NEWS
HW1 GRADING

- Grading is in progress, back in a few days
- Style definitely matters
  - Don’t repeat yourself
  - Don’t use a ton of nested ifs
  - If you’re not sure, hlint/ask us
- No points off for style this time
  - May deduct style points starting HW2

Read our comments on your HW!
LAST TIME: TYPECLASSES

1. Declare class with required functions
2. Implement class for your type
3. Fns can use typeclass constraints
```haskell
class Eq a => Ord a where
    (<) :: a -> a -> Bool
    -- ... more stuff ...

data Nat = Zero | Succ Nat

instance Ord Nat where
    Zero < Zero = False
    Succ _ < Zero = False
    Zero < Succ _ = True
    Succ n < Succ m = n < m
    -- ... more stuff

sort :: Ord a => [a] -> [a]
sort list = -- ... < ...
```
A PEEK UNDER THE HOOD
Given `class` declaration...

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

Compiler makes dictionary `type`...

```haskell
data OrdDict a = MkOrdDict { (<> :: a -> a -> Bool
  , (<=) :: a -> a -> Bool }
ENCODE INSTANCE INFO

• Given instance declaration for type...

```haskell
instance Ord Nat where
    n < n' = natLessThan n n'
    n <= n' = natLeqThan n n'
```

• Compiler makes dictionary...

```haskell
NatOrdDict :: OrdDict Nat
NatOrdDict = MkOrdDict { (<) = natLessThan
                           , (<=) = natLeqThan }
```
Say we have function to find the bigger tuple element

```
max :: Ord a => a -> a -> a
max x y |
  | x < y    = y
  | otherwise = x
```

Compiler replaces constraint with dictionary

gets method instances from the dictionary

```
max' :: OrdDict a -> a -> a -> a
max' dict x y |
  | ((<) dict) x y    = y
  | otherwise          = x
```
ADJUST FUNCTION CALLS

- Say we call the `max` function

```haskell
bigger = max Zero (Succ Zero)
```

- Compiler adds in the dictionary for `Nat`

```haskell
bigger = max' NatOrdDict Zero (Succ Zero)
```

- Voilà! No more typeclasses, just plain functions
MAX ON OTHER TYPES

• Say we call the max function on Char

bigger = max 'a' 'b'

• Compiler adds in the dictionary for Char

bigger = max' CharOrdDict 'a' 'b'
TODAY: FUNCTOR
GOING UP A LEVEL

- So far: typeclass instances for types
- Many things in Haskell are not types:
  - `Maybe`
  - `[]`
- They need a type argument to become a type:
  - `Maybe Int`
  - `[Int]`

Define typeclasses for these things!
We can map over many things: \texttt{Maybe}, lists, trees, ...

Factor this into a type class:

\texttt{class Functor f where}
\texttt{  fmap :: (a -> b) -> f a -> f b}

Think: a container \( f \) is “mappable” if it has a \texttt{fmap}

\begin{itemize}
\item Note: \( f \) doesn’t always need to be a “container”
\end{itemize}
EXAMPLES OF FUNCTOR
WARMUP: LISTS

• We already know a mapping function for lists:

```haskell
instance Functor ([]) where
  fmap = map
  -- infix: foo <$> bar === fmap foo bar

  -- What's the type?
  -- fmap :: (a -> b) -> [a] -> [b]
```
Would like to map over a `Maybe`:

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

-- What's the type?
-- fmap :: (a -> b) -> Maybe a -> Maybe b
```
“READER”

- Previous examples: containers
- This example: type of “reader” functions
  - Conversions from type \( r \) to something else

```haskell
instance Functor ((->) r)
    -- What's the heck is this type??
    -- fmap :: (a -> b) -> ((->) r a) -> ((->) r b)
    -- fmap :: (a -> b) -> (r -> a) -> (r -> b)
    -- Solution is now clear:

    fmap fab fra = \r -> fab (fra r)
```
FUNCTOR LAWS
A BROKEN fmap

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

-- What's the type?
-- fmap :: (a -> b) -> Maybe a -> Maybe b
```

- Type is OK, but it doesn’t seem to “map”...
FOLLOW THE LAWS

• Many Haskell typeclasses come with “laws”
  ▪ Expected equations that should hold
• You should check the laws hold
  ▪ Compiler won’t check these laws for you
  ▪ Breaking laws is almost always a bug
FUNCTOR LAW: IDENTITY

-- Identity function id
-- id :: a -> a

fmap id === id

• Mapping a do-nothing function should do nothing
A BROKEN fmap

\begin{verbatim}
instance Functor Maybe where
    fmap _ Nothing = Nothing
    fmap _ (Just x) = Nothing

-- What's the type?
-- fmap :: (a -> b) -> Maybe a -> Maybe b
\end{verbatim}

- Breaks law: \( \text{fmap id (Just 42)} === \text{Nothing} \)
FUNCTOR LAW: COMPOSITION

-- Suppose: \( f :: a \rightarrow b, g :: b \rightarrow c \)
\[ \text{fmap } (g \ . \ f) \equiv \text{fmap } g \ . \ \text{fmap } f \]

- Map \( f \) then map \( g \) is same as map \( g \ . \ f \)