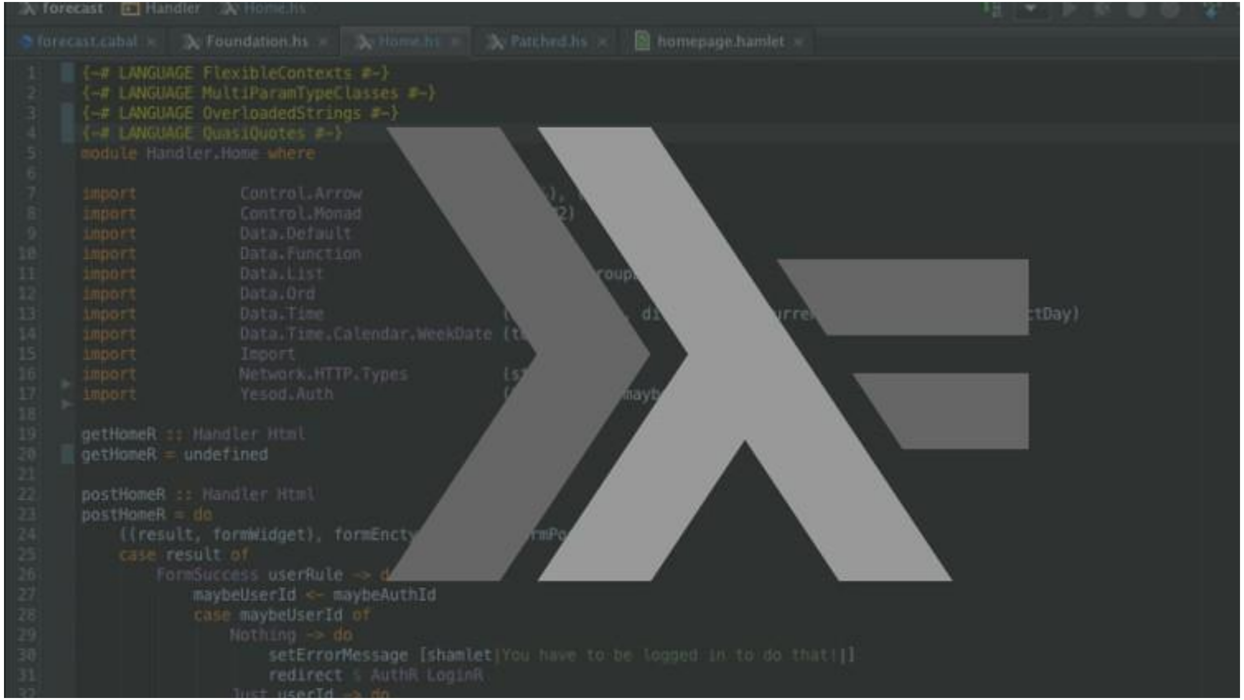
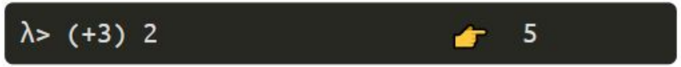
Binary Trees as Functors



```haskell
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes #-}
module Handler.Home where

import           Control.Arrow                     ), 
import           Control.Monad                      2)
import           Data.Default
import           Data.Function
import           Data.List                           roup
import           Data.Time                        (       , di      urre                 tDay)
import           Data.Time.Calendar.WeekDate (t
import           Import
import           Network.HTTP.Types          (s
import           Yesod.Auth                  (              mayb

getHomeR :: Handler Html
getHomeR = undefined

postHomeR :: Handler Html
postHomeR = do
    ((result, formWidget), formEncty              rmPo
    case result of
        FormSuccess userRule -> d
            maybeUserId <- maybeAuthId
            case maybeUserId of
                Nothing -> do
                    setErrorMessage [shamlet|You have to be logged in to do that!|]
                    redirect $ AuthR LoginR
                Just userId -> do
```

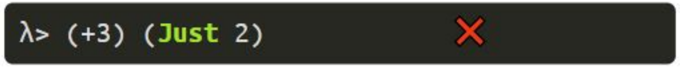
Functors

We already know how to apply functions:



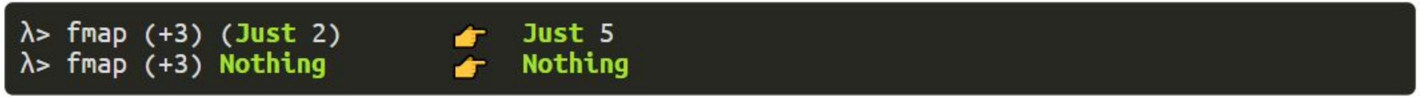
```
λ> (+3) 2                 👉  5
```

But...



