From Failures to Lists of Successes

Recap: Monad

`Monad` is a typeclass with two functions

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

A `Maybe` Monad

We can define a `Maybe` a type to represent "maybe-null" values

```
data Maybe val
  = Just val            -- ^ "Just one value" :-)
  | Nothing             -- ^ "No value" :-(
```

A the `Monad` instance for `Maybe`

Can you help me fill this in?

```
instance Monad Maybe where
  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing   >>= _ = ???
  (Just v)  >>= f = ???

  return :: a -> Maybe a
  return v = ???
```

`Maybe` represents computations that may produce no value

A value of type `Maybe` a is either

- `Nothing` which we can think of as representing *failure*, or
- `Just` x for some x of type a, which we can think of as *success*

Using `Maybe` for computations that may produce no value

We saw how to write an `eval` function that doesn't crash

- But instead gracefully returns a `Nothing` (if there is a div-by-zero)

```
eval :: Expr -> Maybe Int
eval (Number n)  = Just 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 Nothing
                         else Just (n1 `div` n2)
```

Replacing Failure by a List of Successes

Lets *generalize* the `Maybe` monad into a *list* monad!

- `Nothing` is the empty list `[]`
- `Just` v is the singleton list `[v]`

.. but maybe there's something sensible for lists with *many* elements?

QUIZ

Lets make lists an instance of `Monad` by:

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

instance Monad [] where
  return = returnForList
  (>>=) = bindForList
```

What must the type of `returnForList` be ?

A. `[a]`

B. `a -> a`

C. `a -> [a]`

D. `[a] -> a`

E. `[a] -> [a]`

A Monad Instance for Lists

Lets implement the `Monad` instance for lists!

```
-- returnForList :: a -> [a]
returnForList x = ???
```

What's the only sensible implementation?

QUIZ

Lets make lists an instance of `Monad` by:

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

instance Monad [] where
  return = returnForList
  (>>=) = bindForList
```

What must the type of `bindForList` be?

A. `[a] -> [b] -> [b]`

B. `[a] -> (a -> b) -> [b]`

C. `[a] -> (a -> [b]) -> b`

D. `[a] -> (a -> [b]) -> [b]`

E. `[a] -> [b]`

QUIZ

Which of the following is a valid

```
bindForList :: [a] -> (a -> [b]) -> [b]
bindForList = bfl
```

```
-- a
bfl f []     = []
bfl f (x:xs) = f x : bfl f xs
```

```
-- b
bfl f []     = []
bfl f (x:xs) = f x ++ bfl f xs
```

```
-- c
bfl []     f = []
bfl (x:xs) f = f x ++ bfl f xs
```

```
-- d
bfl []     f = []
bfl (x:xs) f = f x : bfl f xs
```

```
-- e
bfl []     f = []
bfl (x:xs) f = x : f xs
```

The List Monad

Lets "run" the `>>=` on some inputs to see how it behaves!

```
(>>=) :: [a] -> (a -> [b]) -> [b]
[]     >>= _ = []
(x:xs) >>= f = f x ++ (xs >>= f)
```

```
[] >>= f  ===> []
```

```
[x3] >>= f  ===> f x3 ++ ([] >>= f)  ===> f x3
```

```
[x2,x3] >>= f  ===> f x2 ++ ([x3] >>= f)   ===> f x2 ++ f x3
```

```
[x1,x2,x3] >>= f  ===> f x1 ++ ([x2,x3] >>=f ) ===> f x1 ++ f x2 ++ f x3
```


QUIZ

What does the following program evaluate to?

```
quiz = do x <- ["cat", "dog"]
          y <- [0, 1]
          return (x, y)
```

A. `[("cat", 0)]`

B. `[("dog", 1)]`

C. `[("cat", "dog", 0, 1]`

D. `[("cat", 0, "dog", 1]`

E. `[("cat", 0), ("cat", 1), ("dog", 0), ("dog", 1)]`

Whoa, behaves like a `for`-loop!

Lets work it out.

```
do {x <- ["cat", "dog"]; y <- [0, 1]; return (x, y)}

== ["cat", "dog"] >>= (\x -> [0, 1] >>= (\y -> return (x, y)))
```

Now lets break up the evaluation

```
["cat","dog"] >>= (\x -> [0, 1] >>= (\y -> return (x, y)))
x:  "cat"
==> [0, 1] >>= (\y -> return ("cat", y))
y:  0
==> [("cat", 0)]
y:  1
==> [("cat", 0), ("cat", 1)]
```

So

```
["cat", "dog"] >>= (\x -> [(x, 0), (x, 1)])

==> [("cat", 0), ("cat", 1)] ++ ["dog", 0), ("dog", 1)]

==> [("cat", 0), ("cat", 1), ("dog", 0), ("dog", 1)]
```

QUIZ

What does `quiz` evaluate to?

```
foo f xs = do
  x <- xs
  return (f x)

quiz = foo (\n -> n*n)  [0,1,2,3]
```

A. `[0]` **B.** `[0,1,4,9]` **C.** `[9]` **D.** *Type Error* **E.** *Runtime Exception*

QUIZ

What does the following evaluate to?

```
triples :: [(Int, Int, Int)]
triples = do
    x <- [0,1]
    y <- [10,11]
    z <- [100,101]
    []
```

A. `[(0,10,100), (0,10,101),(1,10,100),(1,10,101),(0,11,100), (0,11,101)]`

B. `[]`

C. `[[]]`

D. `[(0,10,100), (1,11,101)]`

E. `[0,1,10,100,100,101]`

EXERCISE: Using the List Monad

A **Pythagorean Triple** is a

- triple of positive integers `a`, `b`, `c`
- such that `a*a + b*b = c*c`

Lets implement a function to return all triples where

- `a`,`b`,`c` are between `0..n`

```
pyTriples :: Int -> [(Int, Int, Int)]
pyTriples n = do
  a <- ???
  b <- ???
  c <- ???
  ???
```

HINT: You can write `[i..j]` to generate the list of numbers between `i` and `j`

```
>>> [0..5]
[0,1,2,3,4,5]
```

Using the List Monad

So lets implement a function

```
bits :: Int -> [String]
```

Such that

```
>>> bits 0
[]
>>> bits 1
["0", "1"]
>>> bits 2
["00", "01", "10", "11"]

>>> bits 3
["000", "001", "010", "011", "100", "101", "110", "111"]
```

Summary

The `Maybe` or `Result` monad instance

- Conveniently work with computations that *may fail*

Generalize to `List` monad instance

- *empty list* is *failure*
- *non-empty* list is *successes*

Gives us a `for`-loop or iterator *for free*.