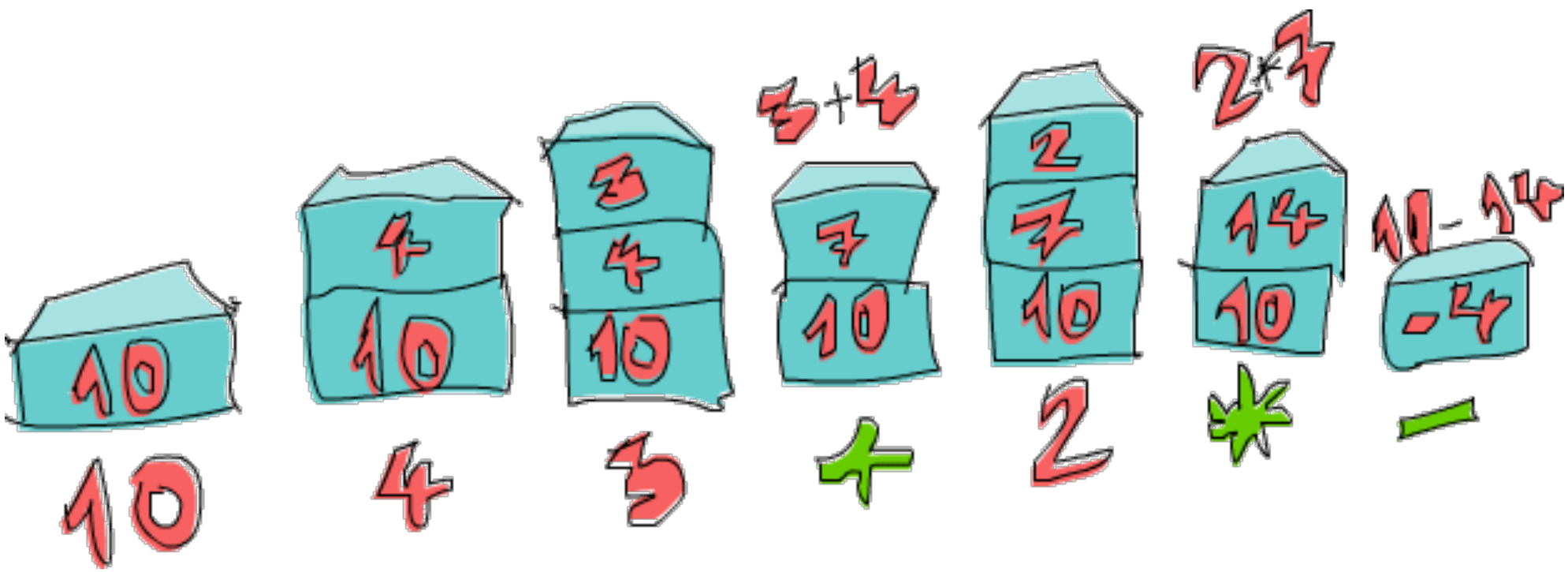
Higher-Order programming

Reverse polish notation



"10 4 3 + 2 * -"

Let’s think about the type

```
solveRPN :: (Num a) => String -> a.
```

Chop the string into words

```
"10 4 3 + 2 * -"
```

```
["10","4","3","+","2","*","-"]
```

```
words :: String -> [String]
```

Traverse the list from left to right

```
["10","4","3","+","2","*","-"]
```

We can use a fold !

```
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

An ad-hoc stack

[ ]

Let’s use a list

top of the list in front

Let’s start !

```
import Data.List

solveRPN :: (Num a) => String -> a
```

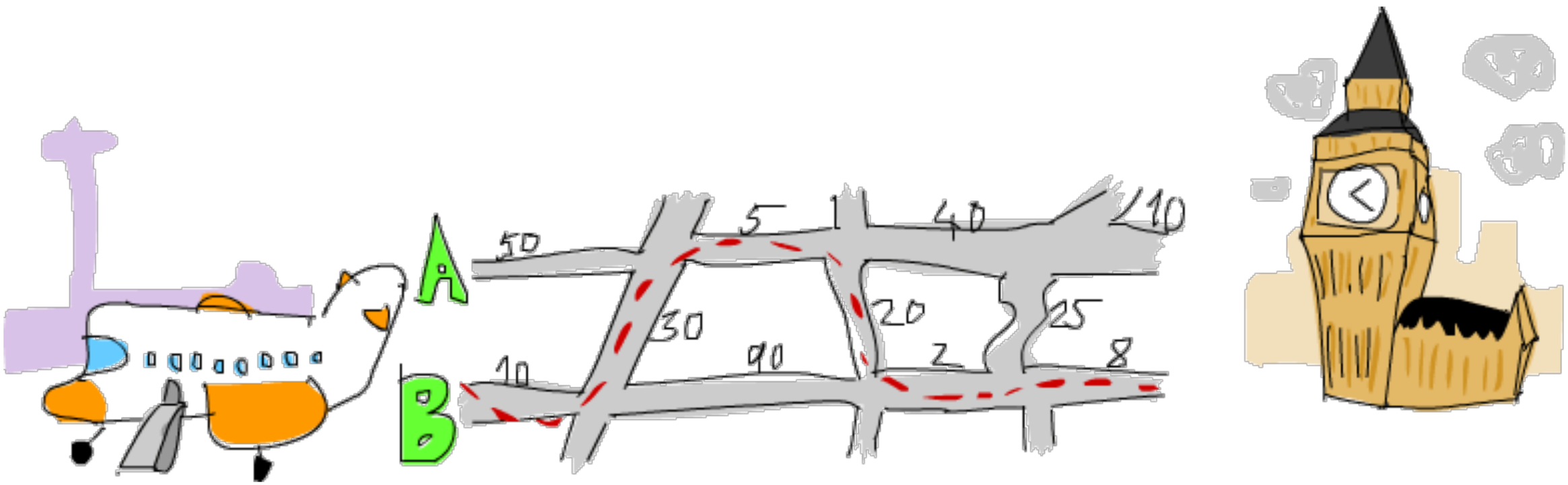
result *traverse* *chop*

