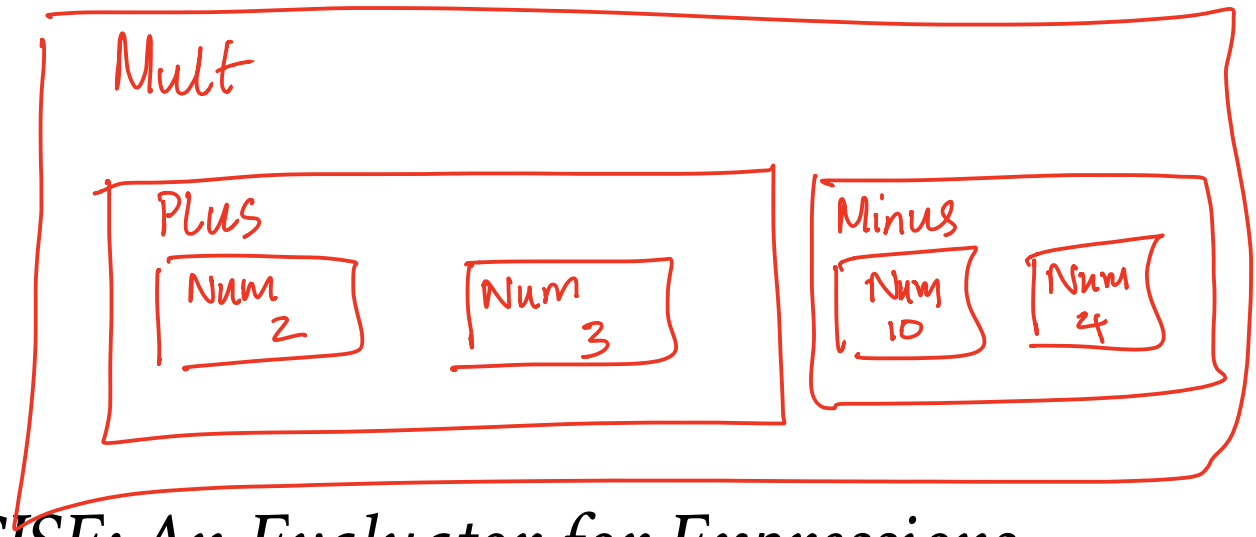
```
-- >>> fmap (\n -> n + 1) (Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf
Leaf))
-- (Node 4 (Node 1 Leaf Leaf) (Node 9 Leaf Leaf))

-- >>> fmap show [1,2,3]
-- ["1", "2", "3"]
```

A Type to Represent Expressions

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

Some Example Expressions

```
e1 = Plus  (Number 2) (Number 3)      -- 2 + 3
e2 = Minus (Number 10) (Number 4)     -- 10 - 4
e3 = Mult e1 e2                       -- (2 + 3) * (10 - 4)
e4 = Div  e3 (Number 3)               -- ((2 + 3) * (10 - 4)) / 3
```



EXERCISE: An Evaluator for Expressions

Fill in an implementation of `eval`

```
eval :: Expr -> Int
eval e = ???
```

so that when you're done we get

```
-- >>> eval e1
-- 5
-- >>> eval e2
-- 6
-- >>> eval e3
-- 30
-- >>> eval e4
-- 10
```

QUIZ

What does the following evaluate to? 60 / (5 - 5)

```
quiz = eval (Div (Number 60) (Minus (Number 5) (Number 5)))
```

A. `0`

B. `1`

C. Type error

D. Runtime exception

E. `NaN`

To avoid crash, return a `Result`

Lets make a data type that represents `Ok` or `Error`

```
data Result v
  = Ok    v       -- ^ a "successful" result with value `v`
  | Error String -- ^ something went "wrong" with `message`
  deriving (Eq, Show)
```

EXERCISE

Can you implement a `Functor` instance for `Result`?

```
instance Functor Result where
  fmap f (Error msg) = ???
  fmap f (Ok val)    = ???
```

When you're done you should see

```
-- >>> fmap (\n -> n ^ 2) (Ok 9)
-- Ok 81

-- >>> fmap (\n -> n ^ 2) (Error "oh no")
-- Error "oh no"
```

Evaluating without Crashing

Instead of *crashing* we can make our `eval` return a `Result Int`

```
eval :: Expr -> Result Int
```

- If a sub-expression has a *divide by zero* return `Error "..."`
- If all sub-expressions are *safe* then return `Ok n`

