Week 10: Representing Failing Computations

CSC324 Principles of Programming Languages

David Liu, Department of Computer Science
Recap: Monads
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
\( f = \)
\[
\begin{align*}
  m1 & \gg= \backslash x1 \rightarrow \\
  m2 & \gg= \backslash x2 \rightarrow \\
  \ldots \\
  mn & \gg= \backslash xn \rightarrow \\
  mFinal
\end{align*}
\]

\( f = \textbf{do} \)
\[
\begin{align*}
  \text{x1} & \leftarrow m1 \\
  \text{x2} & \leftarrow m2 \\
  \ldots \\
  \text{xn} & \leftarrow mn \\
  mFinal
\end{align*}
\]
What do you do when a computation can fail?
Connection Lost

Internal Exception: java.lang.NullPointerException

Back to title screen
NoMethodError in Rms::Properties#index

Showing /home/sreena/ruby/indiaassets/app/views/rms/properties/index.html.erb where line #29 raised:

undefined method 'name' for nil:NilClass

Extracted source (around line #29):

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>&lt;td&gt;=% property.seller_phone %&lt;/td&gt;</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>&lt;td&gt;=% property.category.name %&lt;/td&gt;</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>&lt;td&gt;=% property.property_name %&lt;/td&gt;</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>&lt;td&gt;=% property.property_details %&lt;/td&gt;</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>&lt;td&gt;=% property.location %&lt;/td&gt;</td>
<td></td>
</tr>
</tbody>
</table>
Segmentation fault
Segmentation fault
`/bin/sh: error while loading shared libraries: F8... ET...`
cannot open shared object file: No such file or directory
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
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`:

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

head :: [a] -> a
safeHead :: [a] -> Maybe a

(!) :: Ord k => Map k a -> k -> a
lookup :: Ord k => k -> Map k a -> 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

betterRecentScore ::
    Map String [Integer] -> String -> String -> String
betterRecentScore scores user1 user2 =
    let score1 = head (scores ! user1)
        score2 = head (scores ! user2)
    in
        if score1 >= score2 then user1 else 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 ?:

```plaintext
x ?: y == x ? x : y
```

Safe navigation operator ?.

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

```haskell
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”

```haskell
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**!

```haskell
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"

```haskell
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
Maybe is a monad!

```haskell
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:

```haskell
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`. 
The power of abstraction

Writing code using only (>>=) and return enables that code to work for any Monad instance.

Examples: collectTreeState and collectTreeM; new demo now!
The List monad
It turns out that the Haskell List type is also a monad.

```haskell
instance Monad [] where
    return :: a -> [a]

    (>>=) :: [a] -> (a -> [b]) -> [b]
```
It turns out that the Haskell list type is also a monad.

```haskell
instance Monad [] where
    return :: a -> [a]
    return x = [x]

    (>>=) :: [a] -> (a -> [b]) -> [b]
    lst >>= f = concatMap f lst
```
The list type as a monad models non-deterministic choice.

A familiar demo...
Lists vs. the ambiguous operator

\[
(+ 3 (\langle 1 2 3))
\]

\[
3 + [1, 2, 3]
\]