```
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction stack item =
```

Finishing up

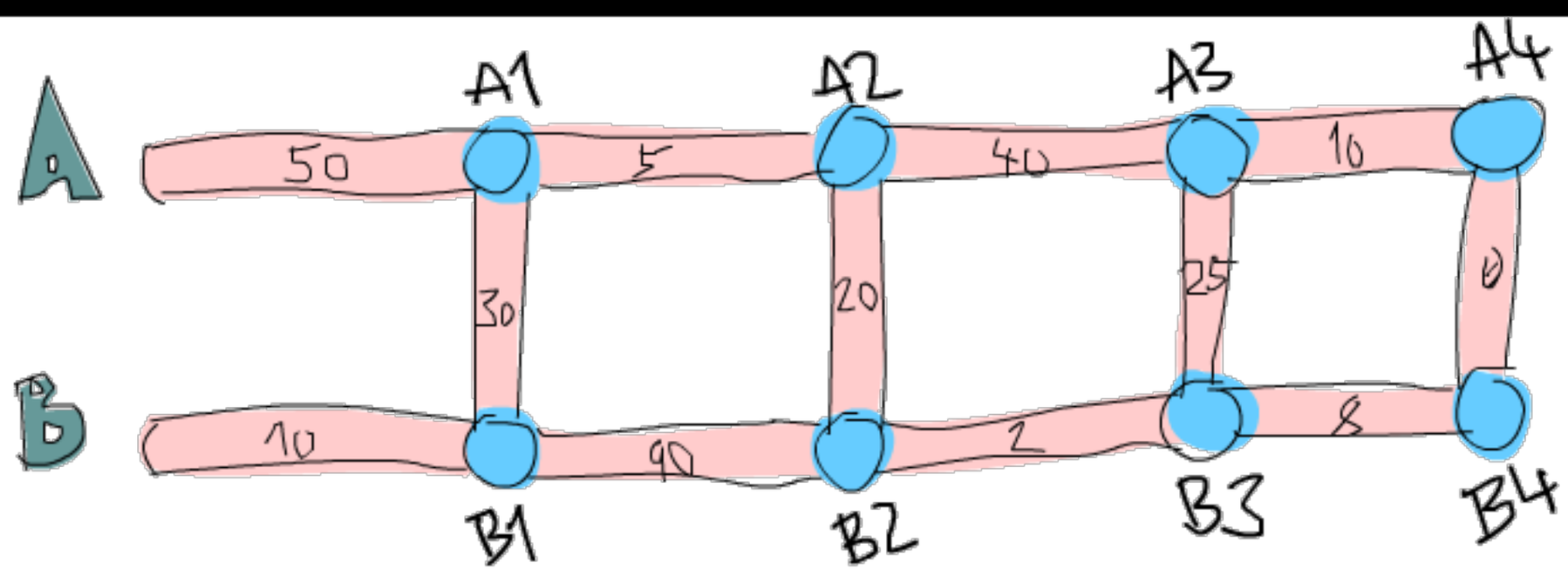
```haskell
import Data.List

solveRPN :: (Num a, Read a) => String -> a
solveRPN = head . foldl foldingFunction [] . words
    where   foldingFunction (x:y:ys) "*" = (x * y):ys
            foldingFunction (x:y:ys) "+" = (x + y):ys
            foldingFunction (x:y:ys) "-" = (y - x):ys
            foldingFunction xs numberString = read numberString:xs
```

Heathrow to London



Data representation