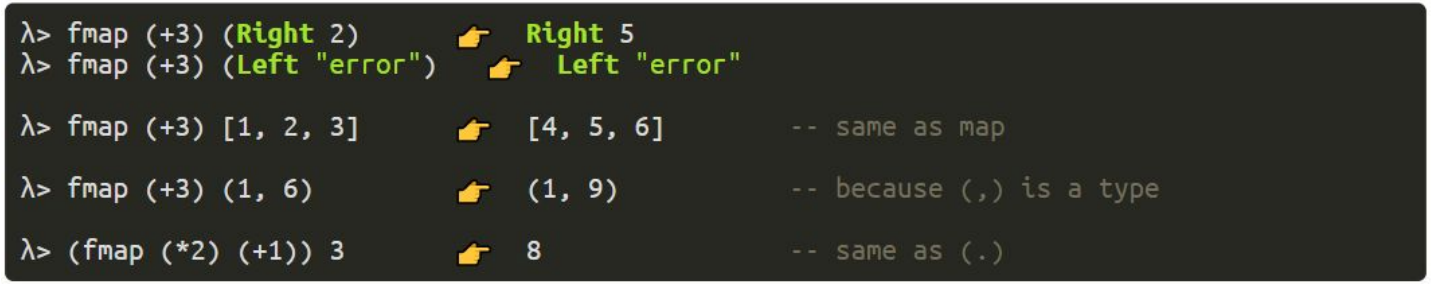
```
λ> (+3) (Just 2)               ✗
```

In this case, we can use `fmap`!



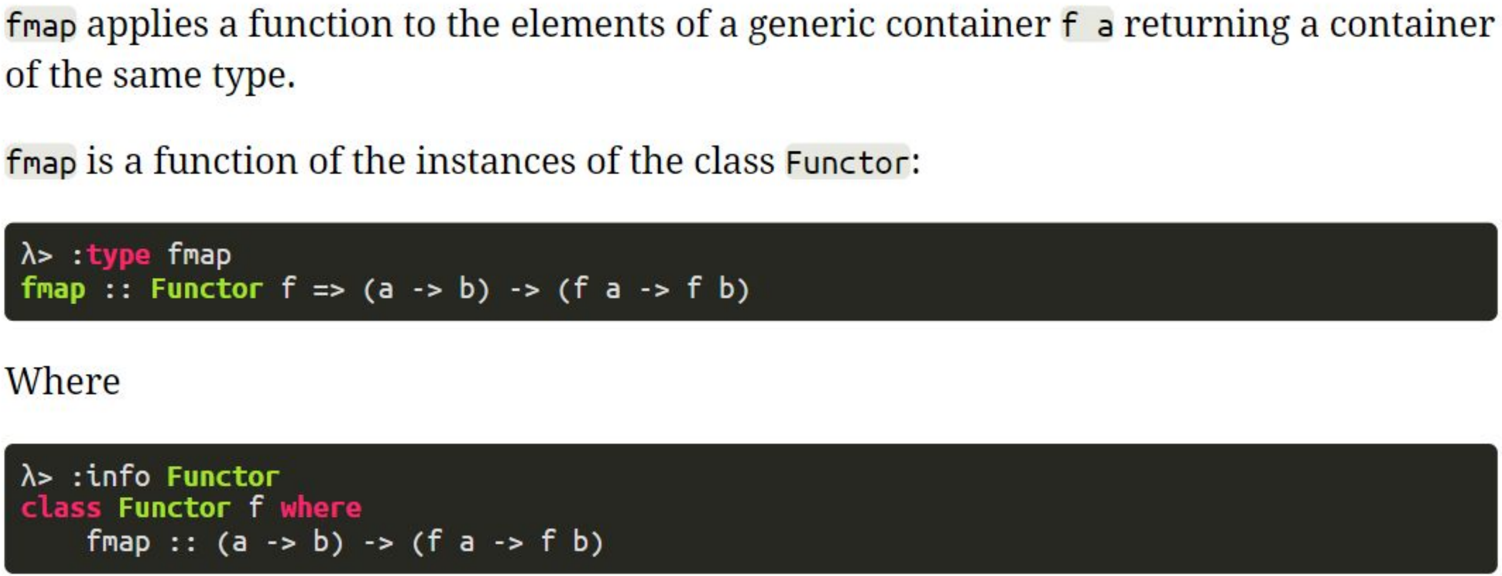
```
λ> fmap (+3) (Just 2)     👉  Just 5
λ> fmap (+3) Nothing      👉  Nothing
```

And it also works with `Either`, lists, tuples and functions:



```
λ> fmap (+3) (Right 2)     👉  Right 5
λ> fmap (+3) (Left "error")  👉  Left "error"

λ> fmap (+3) [1, 2, 3]     👉  [4, 5, 6]        -- same as map

λ> fmap (+3) (1, 6)        👉  (1, 9)           -- because (,) is a type

λ> (fmap (*2) (+1)) 3      👉  8                -- same as (.)
```

Implementation of fmap



`fmap` applies a function to the elements of a generic container `f a` returning a container of the same type.

