can be an action that produces some effect in the context and returns the appropriate value.

```
parserContact = Contact <$> parserPhone <*> parserEmail
```

## What to remember

- **Applicative** allows to “fmap functions with more than one parameter” into a context.
- Each of the arguments to the function can produce effects.
- `pure` allows to “inject” values into a context without producing any effect.

To learn more

Haskell Programming from first principles:

<http://haskellbook.com/>

Typeclassopedia:

<https://wiki.haskell.org/Typeclassopedia>

Thanks