```
50
10
30
5
90
20
40
2
25
10
8
0
```



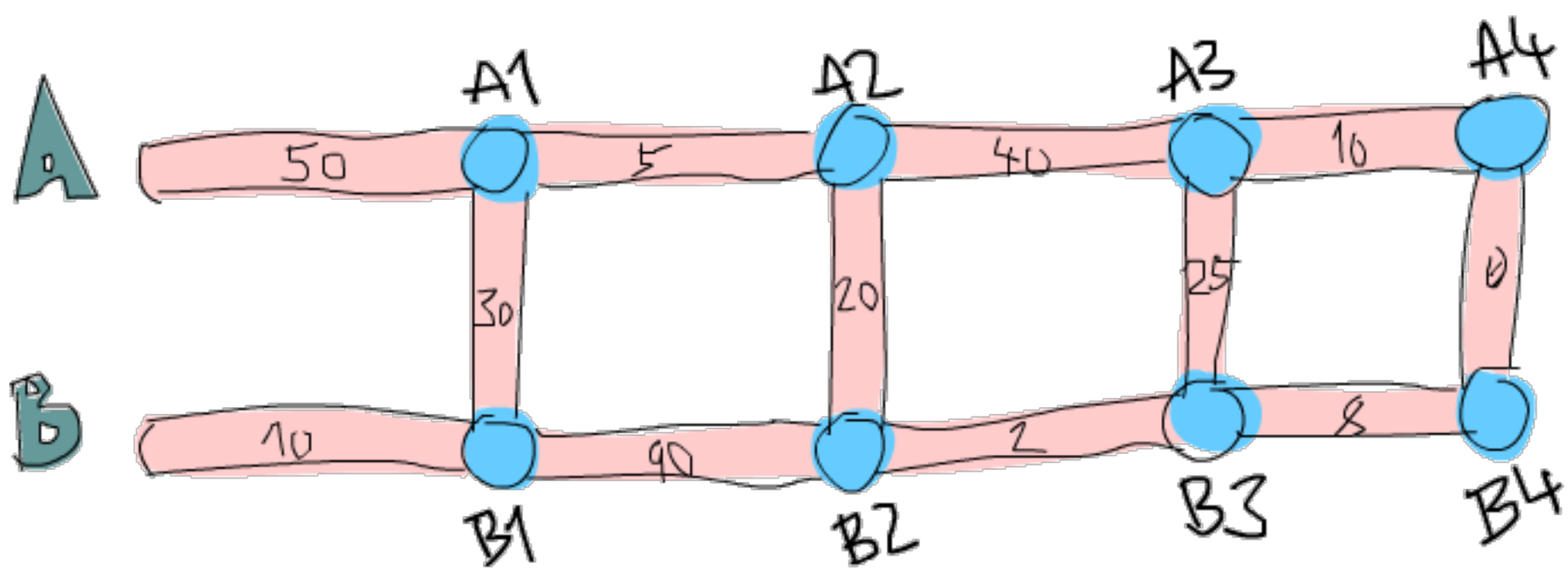
As an Algebraic DataType

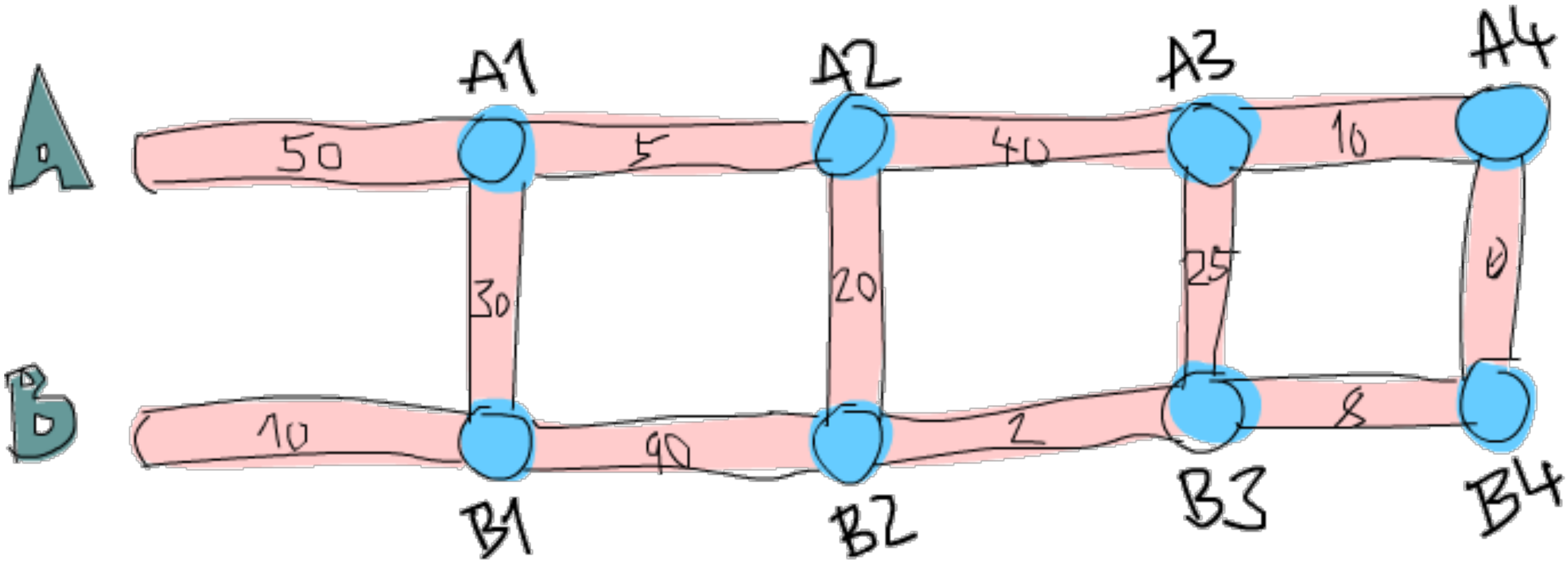
```haskell
data Node     = Node Road Road | EndNode Road
data Road     = Road Int Node
```

```haskell
data Section = Section { getA :: Int,
                         getB :: Int,
                         getC :: Int }
                         deriving (Show)

type RoadSystem = [Section]
```

Heathrow To London

```
heathrowToLondon :: RoadSystem
heathrowToLondon = [Section 50 10 30,
                    Section 5 90 20,
                    Section 40 2 25,
                    Section 10 8 0]
```




A
B
A1
A2
A3
A4
B1
B2
B3
B4
50
5
40
10
30
20
25
0
10
90
2
8

```haskell
data Label = A | B | C deriving (Show)
type Path = [(Label, Int)]

cost :: Path -> Int
cost = sum . map snd

shortest :: Path -> Path -> Path
shortest a b | cost a <= cost b = a
             | otherwise        = b

step l c p = (l,c):p

roadStep :: (Path, Path) -> Section -> (Path, Path)
roadStep (pathA, pathB) (Section a b c) = (newPathToA, newPathToB)
                 where newPathToA = shortest (step A a pathA)
                                             (step C c (step B b pathB))
                       newPathToB = shortest (step B b pathB)
                                             (step C c (step A a pathA))
```

