Recap: Monads
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a

f =
  m1 >>= \x1 ->
  m2 >>= \x2 ->
  ...
  mn >>= \xn ->
  mFinal

f = do
  x1 <- m1
  x2 <- m2
  ...
  xn <- mn
  mFinal
What do you do when a computation can fail?
NoMethodError in Rms::Properties#index

Showing: /home/seems/hub/index.html/app/views/rms/properties/index.html.erb where line #29 raised:

undefined method 'name' for nil:NilClass

Extracted source (around line #29):

```html
<tr>
  <td>&lt;%= property.seller_phone %&gt;</td>
  <td>&lt;%= property.category.name %&gt;</td>
  <td>&lt;%= property.property.name %&gt;</td>
  <td>&lt;%= property.property_details %&gt;</td>
  <td>&lt;%= property.location %&gt;</td>
</tr>
```
C Compiler

Y U NO TELL ME WHERE THE SEGMENTATION FAULT IS
Singaling an error (I)

Modern approach: `raise` an error by interrupting regular control flow. This requires a special syntactic construct (e.g. `try-except`).

Singaling an error (II)

More primitive approach: use the `return value` to indicate an error.

Python `None`, Ruby `nil`, JavaScript `undefined`, C/Java `null`
In Haskell, types are “non-null”. If a function’s return type is `Integer`, it can’t return null/None.

To encode the possibility of failure in the return type, we use `Maybe`:

```
data Maybe a = Nothing | Just a
```

Examples

```
head     :: [a] -> a
safeHead :: [a] -> Maybe a
(!)      :: Ord k => Map k a -> k -> a
lookup   :: Ord k => Map k a -> k -> Maybe a
```
Chaining failing computations

Given a Map String [Integer] that maps user names to a list of scores (in reverse chronological order), and given two user names, return which user’s most recent score was higher.
def better_recent_score(scores, user1, user2):
    score1 = scores[user1][0]
    score2 = scores[user2][0]

    if score1 >= score2:
        return user1
    else:
        return user2

def better_recent_score(scores, user1, user2):
    scores1 = scores.get(user1)
    if scores1 is None:
        return None
    else:
        score1 = scores1.safeIndex(0)
        if score1 is None:
            return None
        else:
            scores2 = scores.get(user2)
            if scores2 is None:
                return None
            else:
                score2 = scores2.safeIndex(0)
                if score2 is None:
                    return None
                else:
                    if score1 >= score2:
                        return user1
                    else:
                        return user2
betterRecentScore2 :: Map String [Integer] -> String -> String
-> Maybe String
betterRecentScore2 scores user1 user2 =
  case lookup user1 scores of
      Nothing -> Nothing
      Just scores1 ->
       case safeHead scores1 of
            Nothing -> Nothing
            Just score1 ->
              case lookup user2 scores of
                    Nothing -> Nothing
                    Just scores2 ->
                      case safeHead scores2 of
                          Nothing -> Nothing
                          Just score2 ->
                            if score1 >= score2 then Just user1 else Just user2

Syntactic shortcuts in other languages

Null coalescing ("Elvis") operator ?:

x ?: y == x ? x : y

Safe navigation operator ?.

x?.y?.z
Given a `Maybe a`, we want to perform an operation on the `a` (if present).
“If null then null, else do something”

```
add10Maybe :: Maybe Integer -> Maybe Integer
add10Maybe Nothing = Nothing
add10Maybe (Just x) = Just (add10 x)

lengthMaybe :: Maybe [a] -> Maybe Integer
lengthMaybe Nothing = Nothing
lengthMaybe (Just xs) = Just (length xs)
```

“If null then null, else do something”

```
add10Maybe :: Maybe Integer -> Maybe Integer
add10Maybe Nothing = Nothing
add10Maybe (Just x) = Just (add10 x)

lengthMaybe :: Maybe [a] -> Maybe Integer
lengthMaybe Nothing = Nothing
lengthMaybe (Just xs) = Just (length xs)
```
Maybe is a **Functor**!

```
try :: (a -> b) -> Maybe a -> Maybe b
try _ Nothing = Nothing
try f (Just x) = Just (f x)
```

“If null then null, else do something that might return null”

```
recipMaybe :: Maybe Float -> Maybe Float
recipMaybe Nothing = Nothing
recipMaybe (Just x) =
    if x == 0
    then Nothing
    else Just (1 / x)

headMaybe :: Maybe [a] -> Maybe a
headMaybe Nothing = Nothing
headMaybe (Just xs) =
    if null xs
    then Nothing
    else Just (head xs)
```
tryFail :: (a -> Maybe b) -> Maybe a -> Maybe b
tryFail _ Nothing = Nothing
tryFail f (Just x) = f x

instance Monad Maybe where
  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= _ = Nothing
  (Just x) >>= f = f x

return :: a -> Maybe a
return = Just
Demo: cleaning up betterRecentScore

Reporting errors with Either
Consider the following “union” generic data type:

```
data Either a b = Left a | Right b
```

We often use `Either String b` to represent a successful value `Right x`, or an error with message `Left msg`. 