In-class Exercises on Monoids, Foldables, and Monads

1

- We had Quiz 2 on Tuesday
- PS5 is due October 15th
- Project proposal is due October 15th
  - Team up with a partner. A team of two is required
  - Please read carefully
  - Do some research and think through carefully

2

Outline

- Monoids
  - Over integers, over boolean, over lists
- Foldables
  - Lists and trees
- Generic monadic functions
  - Write definition for each function
    - Write using >>=
    - Then write using do notation (or vice versa)
  - Then write at least one additional use case!

3

Semigroup and Monoid

An associative operation combining elements of a set:

```
class Semigroup a where
  (<>) :: a -> a -> a
```

An identity elements:

```
class Semigroup a => Monoid a where
  mempty :: a
```

LAWS: mempty <> x = x
x <> mempty = x

4

List is a Monoid

What are some instances of Monoid?

Why [a] and not [] ?

```
instance Semigroup [a] where
  (<>) :: [a] -> [a] -> [a]
  (<>) = (++)

instance Monoid [a] where
  mempty = []
```


newtype Sum = Sum { getSum :: Int }

We can define a numeric monoid for addition:

```
newtype Sum a = Sum { getSum :: a }
> x = Sum 10
> getSum x
```

```
instance Num a => Semigroup (Sum a) where
  (<>) :: (Sum a) -> (Sum a) -> (Sum a)
  Sum x <> Sum y = Sum $ x+y
instance Num a => Monoid (Sum a) where
  mempty = Sum 0
```

We can define an analogous numeric monoid for multiplication.

We can define a boolean monoid for logical and:

```
newtype All = All { getAll :: Bool }
> x = All True
> getAll x
```

```
instance Semigroup All where
  (<>) :: All -> All -> All
  All x <> All y = All $ x && y
instance Monoid All where
  mempty = All True
```

And an analogous boolean monoid Any, for logical or.

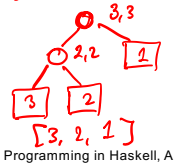
PS5 Tree

Vec a — first, final, insert, delete — index

Can we define an instance of Semigroup? Monoid?

```
data AVL a = Atom a -- leaf
           | Node
             Int -- cached number of elements
             Int -- cached height
             (AVL a) -- left branch
             (AVL a) -- right branch
```

t = Node (Node (Atom 3) (Atom 2)) (Atom 1)



t1 <> t2 → t3

toList (t1) ++ toList (t2) = toList (t3)

PS5 Vec

Can we define an instance of Semigroup? Monoid?

```
data Vec a = Empty
           | Tree (AVL a) -- non-empty tree with data at leaves
```


PS4 MergesortList

A wrapper around the list type to represent sorted lists.

```
newtype MergesortList a = ML [a]
                        deriving (Eq,Show)
```

Can we define an instance of Semigroup? Monoid?

Func (+1)

Can we make function type **b->b** an instance of Monoid?

```
newtype Func b = Func {getFunc :: b->b}

instance Semigroup (Func b) where
  -- <> :: (Func b) -> (Func b) -> (Func b)
  Func f1 <> Func f2 = Func (f1 . f2)

instance Monoid (Func b) where
  mempty = Func (\x->x)   -- Func id
```


Why are Monoids and wrapper types useful anyway?

```
foldList :: Monoid a => [a] -> a
foldList xs = List.foldr (<>) mempty xs
```

Foldable Type Class

Data of types that are instances of Foldable can be "folded" (or "reduced") to single values

Monoids are closely related to Foldables. They combine all values in a foldable structure to give a single value

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr :: (a -> b -> b) -> b -> t a -> b
  length :: t a -> Int
  toList :: t a -> [a]

  … -- other functions

  {-# MINIMAL foldMap | foldr #-}
```

minimal complete definition

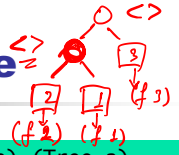
List is Foldable

Why not [a]? Takes types of kind * -> *

```
instance Foldable [] where
  -- foldMap :: Monoid m => (a -> m) -> [a] -> m
  foldMap f [] = mempty
  foldMap f (x:xs) = f x <> foldMap f xs
```

~~Reference~~

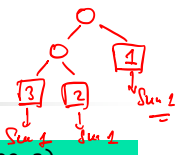
Tree is Foldable