```haskell
roadStep :: (Path, Path) -> Section -> (Path, Path)
roadStep (pathA, pathB) (Section a b c) =
    let priceA = sum $ map snd pathA
        priceB = sum $ map snd pathB
        forwardPriceToA = priceA + a
        crossPriceToA   = priceB + b + c
        forwardPriceToB = priceB + b
        crossPriceToB   = priceA + a + c
        newPathToA = if forwardPriceToA <= crossPriceToA
                     then (A,a):pathA
                     else (C,c):(B,b):pathB
        newPathToB = if forwardPriceToB <= crossPriceToB
                     then (B,b):pathB
                     else (C,c):(A,a):pathA
    in  (newPathToA, newPathToB)
```

Optimal path

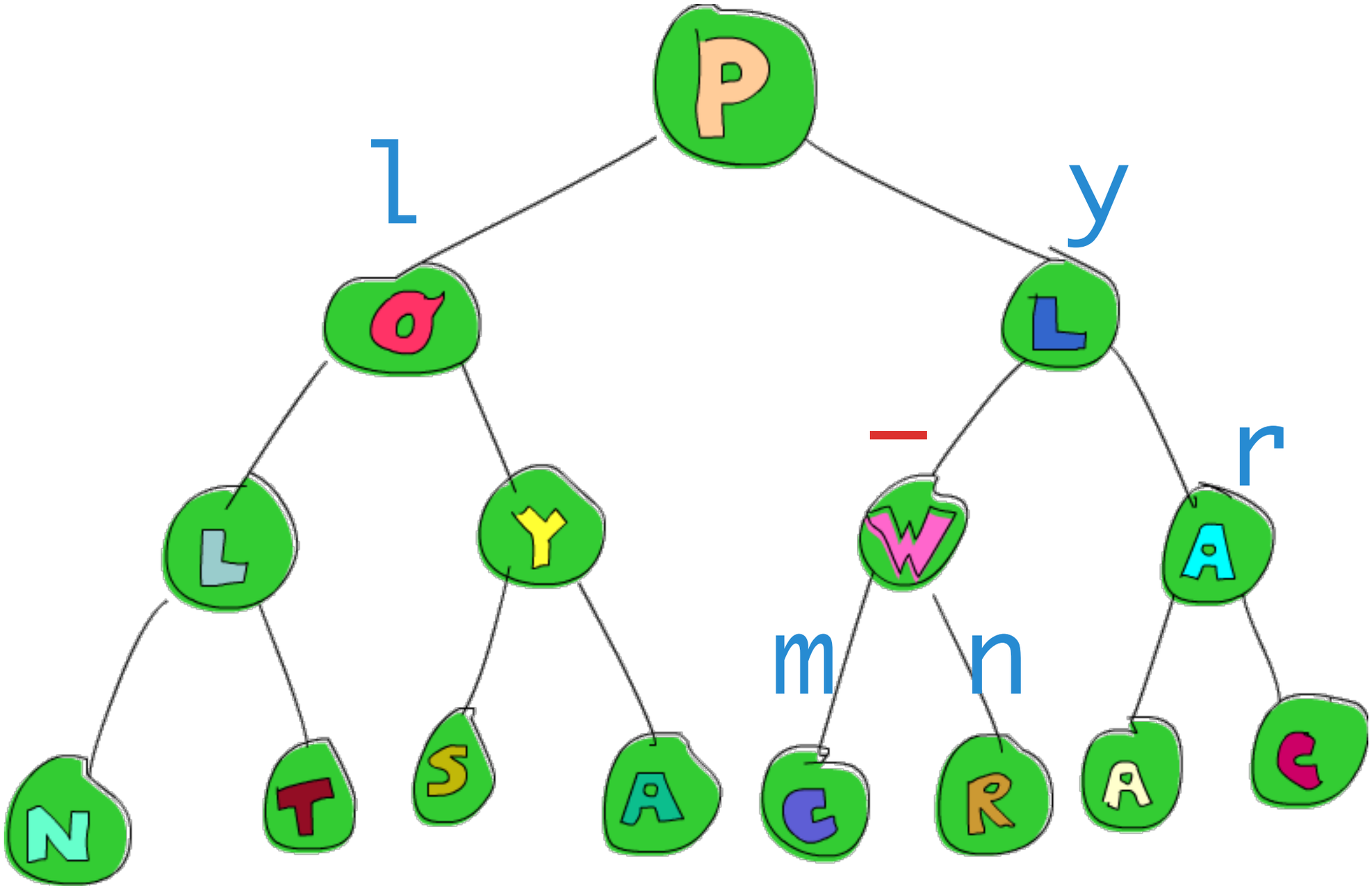
```
optimalPath :: RoadSystem -> Path
optimalPath roadSystem =  reverse $ shortest bestAPath bestBPath
      where (bestAPath, bestBPath) = foldl roadStep ([],[]) roadSystem
```

Reading from file

```haskell
groupsOf :: Int -> [a] -> [[a]]
groupsOf 0 _ = undefined
groupsOf _ [] = []
groupsOf n xs = take n xs : groupsOf n (drop n xs)

main = do
    contents <- getContents
    let threes      = groupsOf 3 (map read $ lines contents)
        roadSystem = map (\[a,b,c] -> Section a b c) threes
        path        = optimalPath roadSystem
        pathString = concat $ map (show . fst) path
        pathPrice  = sum $ map snd path
    putStrLn $ "The best path to take is: " ++ pathString
    putStrLn $ "The price is: " ++ show pathPrice
```

Zippers

```haskell
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)


freeTree :: Tree Char
freeTree =
    Node 'P'
        (Node 'O'
            (Node 'L'
                (Node 'N' Empty Empty)
                (Node 'T' Empty Empty)
            )
            (Node 'Y'
                (Node 'S' Empty Empty)
                (Node 'A' Empty Empty)
            )
        )
        (Node 'L'
            (Node 'W'
                (Node 'C' Empty Empty)
                (Node 'R' Empty Empty)
            )
            (Node 'A'
                (Node 'A' Empty Empty)
                (Node 'C' Empty Empty)
            )
        )
```




```
changeToP :: Tree Char -> Tree Char
changeToP (Node x l (Node y (Node _ m n) r)) = Node x l (Node y (Node 'P' m n) r)
```

Towards Zippers

```haskell
data Direction  = L | R deriving (Show)
type Directions = [Direction]

changeToP :: Directions-> Tree Char -> Tree Char
changeToP (L:ds) (Node x l r) = Node x (changeToP ds l) r
changeToP (R:ds) (Node x l r) = Node x l (changeToP ds r)
changeToP []     (Node _ l r) = Node 'P' l r
```

Accessing