EXERCISE: Implement `eval` with `Result`

```
eval :: Expr -> Result Int
eval (Number n)    = ?
eval (Plus  e1 e2) = ?
eval (Minus e1 e2) = ?
eval (Mult  e1 e2) = ?
eval (Div   e1 e2) = ?
```

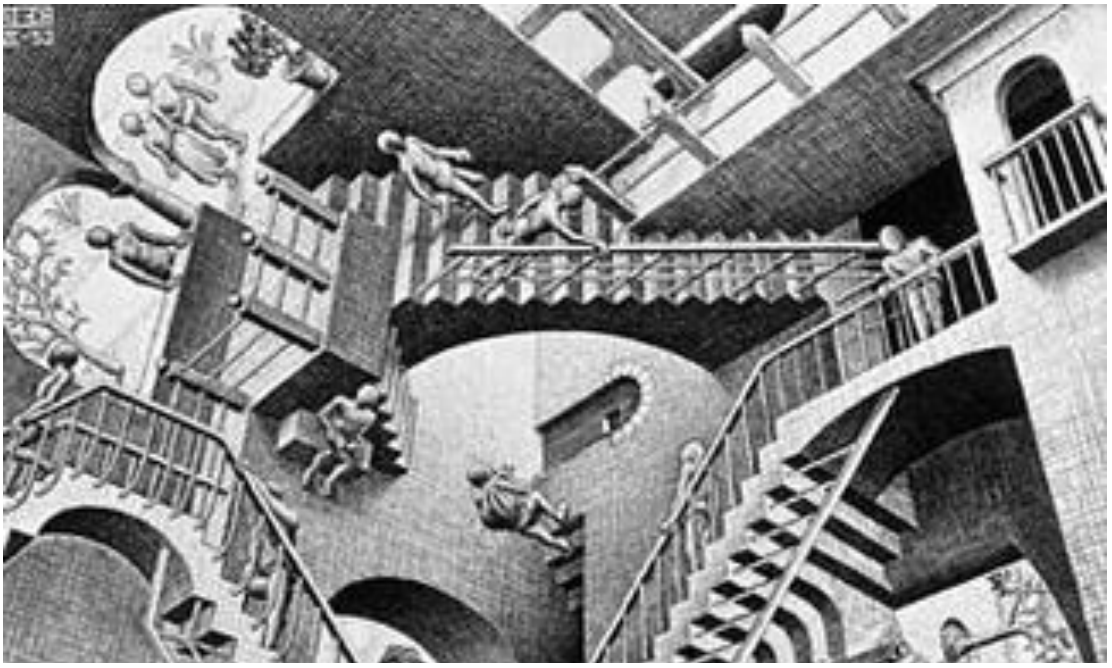
The Good News

No nasty exceptions!

```
>>> eval (Div (Number 6) (Number 2))
Ok 3

>>> eval (Div (Number 6) (Number 0))
Error "yikes dbz:Number 0"

>>> eval (Div (Number 6) (Plus (Number 2) (Number (-2))))
Error "yikes dbz:Plus (Number 2) (Number (-2))"
```

The BAD News!

The code is **super gross**



Escher's Staircase

Lets spot a Pattern

The code is gross because we have these cascading blocks

```
case e1 of
  Error err1 -> Error err1
  Ok    v1   -> case e2 of
                  Error err2 -> Error err2
                  Ok    v1   -> Ok    (v1 + v2)
```

but *look closer* ... both blocks have a **common pattern**

```
case e of
  Error err -> Error err
  Value v   -> {- do stuff with v -}
```

```
bottle e doStuff =
  case e of
    Error err -> Error err
    Ok   v    -> doStuff v
```

bottle e doStuff

1. Evaluate `e`
2. If the result is an `Error` then *return* that error.
3. If the result is a `Value v` then *further process* with `v`.

doStuff v

Lets Bottle that Pattern in Two Functions

Bottling a Magic Pattern

- `>>=` (pronounced *bind*)
- `return` (pronounced *return*)

```
(>>=) :: Result a -> (a -> Result b) -> Result b
(Error err) >>= _       = Error err
(Value v)   >>= process = process v

return :: a -> Result a
return v = Ok v
```

NOTE: `return` **is not a keyword**

- it is the name of a function!