```
data Tree = Leaf a | Node (Tree a) (Tree a)
```

```
instance Foldable Tree where
  -- flodMap :: Monoid m => (a->m) -> (Tree a) -> m
  foldMap f (Leaf x) = f x
  foldMap f (Node lt rt) =
              foldMap f lt <> foldMap f rt
```

```
t :: Tree Int
t = Node (Node (Leaf 2) (Leaf 1)) (Leaf 3)
```


```
data Tree = Leaf a | Node (Tree a) (Tree a)
```

Define a tree fold that counts the leaves of the tree as a single call to `foldMap`.

```
count :: Tree a -> Int
count t = getSum $ foldMap (\_ -> Sum 1) t
```

Define a tree fold that computes the minimum in a tree of ~~numeric values~~ Ints, again as a call to `foldMap`. Need Min monoid

```
minimum :: Tree Int -> Int
minimum t = getMin $ foldMap (\x -> Min x) t
```

NL is Foldable

```
data NL a = Atom a | List [NL a]
```

```
instance Foldable NL where
  -- foldMap :: Monoid m => (a -> m) -> NL a -> m
  foldMap f (Atom x) = f x
  foldMap f (List xs) = foldr (<>) mempty (map (foldMap f) xs)
```

```
x = List [Atom 1, Atom 2, Atom 3]
```

```
z = List [Atom 0, List [List [Atom 1]]]
```

17

An Equivalent foldMap

```
data NL a = Atom a | List [NL a]
```

```
instance Foldable NL where
  -- foldMap ::
  foldMap f (Atom x) = f x
  foldMap f (List []) = mempty
  foldMap f (List (x:xs)) = foldMap f x <> foldMap f (List xs)
```

```
x = List [Atom 1, Atom 2, Atom 3]
```

```
z = List [Atom 0, List [List [Atom 1]]]
```


18

```
data NL a = Atom a | List [NL a]
```

Define atomcount and flatten as a call to generic foldMap.

```
atomcount :: NL a -> Int
atomcount nl = getSum $ foldMap (\_ -> Sum 1) nl
```

```
flatten ::
flatten  = toList
```


19

Exercise

`{-# MINIMAL foldMap | foldr #-}` means one need only define `foldMap` or `foldr`. Haskell can derive all other functions from either one of these

Define `foldMap` in terms of `foldr`

```
foldMap :: Monoid m => (a -> m) -> t a -> m
foldMap f xs = foldr (\el i -> f el <> i) mempty xs
```

Trickier. Define `foldr` in terms of `foldMap`

```
foldr :: (a -> b -> b) -> b -> t a -> b
```


20

DivideList is Foldable

```
instance Foldable DivideList where
    -- Monoid m => (a -> m) -> DivideList a -> m
    -- (a -> MergesortList a) -> DivideList a ->
                                        MergesortList a
    foldMap f xs =
        case divide xs of
            (DL [], DL []) -> mempty
            (DL [], DL [x]) -> f x
            (dl1, dl2) -> (foldMap f dl1) <> (foldMap f dl2)
    -- vs.

    foldMap' f (DL []) = mempty
    foldMap' f (DL (x:xs)) = f x <> foldMap' f (DL xs)

mergeSort xs = toList $ foldMap singleton (DL xs)
```


Generic Functions over Foldable Types

- What is the benefit of Foldable?

```
average :: [Int] -> Int
average ns = sum ns `div` length ns
```

```
average :: Foldable t = > t Int -> Int
average ns = sum ns `div` length ns

> average [1..10]
5
> average (Node (Leaf 1) (Leaf 3))
2
```



```
-- Like and on list, ands all values in foldable
and :: Foldable t => t Bool -> Bool
and = getAll . foldMap All

-- Like all on list, but acts on foldable
all :: Foldable t => (a -> Bool) -> t a -> Bool
all p = getAll . foldMap (All . p)


and [True,False,True]
>
and (Node (Leaf True) (Leaf False)
>
and (List [List [Atom True, List []], Atom False])
>
```

Can apply on lists, trees and NL-trees!


