

CS 252:

Advanced Programming Language Principles



Functors

Prof. Tom Austin

San José State University

Review:

Add 1 to each element
in a list of numbers

The Functor typeclass

```
class Functor f where
```

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

Compare fmap to map:

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

A **functor** is something
that can be mapped over.

Box analogy for functors



Maps as functors

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

```
Prelude> map (+1) [1,2,3]  
[2,3,4]
```

```
Prelude> fmap (+1) [1,2,3]  
[2,3,4]
```

```
Prelude> fmap (+1) []  
[]
```

```
Prelude> fmap (+1) $ Just 3  
Just 4
```

```
Prelude> fmap (+1) $ Nothing  
Nothing
```

Maybe as a functor

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


Either type

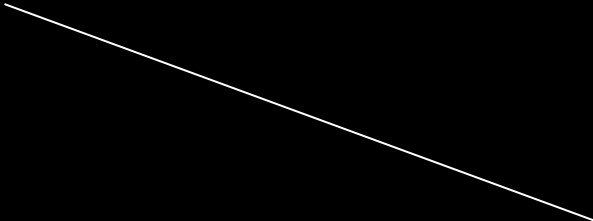
```
data Either a b = Left a  
                | Right b  
deriving (Eq, Ord, Read, Show)
```

```
Prelude> fmap (+1) $ Right 20
```

```
Right 21
```

```
Prelude> fmap (+1) $ Left 20
```

```
Left 20
```



???

Either type

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

```
deriving (Eq, Ord, Read, Show)
```

Expected type

Either type

```
data Either a b = Left a  
                | Right b  
deriving (Eq, Ord, Read, Show)
```



Error type

Either as a functor

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

Haskell Input/Output



Side effects & monads

- Haskell avoids side effects
 - Inevitable in real programs
- Monads
 - related to functors
 - used to compartmentalize side effects

Haskell Input/Output

- Why does `main` have this type?

`main :: IO ()`

- Why does `getLine` have this type?

`getLine :: IO String`

Hello world in Haskell

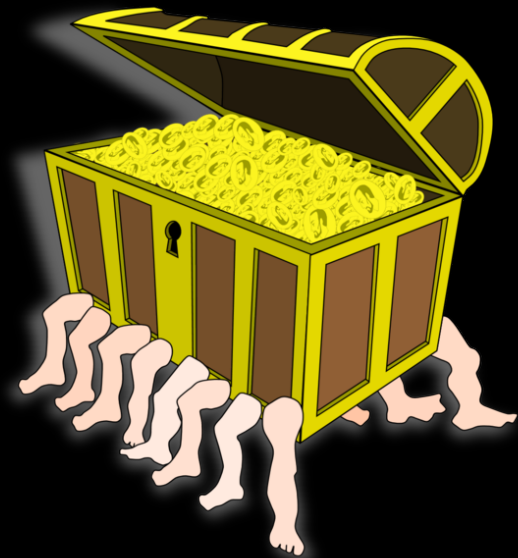
```
main = putStrLn "hello"
```

We can call other functions
that perform file I/O, but
their type will also include
an `IO` somewhere

Do syntax


```
main = do
  putStrLn "Who goes there?"
  name <- getLine
  putStrLn $ "Welcome, " ++ name
```

Ppulling data out
of an IO "box"



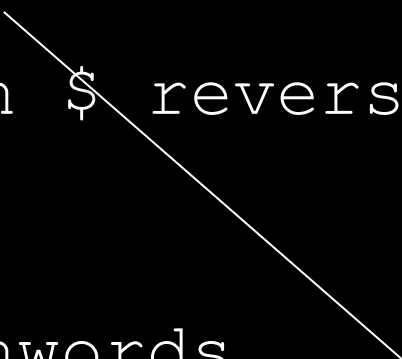
A more complex example

```
main = do
  line <- getLine
  if null line
    then return ()
    else do
      putStrLn $ reverseWords line
      main
```



The unit type

```
reverseWords = unwords .
  map reverse . words
```



Ah! Something
familiar.

No

IF YOU THINK THIS HAS A HAPPY ENDING

**YOU HAVEN'T BEEN PAYING
ATTENTION**

quickmeme.com

Haskell's `return`: the single
worst named keyword in any
language ever made.

Haskell's `return`

- unrelated to `return` in other languages
- better names: "wrap" or "box":

`return`

puts a value in a "box"

`->`

gets contents of a "box"

Haskell's return

```
*Main> :t ()
```

```
() :: ()
```

```
*Main> :t (return ())
```

```
(return ()) :: Monad m => m ()
```



We'll come back to
Monads later

Is Io a Functor?

```
main = do
  line <- fmap (++"!!!") getLine
  putStrLn line
```

fmap appends
the string "!!!"
to the input from
getLine.

Functor IO

```
instance Functor IO where  
  fmap f action = do  
    result <- action  
    return (f result)
```

Take the value out
of its box

Apply f to result,
then put the value
back in the box

Functor Laws

(or at least strong suggestions)



Functor Law #1

If we map the `id` function over a functor, the functor that we get back should be the same as the original functor.

```
Prelude> fmap id (Just 3)
```

```
Just 3
```

```
Prelude> fmap id Nothing
```

```
Nothing
```

```
Prelude> fmap id [1,2,3]
```

```
[1,2,3]
```

Functor Law #2

Composing two functions and then mapping the resulting function over a functor should be the same as first mapping one function over the functor and then mapping the other one.

More formally written:

$$\text{fmap } (f \cdot g) = \text{fmap } f \cdot \text{fmap } g$$

The functor laws are not enforced, but they make your code easier to reason about.

Lab: Functors

Add support for fmap to the Tree type.

Download functors.lhs from the course website.