Mixing Monads

Monads Can Be Used for Many Things!

- Partial Functions
- Global Variables
- Parsing
- Exceptions
- Test Generation
- Concurrency
- ...

```
eval :: Expr → Result Int

data Result a =
  10k a
  | Err String.

ST s a = s → (s, a)

"global var"

Parser a = String → [(String, a)]
```
Recall our expressions with division

```haskell
data Expr
    = Number Int -- ^ 0,1,2,3,4
    | Plus Expr Expr -- ^ e1 + e2
    | Div Expr Expr -- ^ e1 / e2
  deriving (Show)
```

We had a **potentially crashing** evaluator

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

-- >>> eval (Div (Val 10) (Plus (Number 5) (Number (-5))))
-- Exception: Divide by zero
```
We defined a **Result** type

```haskell
data Result a = Ok a | Err String
```

made it a Monad

```haskell
instance Monad Result where
  return x       = Ok x
  (Ok v) >>= f   = f v
  (Err s) >>= _  = Err s
```

and then we can write

```haskell
eval :: Expr -> Result Int
eval (Number n)   = return n
eval (Plus e1 e2) = do { n1 <- eval e1; n2 <- eval e2; return (n1 + n2) }
eval (Div e1 e2) = do { n1 <- eval e1;
                        n2 <- eval e2;
                        if n2 /= 0
                          then return (n1 `div` n2)
                          else Err ("DBZ: " ++ show e2)
                    }
```

which doesn’t crash but returns an *Err*

```haskell
>>> eval (Div (Number 10) (Plus (Number 5) (Number (-5))))
Err "DBZ: Plus (Number 5) (Number (-5))"
```

and when it succeeds it returns an *Ok*

```haskell
>>> eval (Div (Number 10) (Plus (Number 5) (Number (-5))))
Ok 1
```
Generalizing `Result` to `Either`

The standard library generalizes the `Result` type to `Either`

```haskell
data Result a = Err String | Ok a

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

- `Err s` becomes `Left s`
- `Ok v` becomes `Right v`
- `Result a` becomes `Either String a`

(But we can data other than `String` in the `Left` values)
EXERCISE: Generalizing Result Monad to Either Monad

Lets translate the old Monad instance for Result

```haskell
instance Monad Result where

  -- return :: a -> Result a
  return x    = Ok x

  -- (>>=) :: Result a -> (a -> Result b) -> Result b
  (Ok v) >>= f = f v
  (Err s) >>= _ = s
```

into a Monad instance for Either

```haskell
instance Monad (Either e) where

  -- return :: a -> Either e a
  return x    = ???

  -- (>>=) :: Either e a -> (a -> Either e b) -> Either e b
  (Right v) >>= f = ???
  (Left  s) >>= _ = ???
```
We can rewrite eval to return an Either

\[
\text{eval :: Expr} \rightarrow \text{Either Expr Int}
\]

\[
\text{eval (Number n)} = \text{return } n
\]

\[
\text{eval (Plus e1 e2)} = \text{do } n1 \leftarrow \text{eval e1 } \\
\quad n2 \leftarrow \text{eval e2 } \\
\quad \text{return } (n1 + n2)
\]

\[
\text{eval (Div e1 e2)} = \text{do } n1 \leftarrow \text{eval e1 } \\
\quad n2 \leftarrow \text{eval e2 } \\
\quad \text{if } n2 \neq 0 \text{ then return } (n1 \div n2) \text{ else Left e2}
\]

What does quiz evaluate to?

\[
\text{quiz = eval (Div (Val 10) (Plus (Number 5) (Number (-5)))}
\]

A. Err "DBZ: Plus (Number 5) (Number (-5))"

B. Left "DBZ: Plus (Number 5) (Number (-5))" :: Either String ...

C. Run-time Exception

D. Plus (Number 5) (Number (-5))

E. Left (Plus (Number 5) (Number (-5))) :: Either Expr ...
Either is an Exception Monad!

What can you do with exceptions?