```haskell
elemAt :: Directions -> Tree a -> a
elemAt (L:ds) (Node _ l _) = elemAt ds l
elemAt (R:ds) (Node _ _ r) = elemAt ds r
elemAt [] (Node x _ _) = x
```

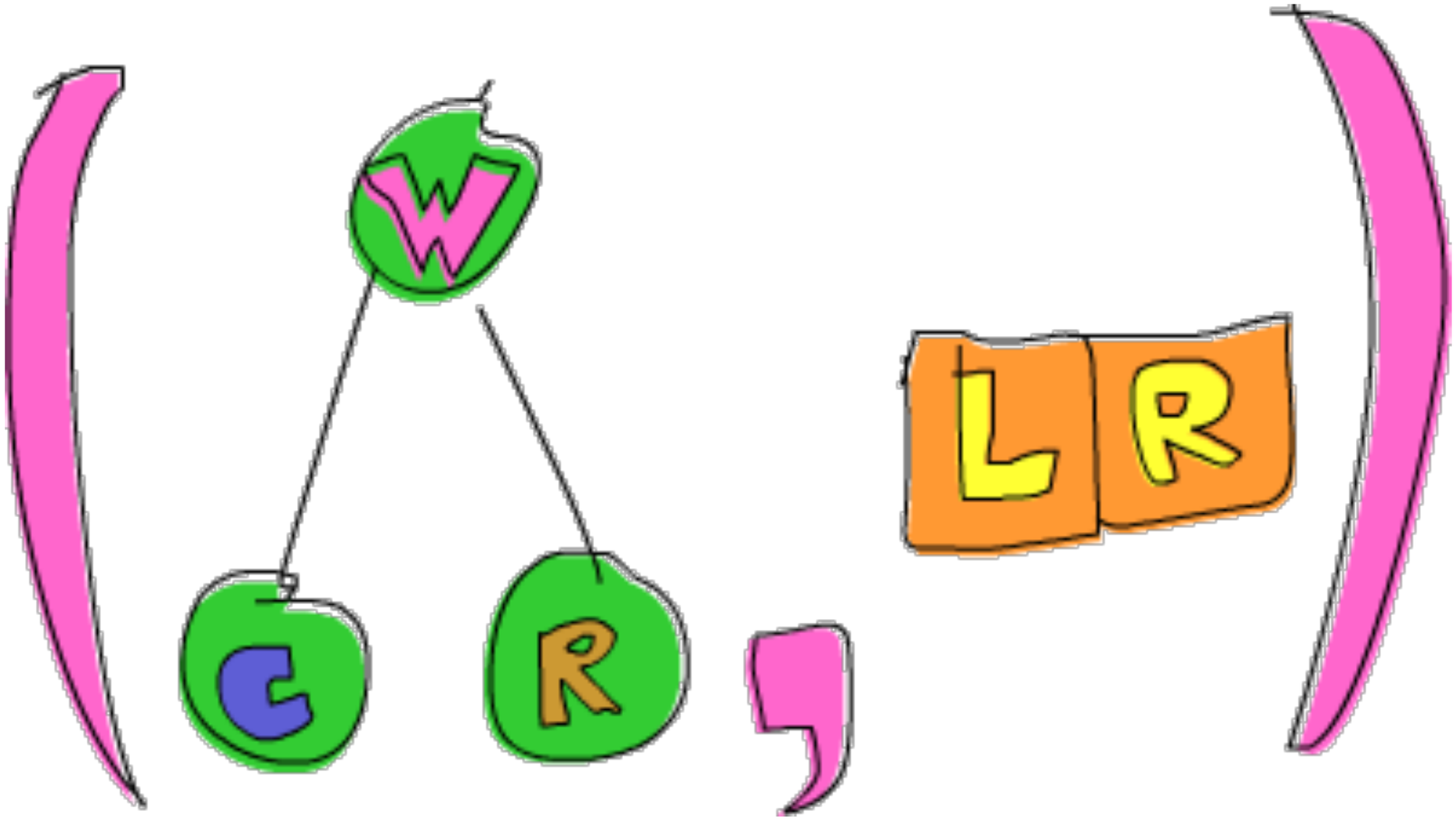
Using our new abstractions

```
ghci> let newTree = changeToP [R,L] freeTree
ghci> elemAt [R,L] newTree
'P'
```

Breadcrumbs

```haskell
type Breadcrumbs = [Direction]

goLeft :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goLeft (Node _ l _, bs) = (l, L:bs)

goRight :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goRight (Node _ _ r, bs) = (r, R:bs)
```

```
ghci> goLeft (goRight (freeTree, []))
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```




```
ghci> goLeft (goRight (freeTree, []))
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```

Breadcrumbs++

```haskell
data Crumb a = LeftCrumb  a (Tree a) |
               RightCrumb a (Tree a)
               deriving (Show)

type Breadcrumbs a = [Crumb a]

goLeft :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)

goRight :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goRight (Node x l r, bs) = (r, RightCrumb x l:bs)