```
-- Like or on list, ors all values in foldable
or :: Foldable t => t Bool -> Bool
or = getAny . foldMap Any

-- Like all on list, but acts on foldable
any :: Foldable t => (a -> Bool) -> t a -> Bool
any p = getAny . foldMap (Any . p)


or [True,False,True]
>
or (Node (Leaf True) (Leaf False)
>
or (List [List [Atom True, List []], Atom False])
>
```

Can apply on lists, trees, and NL-lists!

```
-- E.g., concat ["ab","cd","ef"] yields "abcdef"
concat :: Foldable t => t [a] -> [a]
concat =  foldMap id
```


Generic Monadic Functions

- Define some useful generic functions. What does `map` do?

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f xs =
```

- What are some use cases with `Maybe` and `List` monads?
- Two use cases are shown in slides. Add an additional use case for `mapM`

Exercise

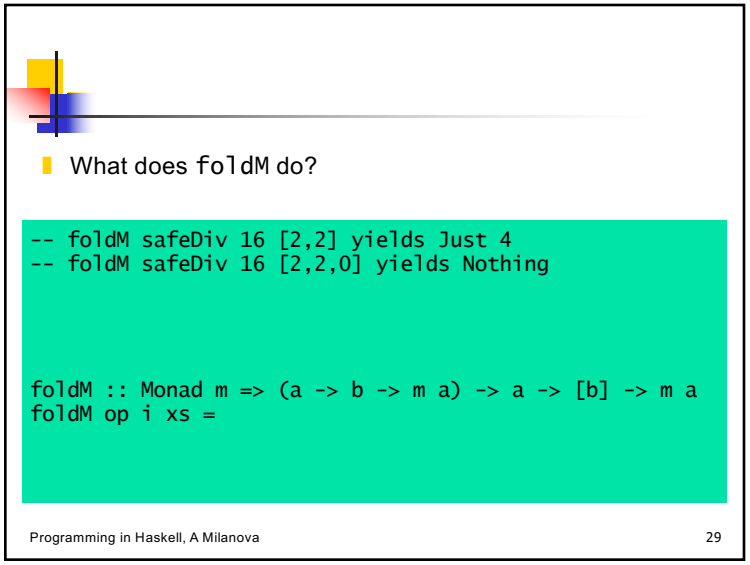
- Define `fun1` in terms of `mapM` and `maybeUpper`:

```
import Data.Char

maybeUpper :: Char -> Maybe Char
maybeUpper x = if isAlpha x
                then Just (toUpper x)
                else Nothing

-- fun1 "abAcd" yields Just "ABACD"
-- fun1 "ab1cd" yields Nothing
fun1 :: String -> Maybe String
fun1    =
```


Exercise

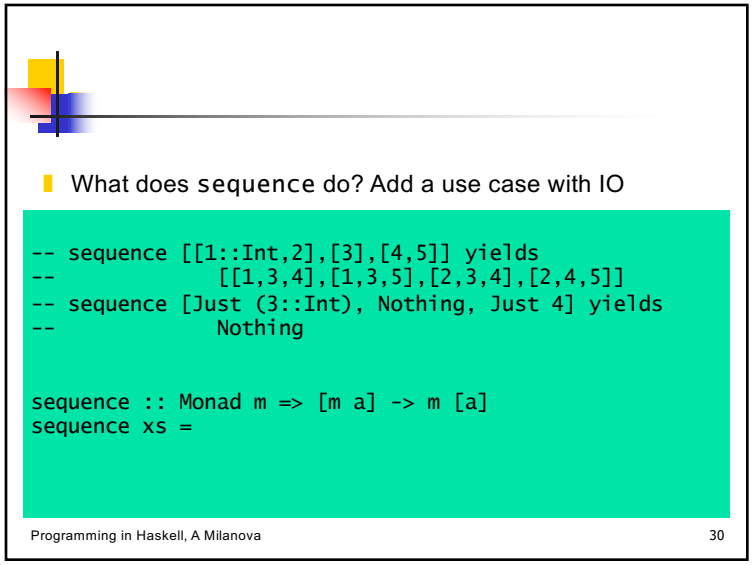
- Define `fun2` in terms of `mapM` and `onlyUpper`:

```
import Data.Char

onlyUpper :: String -> String
onlyUpper = filter isUpper

-- fun2 ["ABC", "Ac"] yields ["AA", "BA", "CA"]
-- fun2 ["ABC", "ac"] yields []
fun2 :: [String] -> [String]
fun2    =
```



