>>> 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?

Lets change our Expr type to

```
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
Implementing \textit{catch}

Let's implement the \textit{catch} function!

\[
\text{catch} :: \text{Either } e \ a \rightarrow (e \rightarrow \text{Either } e \ a) \rightarrow \text{Either } e \ a
\]

\[
\text{catch (Left } e \text{) handler = ???}
\]

\[
\text{catch (Right } a \text{) handler = ???}
\]

\textbf{QUIZ}
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?

Mixing Monads

What if I want Exceptions and Global State?
Profiling with the ST Monad

Lets implement a profiling monad that counts the number of operations

-- A State-Transformer with a "global" `Int` counter

```haskell
type Profile a = State Int a
```

We can write a `runProfile` that

- executes the transformer from 0
- and renders the result
runProfile :: (Show a) => Profile a -> String
runProfile st = showValCount (runState st 0)

showValCount :: (Show v, Show c) => (v, c) -> String
showValCount (val, count) = "value: " ++ show val ++ ", count: "
++ show count

A function to increment the counter

count :: Profile ()
count = do
  n <- get
  put (n+1)
A Profiling Evaluator

We can use `count` to write a profiling evaluator

```haskell
evalProf :: Expr -> Profile Int
evalProf = eval
  where
    eval (Number n)    = return n
    eval (Plus  e1 e2) = do n1 <- eval e1
                            n2 <- eval e2
                            count
                            return (n1+n2)
    eval (Div   e1 e2) = do n1 <- eval e1
                            n2 <- eval e2
                            count
                            return (n1 `div` n2)
```

And now, as there are two operations, we get
>>> e1
Div (Number 10) (Plus (Number 5) (Number 5))

>>> runProfile (evalProf e1)
"value: 1, count: 2"

But what about Divide-by-Zero?

Bad things happen...
>>> e2
Div(Number 10)(Plus(Number 5)(Number (-5)))

>>> runProfile(evalProf e2)
*** Exception: divide by zero
"value:

Problem: How to get global state AND exception handling?
m implements

- no special operations

**Transform it to add some Capabilities**

Transform1 m implements

- m operations and
- operations added by Transform1

**Transform again to add more Capabilities**

Transform2 (Transform1 m) implements

- m operations and
- operations added by Transform1 and
- operations added by Transform2

... And so on
Transform3 (Transform2 (Transform1 m)) implements

- m operations and
- operations added by Transform1 and
- operations added by Transform2 and
- operations added by Transform3 ...


Mixing Monads with Transformers

- Step 1: Specifying Monads with Extra Features
- Step 2: Implementing Monads with Extra Features
Specifying Monads with Extra Features

First, instead of using concrete monads e.g. Profile or Either

- e.g. Profile or Either

We will use type-classes to abstractly specify a monad’s capabilities

- e.g. MonadState s m or MonadError e m
A Class for State-Transformers Monads

The class `MonadState s m` defined in the `Control.Monad.State` (http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Except.html) says

- `m` is a `State-Transformer` monad with state type `s`

```
class Monad m => MonadState s m where
  get :: m s
  put :: s -> m ()
```

That is to say, `m` implements

- `>>=` and `return` operations specified by `Monad` and
- `get` and `put` operations specified by `MonadState`!

Generalize Types to use Classes

So we can generalize the type of `count` to use `MonadState Int m`

```
count :: (MonadState Int m) => m ()
count = do
  n <- get
  put (n+1)
```
A Class for Exception Handling Monads

The class `MonadError e m` defined in `[Control.Monad.Except][6]` says

- `m` is a *Exception-Handling* monad with exception type `e`

```haskell
class Monad m => MonadError e m where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a
```

That is to say, `m` implements

- `>>=` and `return` operations specified by `Monad` *and*
- `throwError` and `catchError` operations specified by `MonadError`!

Generalize Types to use Classes

So we can *generalize* the type of `tryCatch` to use `MonadError e m`