goUp :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```

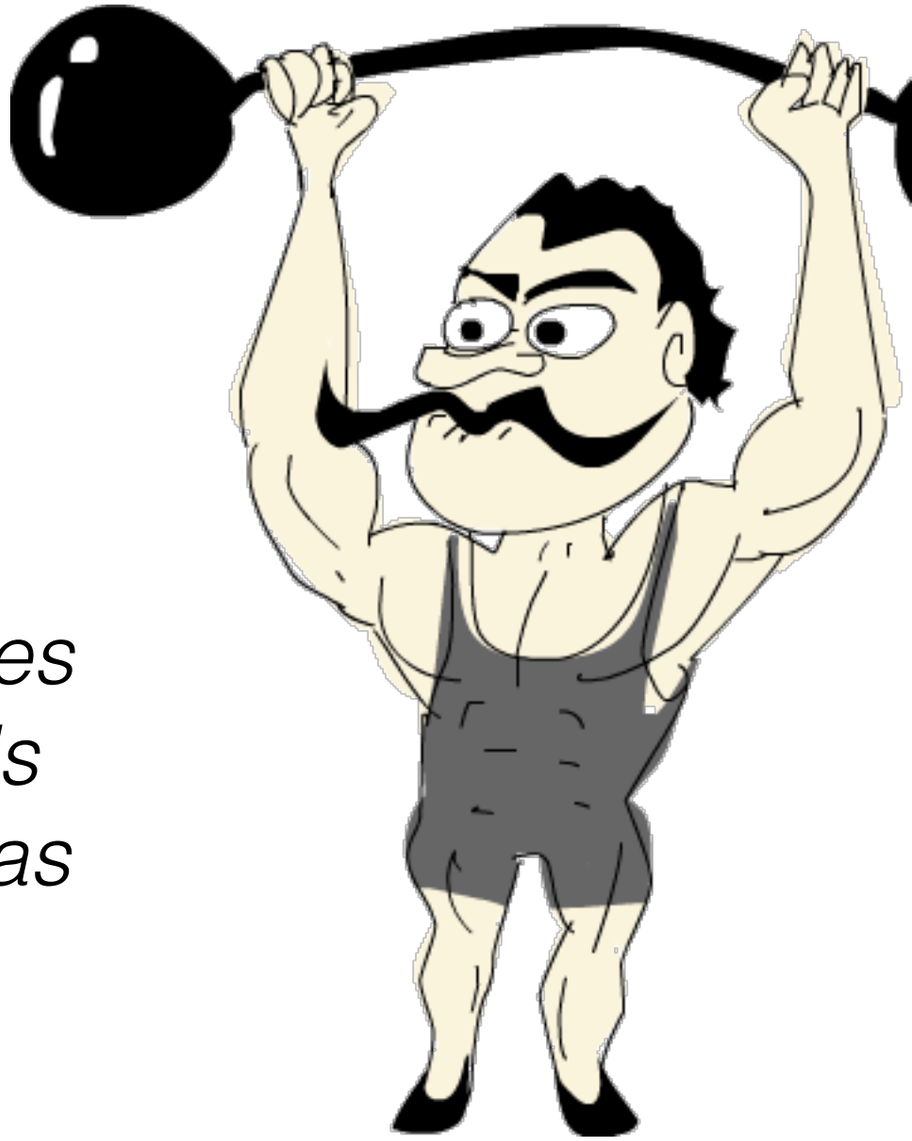
Zipper for Tree

```haskell
type Zipper a = (Tree a, Breadcrumbs a)

modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Node x l r, bs) = (Node (f x) l r, bs)
modify f (Empty, bs) = (Empty, bs)
```

Another view on functors

Lifting

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



We can think of fmap as a function that takes a function and returns a new function that's just like the old one, only it takes a functor as a parameter and returns a functor as the result. ***This is called lifting a function.***

Lifting

```
ghci> :t fmap (*2)
fmap (*2) :: (Num a, Functor f) => f a -> f a
ghci> :t fmap (replicate 3)
fmap (replicate 3) :: (Functor f) => f a -> f [a]
```

```
ghci> let lr = fmap (replicate 3)

ghci> lr  [1,2,3,4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]

ghci> lr  (Just 4)
Just [4,4,4]

ghci> lr (Right "blah")
Right ["blah","blah","blah"]

ghci> lr Nothing
Nothing

ghci> lr (Left "foo")
Left "foo"
```

IO as Functor

```haskell
instance Functor IO where
    fmap f action = do
                    result <- action
                    return (f result)
```

IO Pattern

```
main = do line <- getLine
          let line' = reverse line
          putStrLn $ "You said " ++ line' ++ " backwards!"
          putStrLn $ "Yes, you really said" ++ line' ++ " backwards!"
```

```
main = do line <- fmap reverse getLine
          putStrLn $ "You said " ++ line ++ " backwards!"
          putStrLn $ "Yes, you really said" ++ line ++ " backwards!"
```

Monad Transformers

https://github.com/kqr/gists/blob/master/articles/gentle-introduction-monad-transformers.md

Login

```haskell
data LoginError = InvalidEmail
  deriving Show


getDomain :: Text -> Either LoginError Text
getDomain email =
  case splitOn "@" email of
    [name, domain] -> Right domain
    _              -> Left InvalidEmail
```

Print result

```haskell
printResult' :: Either LoginError Text -> IO ()
printResult' domain =
  case domain of
    Right text        -> putStrLn (append "Domain: " text)
    Left InvalidEmail -> putStrLn "ERROR: Invalid domain"
```

Print result nicer

```
either :: (LoginError -> Text) ->
          (Text -> Text)  ->
          (Either LoginError Text -> Text)
```

```
printResult :: Either LoginError Text -> IO ()
printResult = putStrLn . either
  (const "ERROR: Invalid domain")
  (append "Domain: ")
```

With side effects

```haskell
getToken :: IO (Either LoginError Text)
getToken = do
  putStrLn "Enter email address:"
  email <- getLine
  return (getDomain email)
```