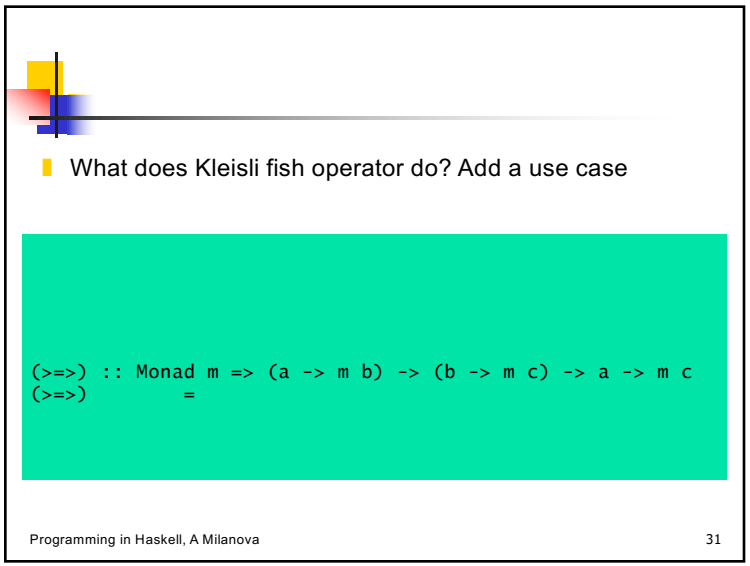
- What does `foldM` do?

```
-- foldM safeDiv 16 [2,2] yields Just 4
-- foldM safeDiv 16 [2,2,0] yields Nothing



foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
foldM op i xs =
```

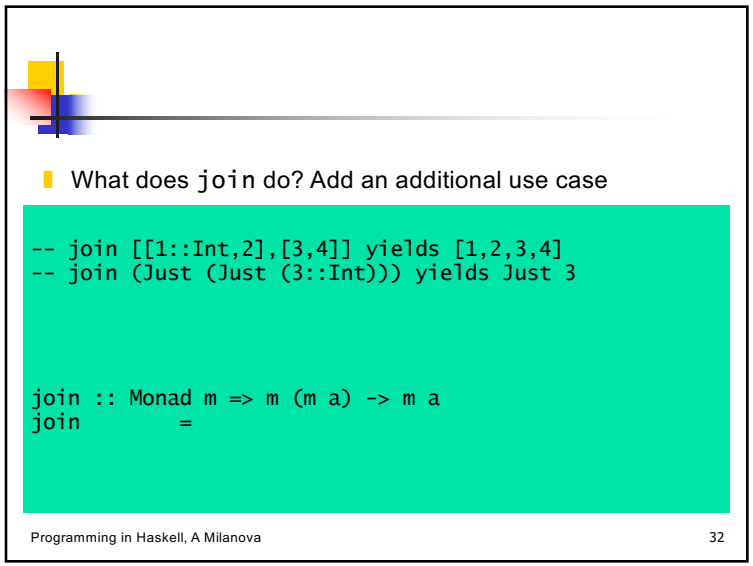
- What does `sequence` do? Add a use case with IO

```
-- sequence [[1::Int,2],[3],[4,5]] yields
--             [[1,3,4],[1,3,5],[2,3,4],[2,4,5]]
-- sequence [Just (3::Int), Nothing, Just 4] yields
--             Nothing


sequence :: Monad m => [m a] -> m [a]
sequence xs =
```

- What does Kleisli fish operator do? Add a use case

```



(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
(>=>)        =
```

- What does `join` do? Add an additional use case

```
-- join [[1::Int,2],[3,4]] yields [1,2,3,4]
-- join (Just (Just (3::Int))) yields Just 3



join :: Monad m => m (m a) -> m a
join        =
```

- What does `liftM` function do?

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM    =
```

- What does `liftM2` do?

```
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
liftM2    =
```