```haskell
tryCatch :: (MonadError e m) => m a -> a -> m a
tryCatch m def = catchError m (\_ -> return def)
```
**Generalize eval to use Constraints**

We can now specify that eval uses a monad \( m \) that implements

- MonadState Int and MonadError Expr
eval :: (MonadState Int m, MonadError Expr m) => Expr -> m Int
eval (Number n)   = return n
eval (Plus  e1 e2) = do n1 <- eval e1
                       n2 <- eval e2
                       count
                       return (n1 + n2)

eval (Div  e1 e2) = do n1 <- eval e1
                       n2 <- eval e2
                       count
                       if (n2 /= 0)
                           then return (n1 `div` n2)
                           else throwError e2

eval (Try e n)    = tryCatch (eval e) n

Lets try to run it!

>>> e1

>>> evalMix e1
... GHC yells "please IMPLEMENT this MAGIC monad that implements BOTH features"
Mixing Monads with Transformers

- Step 1: Specifying Monads with Extra Features
- Step 2: Implementing Monads with Extra Features

Implementing Monads with Extra Features
Transform2 (Transform1 m) implements

- m operations and
- operations added by Transform1 and
- operations added by Transform2

We require

- A basic monad m
- A Transform1 that adds State capabilities
- A Transform2 that adds Exception capabilities

A Basic Monad

First, let's make a basic monad
- only implements \( \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_<b>>=</b> \) and \( \_\_\_\_\_\_\_\_\_\_\_\_\_\_<b>return</b> \)

```haskell
data Identity a = Id a

instance Monad Identity where
    return a = Id a
    (Id a) >>= f = f a
```

A very basic monad: just a \textbf{wrapper} \((\text{Id})\) around the value \((a)\)

- No extra features

---

\textbf{A Transform that adds} \textit{State} Capabilities

The transformer \texttt{StateT s m} defined in the \texttt{Control.Monad.State} module ([http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Except.html](http://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Except.html)) - takes as input monad \texttt{m} and

- transforms it into a new monad \texttt{m'}
such that $m'$ implements

- all the operations that $m$ implements

- *and adds* State-transformer capabilities

StateT $s$ $m$ satisfies the constraint $(\text{MonadState} \ s \ (\text{StateT} \ s \ m))$

**A State-transformer over $\textbf{Int}$ states**

```
type Prof = StateT Int Identity
```

We can go back and give `evalProf` the type

```
 evalProf :: Expr -> Prof Int
```
A Transform that adds *Exception* Capabilities

The transformer `ExceptT e m`

- takes as input a monad `m` and
- transforms it into a new monad `m'`

such that `m'` implements

- all the operations that `m` implements
- and adds Exception-handling capabilities

`ExceptT e m` satisfies the constraint `MonadError e (ExceptT e m)`

An Exception Handler Monad with *Expr*-typed exceptions

```haskell
type Exn = ExceptT Expr Identity
```

We can go back and give `evalThrowCatch` the type

`evalThrowCatch :: Expr -> Exn Int`
Composing Transformers

We can use both transformers to get both powers!

```
type ExnProf a = ExceptT Expr (StateT Int (Identity)) a
```

ExnProf implements State-transformer-over Int and Exception-handling-over-Expr
**EXERCISE: Executing the Combined Transformer**

Recall that

```
type ExnProf a = ExceptT Expr (StateT Int (Identity)) a
```

Let's write a function

```
runExnProf :: (Show a) => ExnProf a -> String
runExnProf epm = ???
```

such that

```
>>> runExnProf (eval e1)
"value: 1, count: 2"

>>> runExnProf (eval e2)
"Plus (Number 5) (Number (-5)) after 2 operations"
```
TRY AT HOME: Combining in a Different Order

We can also combine the transformers in a different order.

```haskell
type ProfExn a = StateT Int (ExceptT Expr (Identity)) a
```

`ExnProf` implements `State-transformer-over Int` and `Exception-handling-over-Expr`.

Can you implement the function

```haskell
runProfExn :: (Show a) => ProfExn a -> String
```

such that when you are done, we can get the following behavior?
>>> runProfExn (eval e1)
"value: 1, count: 2"

>>> runProfExn (eval e2)
"Left (Plus (Number 5) (Number (-5)))"

Summary: Mixing Monads with Many Features

1. Transformers add capabilities to Monads

Transform2 (Transform1 m) implements

- m operations and
- operations added by Transform1 and
- operations added by Transform2
2. **StateT** and **ExceptT** add State and Exceptions

- Start with a *basic monad* **Identity**
- Use **StateT** **Int** to add global **Int** *state-update* capabilities
- Use **ExceptT** **Expr** to add *exception-handling* capabilities

Play around with this in your homework assignment!

(https://plus.google.com/u/0/104385825850161331469)
(https://github.com/ranjitjhala)
