# 1 Monads

## 1.1 Functors

### 1.1.1 Class for Mapping

Recall

```
eval :: Expr -> Result Int
eval (Num n)            = Value n
eval (Plus e1 e2)       =
    case eval e1 of
        Error err1  -> Error err1
        Value v1    -> case eval e2 of
                         Error err2  -> Error err2
                         Value v1    -> Value (v1 + v2)

eval (Div e1 e2)        =
    case eval e1 of
        Error err1  -> Error err1
        Value v1    -> case eval e2 of
                         Error err2  -> Error err2
                         Value v1    -> if v2 == 0
                                         then Error ("DBZ: " ++ show e2)
                                         else Value (v1 ‘div‘ v2)
```

Notice how there is a common pattern. Our goal is to refactor the common pattern so we can avoid repeating it. In this case, the pattern of interest is

```
case eval e2 of
    Error err2  -> Error err2
    Value v1    -> ...
```

We always have some `case ... of`, where the `...` is of type `Result`. If we get an `Error`, then we just return that `Error` If we get a `Value`, we will return that `Value`. So, how do we abstract this into a pattern?

```
bind :: Result a -> (a -> Result b) -> Result b
bind res process = case res of
    Error msg   -> Error msg
    Value v     -> process v
```

This can be refactored to:

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

Finally, we can make this into an infix operator:

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

Rewriting this to look more natural, we have

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

So, `>>=` takes two inputs:

- `Result a`: The result of the first evaluation.
- `a -> Result b`: In case the first evaluation produced a value, what to do *next* with the value.

(Quiz.) With `>>=` defined as before, what does the following evaluate to?

```
eval (Num 5) >>= \v -> Value (v + 1)
```

(a) Type Error.

(b) `5`

(c) `Value 5`

(d) `Value 6`

(e) `Error msg`

The answer is **D**. Recall that our `Result` is defined by

```
data Result a
    = Error String
    | Value a
```

So, `eval (Num 5)` will give us back a `Value 5`. Since this is not an error, we can extract `5` from `Value 5`, and then pass `5` into the function to get `Value 5 + 1 = Value 6`.

(Quiz.) With `>>=` defined as before, what does the following evaluate to?

```
eval (Error "nope") >> \v -> Value (v + 1)
```

(a) Type Error.

(b) `5`

(c) `Value 5`

(d) `Value 6`

(e) `Error "nope"`

The answer is **E**. Because we have an error `Error "nope"`, we can immediately use the `Error` pattern.

So, with this new function, we can do

```
eval :: Expr -> Result Int
eval (Num n)      = Value n
eval (Plus e1 e2) = eval e1 >>= \v1 ->
                    eval e2 >>= \v2 ->
                    Value (v1 + v2)
eval (Div e1 e2)  = eval e1 >>= \v1 ->
                    eval e2 >>= \v2 ->
                    if v2 == 0
                      then Error ("DBZ: " ++ show e2)
                      else Value (v1 `div` v2)
```

## 1.2 Monads

Note that `>>=`, like `fmap` or `show` or `==`, can be useful across many types, not just `Result`. Let us define a type class for it.

```
class Monad m where
    (>>=)   :: m a -> (a -> m b) -> m b          -- Bind
    return  :: a -> m a                          -- Return
```

### 1.2.1 Monad Instance for Result

Now, let's make `Result` an instance of `Monad`.

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

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

With this in mind, we can simplify our `eval` further.

```
eval :: Expr -> Result Int
eval (Num 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 ("DBZ: " ++ show e2)
                            else return (v1 `div` v2)
```

Note that `>>=` is so useful that there is a special syntax for it; known as a `do` block, instead of writing

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

we can just write

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

Therefore, we can simplify our `eval` to

```
eval :: Expr -> Result Int
eval (Num 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 ("DBZ: " ++ show e2)
                            else return (v1 `div` v2)
```

### 1.2.2 Either Monad

Knowing that error handling is a common task, instead of defining our own `Result` type, we can use `Either` from the Haskell standard library. So,

```
data Either a b
    = Left  a       -- Something has gone wrong.
    | Right b       -- Everything has gone right.
```

Since `Either` is already an instance of `Monad`, we do not need to define our own `>>=`.