List as a monad

- The `Maybe` type can be thought of as a monad. Anything else?
- Lists!
- ```
  instance Monad [] where
    return a   = [a]
    lst >>= f = concat (map f lst)
    fail _     = []
  ```

What could it mean?

What does it mean to say that lists form a monad?
It represents the type of computations that may return 0, 1, or more results.
More specifically, it combines actions by applying the operations to *all possible values*.

How does it work?

Take this code (see `LIST.hs`):

```
cart xs ys = do x <- xs
                y <- ys
                return (x, y)
```

What will an application like `cart [1,2,3] [97,98,99]` do, with a `Monad` instance for lists that looks like this?

```
instance Monad [] where
  return a   = [a]
  lst >>= f = concat (map f lst)
  fail _     = []
```

```
Prelude> cart [1,2,3] [97,98,99]
[(1,97),(1,98),(1,99),(2,97),(2,98),(2,99),
(3,97),(3,98),(3,99)]
```

List Comprehensions are monads

- In the list monad, `x <- xs`, where `xs` is a list means that `x` will successively take all values in `xs`.

  `mymap f xs = do { x <- xs ; return (f x) }`

  is the same as the list comprehension:

  `mymap' f xs = [ f x | x <- xs ]`
- The `cart` function is equivalent to

  `cart' xs ys = [ (x,y) | x <- xs, y <- ys ]`
```

Square Root (example)

Consider the following square-root function that generates an error if its argument is negative (see `ROOT.hs`):

```
root x
 = if x < 0
    then fail ("negative! "++show x)
    else return $ sqrt x
```

It is monadic code, because it uses `fail` and `return`
Which monad? `IO`, `Maybe`, `[]`, ?
Let's ask:

```
> :i root
root :: (Floating r, Monad m, Ord r, Show r) => r -> m r
```

It should work in any monad!

Running `root`

```
> root 4 :: Maybe Double
Just 2.0
```

```
> root 4 :: [Double]
[2.0]
```

It works!

Running `root`

```
> root (-1) :: Maybe Double
Nothing
```

```
> root (-1) :: [Double]
[]
```

It does errors too

Back to `cart`

What would you expect this code to return?

```
>cart (Just 4) (Just 5)
(Just (4, 5))
```

```
> cart (Just 4) Nothing
Nothing
```

```
> cart [] [1,2,3]
[]
```

Sum of two roots (example)

Now, we define `sum2roots` in terms of `root`

```
sum2roots x y
 = do rx <- root x
      ry <- root y
      return (rx + ry)
```

Monadic as well, as we use the do-notation.

```
> sum2roots 4 9 :: Maybe Double
Just 5.0
```

```
> sum2roots (4) (-9) :: [Double]
[]
```

All works as expected here as well.

Error messages only in `IO`

- In the above examples, only `IO` actually showed the error message
- Can we see the errors in non-IO, i.e. regular function code?
- We can, if we use the `Either` type constructor:

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

- Instead of `Maybe a`, we could use `Either String a`, so an error would return as `Left "..error message.."`
- `Either` has a `Monad` instance:

  ```
  instance Monad (Either a) where
    return x                  =  Right x
    (Left errstr) >>= _    =  Left errstr
    (Right x)     >>= f    =  f x
  ```

Running `sum2roots` in `Either`

```
> sum2roots (4) (9) :: Either String Double
Right 5.0
> sum2roots (4) (-9) :: Either String Double
*** Exception: negative! -9.0
```

!! The correct answer is fine (uses `Right`), but the error is a Haskell exception, and not a `Left` value.
Look at this:

```
 > sum2roots (4) (9) :: Either a Double
Right 5.0
> sum2roots (4) (-9) :: Either a Double
*** Exception: negative! -9.0
```

We have an arbitrary type `a`, but we still get the same effect!
What's going on?

Default `fail` needed in generic `Either` monad

- The Prelude `Monad` instance is for `Either e`, where `e` is an arbitrary type
- So what can `fail errmsg` do? It cannot return `Left errmsg`, becuse `e` may not itself be `String`
- The `fail` method is not mandatory, and it defaults to

  ```
  fail errmsg =  error errmsg
  ```

  So we get a runtime error, instead of a left value

Wanted: Monad instance for Either String

- We could try to define an instance in our code, with

  ```
  fail errmsg = Left errsmg
  ```

- We hit two problems:
  - It clashes with (overlaps with) the Prelude instance.
    We might used import trickery to hide the Prelude instance,
  - Even so,we still get an error:
    class instances for type-constructors that are partially applied (e.g. `Either e`) have a restriction that the arguments can only be type variables, and not concrete types.
- The solution is to use `newtype` to clone `Either String`, and then give it an instance

Cloning and Instancing Either String

- ```
  newtype ES a = ES (Either String a)
  ```
- ```
  instance Monad ES where
    return x                       =  ES $ Right x
    (ES (Left errstr)) >>= _       =  ES $ Left errstr
    (ES (Right x))     >>= f       =  f x
    fail errstr                    =  ES $ Left errstr
  ```

  We have do a lot of wrap/unwrap of data-constructor `ES`, but that's the edit/compile time cost of `newtype` (no runtime impact at all).
- However, all that wrap/unwrap overhead is buried in the monad instance above, so it doesn't appear in the code for either `root` or `sum2roots`.

ES Monad instance, with annotations

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


instance Monad ES where

  return (x :: a)                = ES (Right (x :: a)) :: ES a

  (ES (Left errstr) :: ES a) >>= _
                                   = ES (Left errstr) :: ES b

  (ES (Right (x :: a))) :: ES a >>= (f :: a -> ES b)
                                               = f x :: ES b

  fail (errstr :: String)          = ES (Left errstr) :: ES a
```

Running ES

Our new monad instance works just fine:

```
> sum2roots (4) (9) :: ES Double
ES (Right 5.0)
> sum2roots (4) (-9) :: ES Double
ES (Left "negative! -9.0")
> sum2roots (-4) (-9) :: ES Double
ES (Left "negative! -4.0")
```

Monad is very generic

When we write code 'monadic-style', using only the Monad class functions then we get highly-reusable code that works with any Monad instance.
This versatility of the class system gives Haskell a lot of its most powerful abstractions.
There is a library of operations `Control.Monad` which specifies many operations in a generic fashion so that they work in any monad

Some practical Monad notes

Let's look at an example
Writing code which unwraps a Monadic value and then applies a function to it right away is fairly common (and tiresome):

```
m >>= \ ls -> return (sum ls)
```

or

```
do ls <- m
   return (sum l)
```

In these cases we can use `liftM` from `Control.Monad`

```
liftM :: (Monad m) => (a -> b) -> m a -> m b

liftM f m = m >>= \ x -> return (f x)
```

So now we can write: `liftM sum m`