Lecture 5 – Functor, Applicative functor, and Monoid

- Functor class defines

```
fmap    :: (a -> b) -> f a -> f b
(<$>)   = fmap
```

- Applicative class defines

```
pure    :: a -> f a
(<*>)   :: f (a -> b) -> f a -> f b
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

- Monoid class defines

```
mempty  :: a
mappend :: a -> a -> a
mconcat :: [a] -> a
```

Semigroup and Monoid

- Semigroup defines

```haskell
(<>)    :: a -> a -> a

-- (a <> b) <> c = a <> (b <> c)
```

- Monoid is a subclass of Semigroup.

```haskell
class Semigroup a => Monoid a where
  mempty  :: a             -- only mempty is required
  mappend :: a -> a -> a -- mappend = (<>)
  mconcat :: [a] -> a      -- mconcat = foldr (<>) mempty
```

Semigroup and Monoid

- Monoid

```haskell
mempty  :: a               -- only mempty is required
(<>)    :: a -> a -> a
mconcat :: [a] -> a        -- mconcat = foldr (<>) mempty
```

- Ordering is an instance of Monoid.

```haskell
data Ordering = LT | EQ | GT

instance Semigroup Ordering where
  LT <> _ = LT
  EQ <> x = x
  GT <> _ = GT

instance Monoid Ordering where
  mempty = EQ
```

Semigroup and Monoid

- Name type

```haskell
data Name = Name {firstName::String, lastName::String}
                  deriving (Show, Eq)

instance Ord Name where
  n1 `compare` n2 =
      (lastName n1  `compare` lastName n2) <>
      (firstName n1 `compare` firstName n2)
```

- Ordering of name is based on last name and then first name

```haskell
x = Name "Adam" "Smith"
y = Name "Adam" "Berger"
z = Name "John" "Smith"
x < y  -- False
x < z  -- True
```

Semigroup and Monoid

- Variable-length name type

```
data Name = Name [String] deriving (Show, Eq)

instance Ord Name where
  Name n1 `compare` Name n2 = mconcat $ map (uncurry compare)
                                      $ reverse n1 `zip'` reverse n2
      where zip' a [] = zip a blank
            zip' [] a = zip blank a
            zip' (x:xs) (y:ys) = (x,y) : zip' xs ys
            blank = repeat ""
```

- Ordering of name is based on reverse order of names

```
x = Name ["Adam", "Gold", "Smith"]
y = Name ["Adam", "Black", "Smith"]
z = Name ["Gold", "Smith"]
w = Name ["Madonna"]
x < y  -- False
x < z  -- False
z < w  -- False
quicksort [x,y,z,w]
-- [Name ["Madonna"],Name ["Adam","Black","Smith"],
--  Name ["Gold","Smith"],Name ["Adam","Gold","Smith"]]
```

Semigroup and Monoid

- List is an instance of Monoid

```
instance Semigroup [a] where
  <> = ++

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

- Flatten list of lists

```
mconcat [[1,2,3], [4], [], [5,6]]
-- [1,2,3,4,5,6]
```

Semigroup and Monoid

- 'Maybe a' is a Monoid (if 'a' is a Semigroup)

```haskell
instance Semigroup a => Semigroup (Maybe a) where
  Just a  <> Just b  = Just $ a <> b
  Nothing <> x       = x
  x       <> Nothing = x

instance Semigroup a => Monoid (Maybe a) where
  mempty = Nothing
```

- Flatten list of maybe lists

```haskell
Just [1,2] <> Just [3] <> Nothing
-- Just [1,2,3]

mconcat [Just [1,2], Just [3], Nothing, Just [4,5]]
-- Just [1,2,3,4,5]
```

Applicative Functors

- Applicative is a functor with a pure and an app method <*>.