return v = Ok v

A Cleaned up Evaluator

The magic bottle lets us clean up our `eval`

```
eval :: Expr -> Result Int
eval (Number n)   = Ok n
eval (Plus e1 e2) = eval e1 >>= \v1 ->
                    eval e2 >>= \v2 ->
                      Ok (v1 + v2)
                      return

eval (Div e1 e2)  = eval e1 >>= \v1 ->
                    eval e2 >>= \v2 ->
                      if v2 == 0
                        then Error ("yikes dbz:" ++ show e2)
                        else Ok (v1 `div` v2)
```

The gross *pattern matching* is all hidden inside `>>=`

Notice the `>>=` takes *two* inputs of type:

- `Result Int` (e.g. `eval e1` or `eval e2`)
- `Int -> Result Int` (e.g. the *processor* takes the `v` and does stuff with it)

In the above, the processing functions are written using `\v1 -> ...` and `\v2 -> ...`

NOTE: It is *crucial* that you understand what the code above is doing, and why it is actually just a "shorter" version of the (gross) nested-case-of `eval`.

A Class for `>>=`

The `>>=` operator is useful across **many** types!

- like `fmap` or `show` or `toJSON` or `==`, or `<=`

Lets capture it in a typeclass:

```
class Monad m where
  -- (>>=)  :: Result a -> (a -> Result b) -> Result b
     (>>=)  :: m a      -> (a -> m b)      -> m b

  -- return :: a -> Result a
     return :: a -> m a
```

`Result` *is an instance of* `Monad`

Notice how the definitions for `Result` fit the above, with `m = Result`

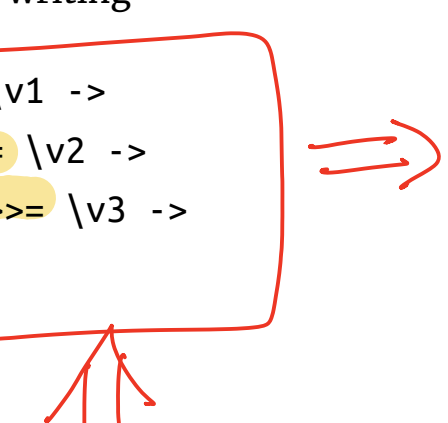
```
instance Monad Result where
  (>>=) :: Result a -> (a -> Result b) -> Result b
  (Error err) >>= _       = Error err
  (Value v)   >>= process = process v

  return :: a -> Result a
  return v = Ok v
```

Syntax for `>>=`

In fact `>>=` is *so* useful there is special syntax for it.

Instead of writing

```
e1 >>= \v1 ->
  e2 >>= \v2 ->
    e3 >>= \v3 ->
      e
```



you can write

```
do v1 <- e1
   v2 <- e2
   v3 <- e3
   e
```

do { $v_1 \leftarrow e_1$;
$v_2 \leftarrow e_2$;
$v_3 \leftarrow e_3$;
e
}

or if you like curly-braces

```
do { v1 <- e1; v2 <- e2; v3 <- e3; e }
```

Simplified Evaluator

Thus, we can further simplify our `eval` to:

$e_1 + e_2$

int

return :: a → t a

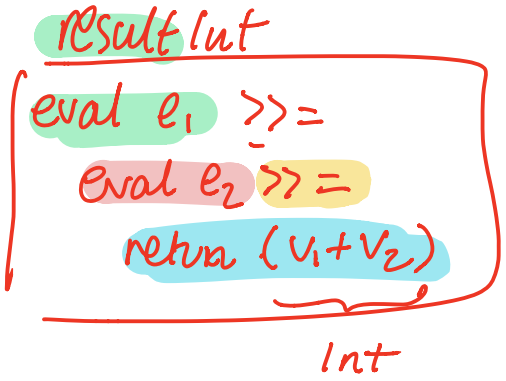
Int → Result Int

(>>=) :: t a → (a → t b) → t b

Result Int

Int → Result b

```
eval :: Expr -> Result Int
eval (Number n)   = return n
eval (Plus e1 e2) = do v1 <- eval e1
                       v2 <- eval e2
                       return (v1 + v2)
eval (Div e1 e2)  = do v1 <- eval e1
                       v2 <- eval e2
                       if v2 == 0
                         then Error ("yikes dbz:" ++ show e2)
                         else return (v1 `div` v2)
```



Which now produces the result

```
>>> evalR exQuiz
Error "yikes dbz:Minus (Number 5) (Number 5)"
```