4-2

```
Sum :: [Int] -> Int
Sum [] = 0
Sum (x:xs) = x + Sum xs
```

```
Product :: [Int] -> Int
Product [] = 1
product (x:xs) = x * product xs
```

```
map :: (a->b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

```
sum' :: [Int] -> Int
sum _ [] = 0
sum id (x:xs) = (id x) + sum id xs
```

init

reducer

abstraction ⇒ fold

foldable

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

x acc^{t} acc^{t+1} init

reducer

b

Sum : Monoid(Int, +)

Product : Monoid(Int, *)

Map f : Monoid([], <>)

```
foldr f z [] = z
foldr f z (x:xs) = f x (fold f z xs)
```

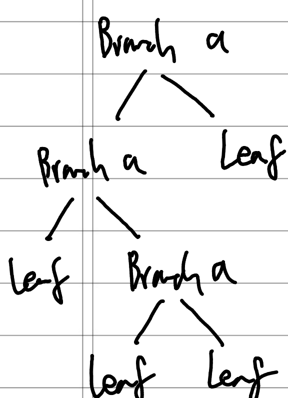
Foldable (Monoid)

```
foldr' :: Foldable m, Monoid b =>
        (a -> b) -> m a -> b
foldr' f  m.empty = b.mempty
foldr' f  m = f (first m) <> (foldr' f (rest m))
```

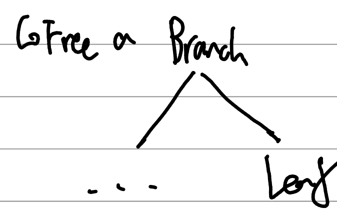
```
class Foldable m a
    first :: m a -> a
    rest  :: m a -> m a
    foldr' :: Monoid b =>
```

instance | List / [ ]
         | Tree : Free, CoFree   e.g. Map, Rose,
                                 BST, DOM

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

data Free f a = | Pure a
                | Free (f (Free a))

data CoFree f a = a :< f(Cofree a)

                = Cofree a f ( Cofree a)

data TreeF a = | Branch a a
               | Branch (TreeF a) a
               | Branch a (TreeF a)
               | Leaf
```



CoFree TreeF a



$[\text{Maybe } a] \xrightarrow{\text{Sequance}} \text{Maybe } [a]$

> example : [Maybe Int]

$[\text{Just } 1, \text{Just } 2, \text{Just } 3] \xrightarrow{\text{Sequence}} \text{Just } [1,2,3]$

Just 1 : Just 2 : Just 3 : Nil

$\text{Sequence Just } 3 \rightarrow \text{Just } \underbrace{[3]}_{3 : \text{Nil}}$

$\text{Map } \underbrace{[\,]}\ \text{Just } 3 = \text{Just } [3]$

list pure ← (· : Nil) :: a → [a]

(as Applicative)

(Maybe wrong*)

Just [3] ? Just 2 → Just (2 : [3])

Just [2,3] ? Just 1 → Just (1 : [2,3])

structural induction (⇐ partial order)

for $\boxed{\text{len any list} >= 0}$

```
len :: [_] -> Int
len [] = 0
len (_:xs) = 1 + len xs
```

[] < [_] < [_,_] < ...

Functor (type class)

Naive explanation: a "container" with certain "shape"

mapping a function over a functor preserves the "shape".

(a -> b) (f a) (f b)

> example: Maybe

```
type Maybe a = Nothing | Just a
map _ Nothing = Nothing
map f (Just x) = Just (f x)
```

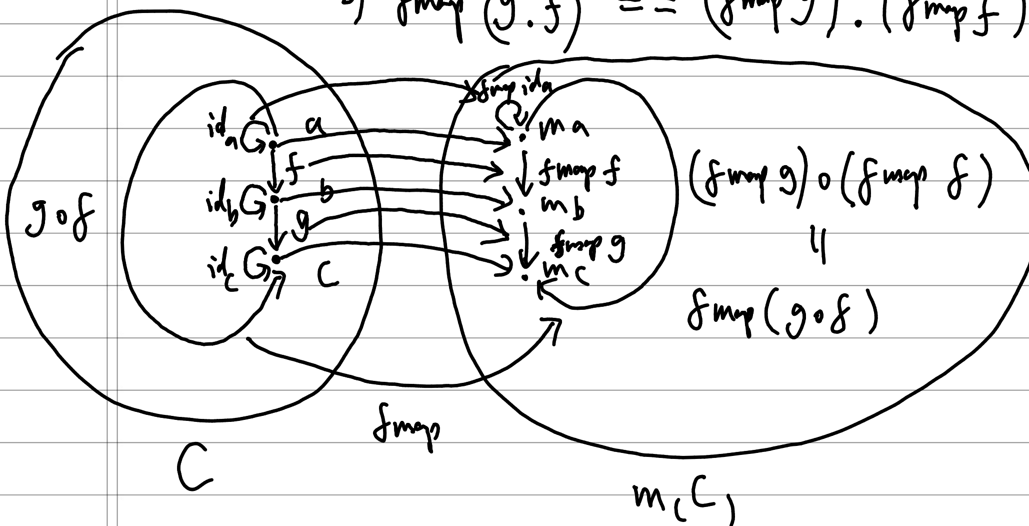
abstract over a class of types that have this "structural preserving" property takes an unary type constructer (* -> *)

class Functor m where

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

<u>Laws</u> 1) fmap id == id

2) fmap (g . f) == (fmap g) . (fmap f)




```
data Tree a = | Leaf
              | Branch (Tree a) a (Tree a)
instance Functor Tree where
  fmap _ Leaf = Leaf
  fmap f (Branch l x r) =
Branch (fmap f l) (f x) (fmap f r)
```

Identity Functor (an identity "function" in Types)

data Identity a = Identity a

instance Functor Identity where
    fmap f Identity x = Identity (f x)

Basically, putting a prefix on the name of a Type.

A counter example to the "container" analogy

able to 'extract' / 'inspect' the values inside (CoMonad)

"environment"

data Reader e a = Reader (e -> a)

Reader type constructor is a binary type constructor (* -> * -> *) | Reader data constructor wraps a function

instance Functor (Reader e) where
    fmap :: (a -> b) -> (Reader e) a -> (Reader e) b
    fmap g (Reader f) = Reader (g . f)

partial application in type constructor, given type e, (Reader e) is unary (* -> *)

unwrap the function encapsulated in Reader (e -> a)

$(a \to b) \circ (e \to a) = e \to b$

(Reader e) b

a level of abstraction

(Higher-order) kind constructor (not full-fledged in Haskell, may have compiler extension available)

type constructor

a level of abstraction

function in type-level

input: type variables output: type

data constructor (same as regular functions on values)

function in value-level

input: (value) variables output: value