`fmap` is a function of the instances of the class `Functor`:

```
λ> :type fmap
fmap :: Functor f => (a -> b) -> (f a -> f b)
```

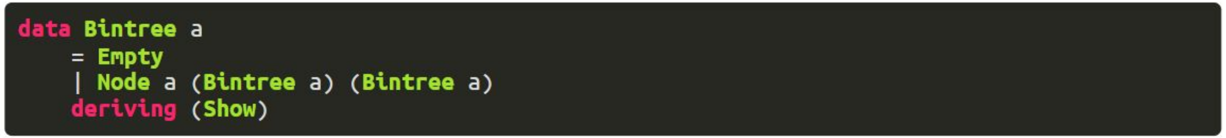
Where

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

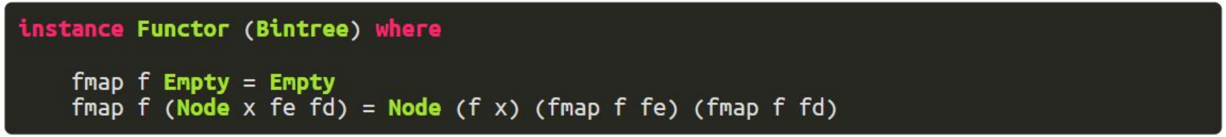
Lucas Bazilio - Udemy

Binary Trees as Functors

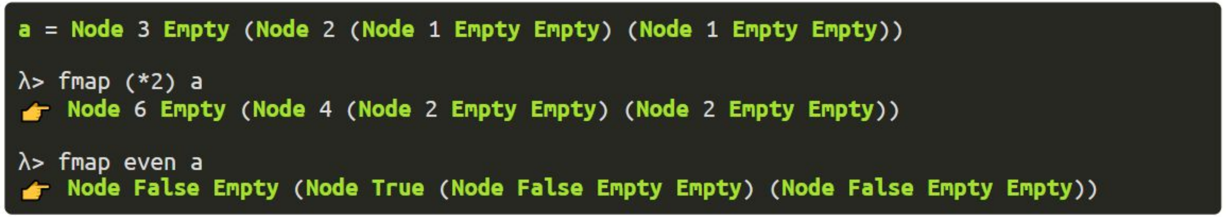
Own instantiation of Functors for Binary Trees:



```haskell
data Bintree a
    = Empty
    | Node a (Bintree a) (Bintree a)
    deriving (Show)
```



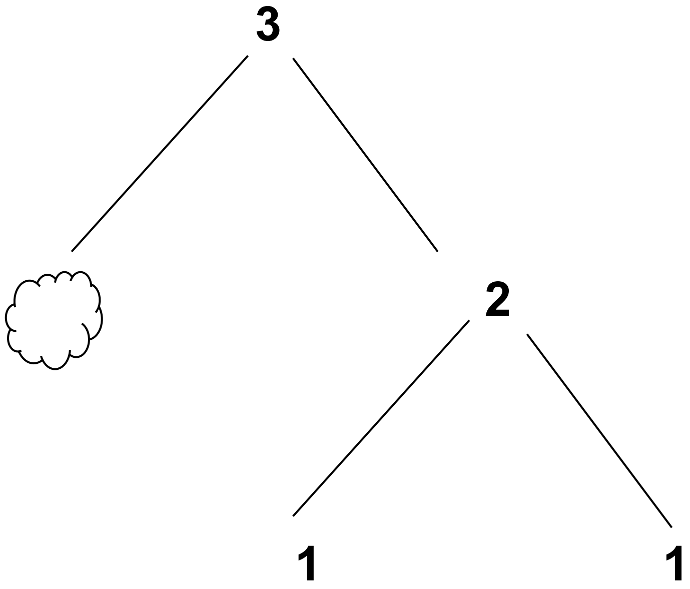
```haskell
instance Functor (Bintree) where

    fmap f Empty = Empty
    fmap f (Node x fe fd) = Node (f x) (fmap f fe) (fmap f fd)
```



```haskell
a = Node 3 Empty (Node 2 (Node 1 Empty Empty) (Node 1 Empty Empty))

λ> fmap (*2) a
👉 Node 6 Empty (Node 4 (Node 2 Empty Empty) (Node 2 Empty Empty))

λ> fmap even a
👉 Node False Empty (Node True (Node False Empty Empty) (Node False Empty Empty))
```