Authentication

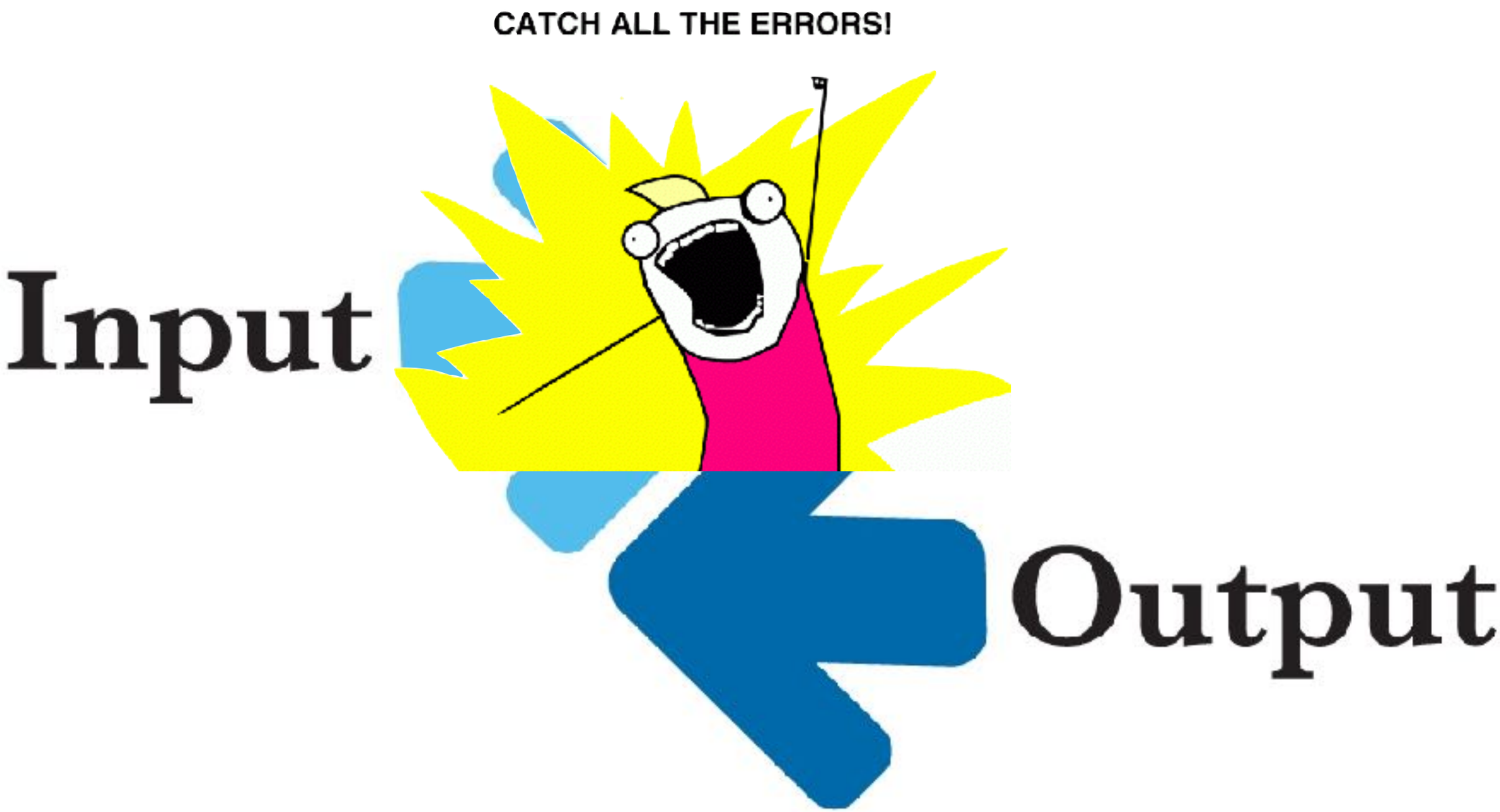
```
users :: Map Text Text
users = Map.fromList [("example.com", “qwerty123"),
                      ("localhost", "password")]
```

```
LoginError = InvalidEmail
           | NoSuchUser
           | WrongPassword
 deriving Show
```

Login

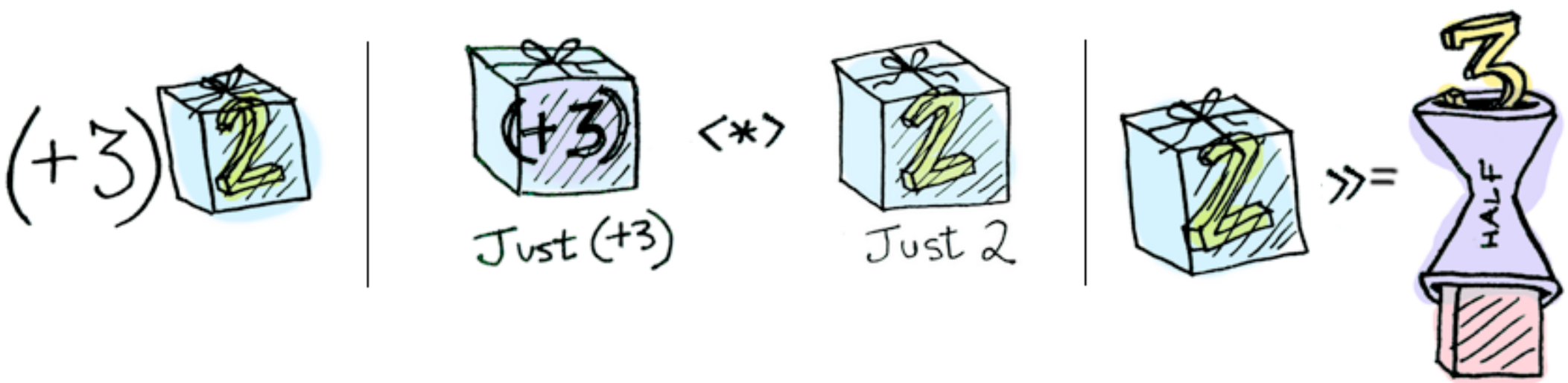
```haskell
userLogin :: IO (Either LoginError Text)
userLogin = do
  token <- getToken

  case token of
    Right domain ->
      case Map.lookup domain users of
        Just userpw -> do
          putStrLn "Enter password:"
          password <- getLine

          if userpw == password
             then return token

             else return (Left WrongPassword)
        Nothing -> return (Left NoSuchUser)
    left -> return left
```

Clash



Rolling our own monad

```haskell
data EitherIO e a = EitherIO {
    runEitherIO :: IO (Either e a)
}
```




```haskell
Functor f
```

```haskell
=> Applicative f
```

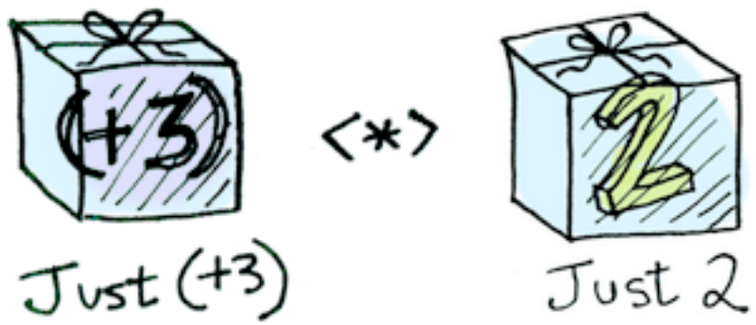
```haskell
=> Monad f
```

Functor

```
instance Functor (EitherIO e) where
  fmap f ex = wrapped
    where
      unwrapped = runEitherIO ex
      fmapped   = fmap (fmap f) unwrapped
      wrapped   = EitherIO fmapped
```

