1 Monads

1.1 Functors

Recall our use of *higher-order functions* to abstract code patterns for *lists*. In particular, we made use of the `map` higher-order function to

- render the values of a list,
- square the values of a list,
- and more.

What about trees?

- Let's suppose we wanted to render the values of a *tree*, where a tree is defined by

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

  So, to render the values of a tree, we can do

  ```
  showTree :: Tree Int -> Tree String
  showTree Leaf         = Leaf
  showTree (Node v l r) = Node (show x) (showTree l) (showTree r)
  ```

- Now, let's suppose we wanted to square the values of a tree. We can use the same pattern as used in the previous part, changing the return type and the return values appropriately.

We can write a generalization of this by writing a `map` function for trees.

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Leaf          = Leaf
mapTree f (Node v l r)  = ...
```

But, observe the following:

```
type List a = [a]
mapList :: (a -> b) -> List a -> List b    -- List
mapTree :: (a -> b) -> Tree a -> Tree b    -- Tree
```

Notice how we have essentially the same signature for both `Lists` and `Trees`.

1.1.1 Class for Mapping

We can make a typeclass to model mapping over some datatypes (not all datatypes support mapping over them).

```
class Functor t where
    fmap :: (a -> b) -> t a -> t b
```

Then, we can do

```
instance Functor [] where
    fmap = mapList

instance Functor Tree where
    fmap = mapTree
```

1.2 Monads

Consider the following `Expr` data type.

```
data Expr
    = Num   Int
    | Plus  Expr Expr
    | Div   Expr Expr
    deriving (Show)

eval :: Expr -> Int
eval (Num n)        = n
eval (Plus e1 e2)   = eval e1   +   eval e2
eval (Div  e1 e2)   = eval e1 `div` eval e2
```

So, for example, we can run

```
$ eval (Div (Num 6) (Num 2))
3
```

But, if we were in an interpreter like Nano, the following can crash Nano:

```
$ eval (Div (Num 6) (Num 0))
*** Exception: divide by zero
```

Let us introduce a new data type which will handle errors for us.

```
data Result a
    = Error String
    | Value a
```

So, instead of returning `Int`, this will now return `Result Int`, where

- If a sub-expression has a divide-by-zero, then return `Error ...`.
- If all sub-expressions are safe, then we can return the actual `Value v`.

Therefore,

```
eval :: Expr -> Result Int
eval (Num n)            = Value n
eval (Plus e1 e2)       =
    case eval e1 of
        Error err1  -> Error err1
        Value v1    -> case eval e2 of
                         Error err2  -> Error err2
                         Value v1    -> Value (v1 + v2)

eval (Div e1 e2)        =
    case eval e1 of
        Error err1  -> Error err1
        Value v1    -> case eval e2 of
                         Error err2  -> Error err2
                         Value v1    -> if v2 == 0
                                          then Error ("DBZ: " ++ show e2)
                                          else Value (v1 `div` v2)
```

Note that this works – this doesn't crash the interpreter. However, there is a lot of repetition.