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

    (<*>) :: f (a -> b) -> f a -> f b
    (<*>) = liftA2 id

    liftA2 :: (a -> b -> c) -> f a -> f b -> f c
    liftA2 f x = (<*>) (fmap f x)
```

- Maybe is an applicative functor.

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

instance Applicative Maybe where
    pure = Just

    Just f  <*> Just x  = Just $ f x
    Nothing <*> _       = Nothing
    _       <*> Nothing = Nothing
```

Applicative Functors

- Maybe is an instance of Applicative Functor.

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

instance Applicative Maybe where
    pure = Just

    Just f  <*> Just x  = Just $ f x
    Nothing <*> _       = Nothing
    _       <*> Nothing = Nothing
```

```
Just (+ 2) <*> Just 3          -- Just 5
Just (+ 2) <*> Nothing         -- Nothing
Nothing    <*> Just 3          -- Nothing
liftA2 (+) (Just 2) (Just 3) -- Just 5
liftA2 (+) (Just 2) Nothing  -- Nothing
(+) <$> (Just 2) <*> Just 3  -- Just 5
(+) <$> (Just 2) <*> Nothing -- Nothing
```

Applicative Functors

- List is an instance of Applicative Functor.

```haskell
instance Functor [] where
    fmap _ [] = []
    fmap f (x:xs) = f x : fmap f xs

instance Applicative [] where
    pure x = [x]

    fs <*> xs  = [f x | f <- fs, x <- xs]
```

```haskell
(+) <$> [1,2] <*> [10,20] -- [11,21,12,22]

liiftA2 (+) [1,2] [10,20] -- [11,21,12,22]
```

Traversable

- Traversable is a class of data structure that can be traversed from left to right, performing an action on each element.

```
class (Functor t, Foldable t) => Traversable (t :: * -> *)
 where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  traverse f = sequenceA . fmap f

  sequenceA :: Applicative f => t (f a) -> f (t a)
  sequenceA = traverse id
```

- List is an instance of Traversable

```
instance Traversable [] where
  sequenceA [] = pure []
  sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

Traversable

- Traversable is a class of data structure that can be traversed from left to right, performing an action on each element.

```
class (Functor t, Foldable t) => Traversable (t :: * -> *)
 where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  traverse f = sequenceA . fmap f

  sequenceA :: Applicative f => t (f a) -> f (t a)
  sequenceA = traverse id
```

```
sequenceA [Just 1, Just 2, Just 3]           -- Just [1,2,3]

sequenceA [Just 1, Just 2, Nothing, Just 3] -- Nothing

sequenceA [[1,2], [10,20]] -- [[1,10],[1,20],[2,10],[2,20]]

sequenceA [getLine, getLine, getLine]
-- hi
-- hello
-- how are you
-- ["hi","hello","how are you"]
```

newtype

- **newtype** defines a *new* type from an existing type. It is barely more than a type alias. Unlike ADT defined using *data*, there can be only one case.

```
newtype Calendar = Events {getEvents :: [Event]}
                   deriving (Show, Read)

data Event = Tag String Calendar | One MeetTime
                   deriving (Show, Read)
```

- *getEvents* is a function to unwrap the wrapped event list.

```
 getEvents :: Calendar -> [Event]
```

- The wrapped event list is not evaluated until needed. ADT defined using *data* is eager – wrapped data is examined in order to tell which case it belongs to.

Calendar is an instance of Monoid

```
newtype Calendar = Events {getEvents :: [Event]}
data Event = Tag String Calendar | One MeetTime

instance Semigroup Calendar where
  (Events l1) <> (Events l2) = Events $ merge l1 l2

    where merge (e1 @ (Tag s1 c1) : rs1) (e2 @ (Tag s2 c2) : rs2)
            | s1 == s2 = (Tag s1 (c1 <> c2)) : rs
            | s1 < s2 = e1 : e2 : rs
            | s1 > s2 = e2 : e1 : rs
            where rs = merge rs1 rs2

          merge (Tag s c : rs) l = Tag s c : merge rs l
          merge l (Tag s c : rs) = Tag s c : merge l rs
          merge [] [] = []

instance Monoid Calendar where
  mempty = Events []

insertEvent calendar tags meet = calendar <> makeEvent tags meet
```