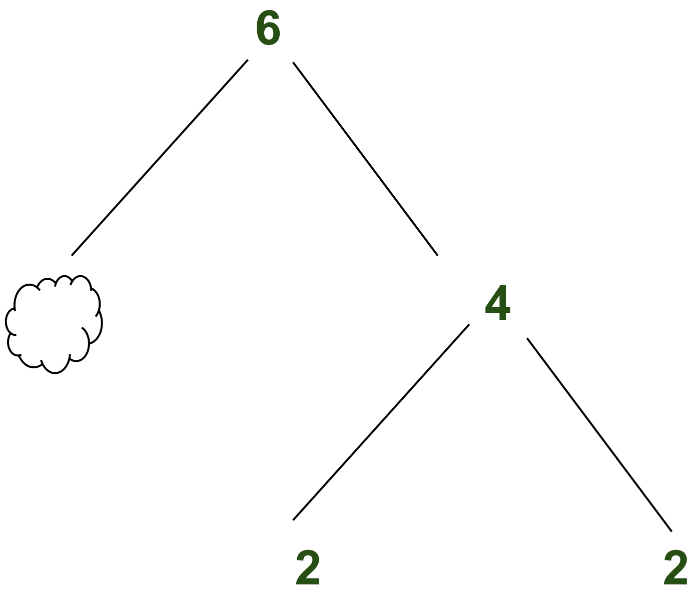
Example

fmap (*2)



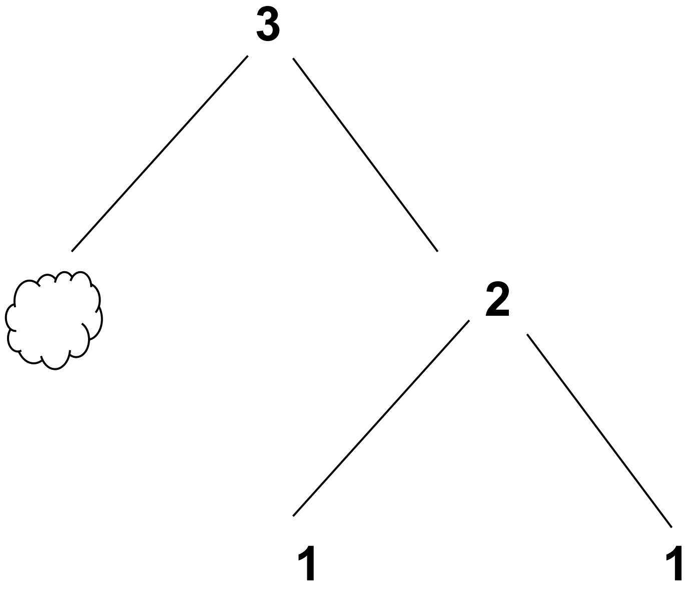
Example

fmap (*2)



Example

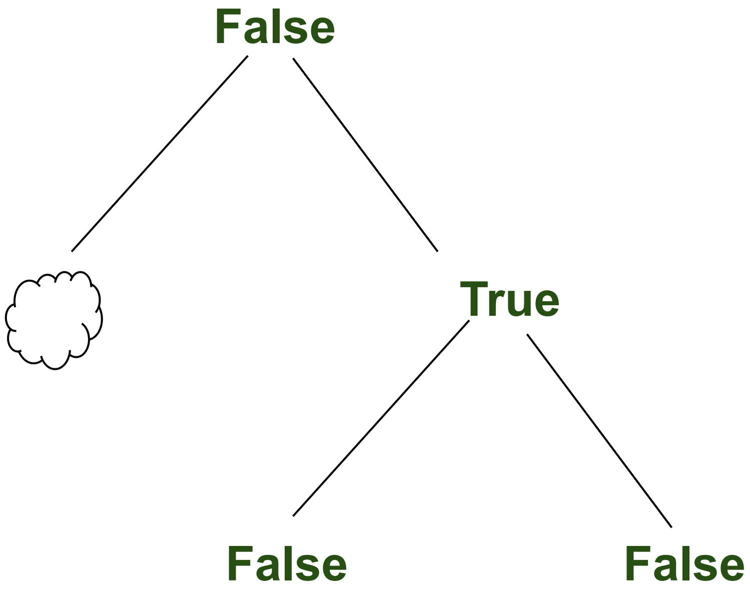
fmap even



Lucas Bazilio - Udemy

Example

fmap even



False

True

False

False