1. `throwError` an exception (with some value) ...
2. `catchError` an exception (and use its value) ...

```
expr : Either e a

an expr that either
- successfully produces a value
- or throws an "uncaught" error e
```
1. **throwing an Exception**

We can simply define

```hs
throw :: e -> Either e a
throw exn = Left exn
```

and now **voila**

```hs
eval :: Expr -> Either Expr Int
eval (Number n)  = return n
eval (Plus  e1 e2) = do n1 <- eval e1
                      n2 <- eval e2
                      return (n1 + n2)
eval (Div   e1 e2) = do n1 <- eval e1
                      n2 <- eval e2
                      if n2 /= 0
                         then return (n1 `div` n2)
                         else throw e2

Exactly the same evaluator

- Result is a Left ==> an *exception* came all the way to the top.
- Either monad ensures the “exception” shoots to the top!
>>> eval (Div (Numer 10) (Plus (Number 5) (Number (-5)))))
Left (Minus (Number 5) (Number 5))

No further evaluation happens after a `throw` because??

**catching an exception**

How to `catch` an exception?

Let's change our `Expr` type to

```haskell
data Expr
    = Number Int       -- ^ 0,1,2,3,4
    | Plus Expr Expr    -- ^ e1 + e2
    | Try Expr Int
    deriving (Show)
```

Informally, `try e n` evaluates to `e` but
- if e is undefined due to *divide-by-zero*
- then evaluate to n

```haskell
eval :: Expr -> Either Expr Int
eval (Number n)  = return n
eval (Plus  e1 e2) = do n1 <- eval e1
                     n2 <- eval e2
                     return (n1+n2)
eval (Div   e1 e2) = do n1 <- eval e1
                     n2 <- eval e2
                     if n2 /= 0
                     then return (n1 `div` n2)
                     else throw e2

eval (Try e n)   = catch (eval e) (\_ -> return n)
```

**QUIZ**

What should the type of `catch` be?

A. Either e a -> (a -> Either e b) -> Either e b

B. Either e a -> (e -> Either e b) -> Either e b

C. Either e a -> (e -> Either e a) -> Either e a

D. Either e a -> Either e a -> Either e a

E. Either e a -> Either e b -> Either e b
Implementing *catch*

Let's implement the catch function!

```haskell
catch :: Either e a -> (e -> Either e a) -> Either e a
catch (Left e) handler = ???
catch (Right a) handler = ???
```

**QUIZ**
catch :: Either e a -> (e -> Either e a) -> Either e a
catch (Left e) handle = ???
catch (Right a) handler = ???

eval :: Expr -> Either Expr Int
eval (Number n) = return n
eval (Plus e1 e2) = do n1 <- eval e1
                          n2 <- eval e2
                          return (n1+n2)
eval (Div e1 e2) = do n1 <- eval e1
                         n2 <- eval e2
                         if n2 /= 0
                            then return (n1 `div` n2)
                            else throw e2
eval (Try e n) = catch (eval e) (\_ -> return n)

e1 = Div (Number 10) (Plus (Number 5) (Number (-5)))
e1' = Try e1 7

quiz = eval (Try e1 7)

What does quiz evaluate to?

A. Right 7
B. Left 7
C. Right 0
D. Left 0
E. Left (Plus (Number 5) (Number (-5)))
Either is an Exception Monad!

1. throw an exception (with some value) ...

2. catch an exception (and use its value) ...

\[
\begin{align*}
\text{throw} &: \ e \rightarrow \text{Either} \ e \ a \\
\text{throw } e &= \text{Left } e \\

\text{catch} &: \text{Either} \ e \ a \rightarrow (e \rightarrow \text{Either} \ e \ a) \rightarrow \text{Either} \ e \ a \\
\text{catch} \ (\text{Left } e) \ \text{handle} &= \ \text{handle } e \\
\text{catch} \ (\text{Right } e) \ _ &= \ \text{Right } e
\end{align*}
\]
Monads Can Be Used for Many Things!

- Partial Functions
- Global State
- Parsing
- Exceptions
- Test Generation
- Concurrency
- ...

... but what if I want Exceptions and Global State?

Data Expr = ...

eval :: Expr → ? Int

- "throw" an error if DBT
- "catch" if using Def
- "count" operations

Mixing Monads

What if I want Exceptions and Global State?