Applicative

```
instance Applicative (EitherIO e) where
  pure    = EitherIO . return . Right
  f <*> x = EitherIO $ (fmap (<*>) (runEitherIO f))
                       <*>
                       (runEitherIO x)
```



Monad

```
instance Monad (EitherIO e) where
  return  = pure
  x >>= f = EitherIO $ runEitherIO x
                       >>=
                       either (return . Left)
                              (runEitherIO . f)
```

Testing time !

```haskell
getToken :: EitherIO LoginError Text
getToken = do
  putStrLn "Enter email address: "
  input <- getLine
  return (getDomain input)
```

Debug time !

```haskell
getToken :: EitherIO LoginError Text
getToken = do
  putStrLn "Enter email address: "
  input <- getLine
  return (getDomain input)
```

- `putStrLn "email"` returns `IO ()`,
  but we want `EitherIO LoginError ()`
- `getLine` returns `IO Text`,
  we want `EitherIO LoginError Text`.
- `getDomain input` returns `Either LoginError Text`,
  we want `EitherIO LoginError Text`.

Either e a -> EitherIO e a

```
return   :: Either e a -> IO (Either e a)
EitherIO :: IO (Either e a) -> EitherIO e a
```

```
EitherIO (return (getDomain input))
```

IO a -> EitherIO e a

```
fmap Right :: IO a -> IO (Either e a)
```

Let’s rewrite our program

```haskell
getToken :: EitherIO LoginError Text
getToken = do
  EitherIO (fmap Right (putStrLn "Enter email address:"))
  input <- EitherIO (fmap Right getLine)
  EitherIO (return (getDomain input))
```

Lifting

```haskell
liftEither :: Either e a -> EitherIO e a
liftEither x = EitherIO (return x)

liftIO :: IO a -> EitherIO e a
liftIO x = EitherIO (fmap Right x)
```

Take 2

```haskell
getToken :: EitherIO LoginError Text
getToken = do
  liftIO (putStrLn "Enter email address:")
  input <- liftIO getLine
  liftEither (getDomain input)
```

User Login Revised

```haskell
userLogin :: EitherIO LoginError Text
userLogin = do
  token      <- getToken
  userpw     <- maybe (liftEither (Left NoSuchUser))
                 return (Map.lookup token users)
  password   <- liftIO (T.putStrLn "Enter your password:" >> T.getLine)

  if userpw == password
     then return token
     else liftEither (Left WrongPassword)
```

User Login Revised

```haskell
userLogin :: EitherIO LoginError Text
userLogin = do
  token      <- getToken
  userpw     <- maybe (liftEither (Left NoSuchUser))
                  return (Map.lookup token users)
  password   <- liftIO (T.putStrLn "Enter your password:" >> T.getLine)

  if userpw == password
     then return token
     else liftEither (Left WrongPassword)
```

Going general !

```haskell
data ExceptT e m a = ExceptT {
    runExceptT :: m (Either e a)
}

instance Functor m => Functor (ExceptT e m) where
  fmap f = ExceptT . fmap (fmap f) . runExceptT

instance Applicative m => Applicative (ExceptT e m) where
  pure    = ExceptT . pure . Right
  f <*> x = ExceptT $ liftA2 (<*>) (runExceptT f) (runExceptT x)

instance Monad m => Monad (ExceptT e m) where
  return  = ExceptT . return . Right
  x >>= f = ExceptT $ runExceptT x >>= either (return . Left) (runExceptT . f)
```

Second Example

(non ghc 7.10) https://en.wikibooks.org/wiki/Haskell/Monad_transformers

Checking valid passwords

```haskell
-- The validation test could be anything we want it to be.
isValid :: String -> Bool
isValid s = length s >= 8
            && any isAlpha s
            && any isNumber s
            && any isPunctuation s
```

Getting a passphrase

```haskell
getPassphrase :: IO (Maybe String)
getPassphrase = do s <- getLine
                   if isValid s then return $ Just s
                                else return Nothing
```

Getting a passphrase

```haskell
askPassphrase :: IO ()
askPassphrase = do putStrLn "Insert your new passphrase:"
                   maybe_value <- getPassphrase
                   if isJust maybe_value
                     then do putStrLn "Storing in database..."
                     else putStrLn "Passphrase invalid."
```

The Maybe Transformer

```haskell
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

MaybeT Functor

```
instance (Monad m) => Functor (MaybeT m) where
    fmap f  =  MaybeT . (fmap $ fmap f)  . runMaybeT
```

MaybeT Applicative

```
instance (Monad m) => Applicative (MaybeT m) where
    pure    = MaybeT . return . Just
    f <*> x = MaybeT $ (fmap (<*>) (runMaybeT f)) <*> (runMaybeT x)
```

The MaybeT Monad

```haskell
instance Monad m => Monad (MaybeT m) where
    return  = MaybeT . return . Just
    (>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b
    x >>= f = MaybeT $ do maybe_value <- runMaybeT x
                          case maybe_value of
                              Nothing    -> return Nothing
                              Just value -> runMaybeT $ f value
```

Monad Plus

```
instance Monad m => MonadPlus (MaybeT m) where
    mzero     = MaybeT $ return Nothing
    mplus x y = MaybeT $ do maybe_value <- runMaybeT x
                            case maybe_value of
                                Nothing    -> runMaybeT y
                                Just _     -> return maybe_value
```

Examples

```haskell
getPassword :: MaybeT IO String
getPassword = do
    s <- lift getLine
    guard (validPassword s)
    return s
```

Examples

```haskell
askPassphrase :: MaybeT IO ()
askPassphrase = do lift $ putStrLn "Insert your new passphrase:"
                   value <- getValidPassphrase
                   lift $ putStrLn "Storing in database..."
```

```haskell
askPassword :: MaybeT IO ()
askPassword = do lift $ putStrLn "Insert your new password:"
                 value <- msum $ repeat getValidPassphrase
                 lift $ putStrLn "Storing in database..."
```

We did it !!!