Introducing "Monads"

- IO in Haskell uses the `IO` type constructor along with
  - Primitive I/O operations that return an "action value" of type `IO t`
  - I/O combinators "bind" `(>>=)`, "seq" `(>>)` and `return`.
- Haskell goes further, though. It uses Haskell's class system to leverage the key concepts.
- Type `IO t` is an instance of the so-called `Monad` class.
  - The term "monad" comes originally from Greek philosophy, more recent material from Leibniz, and even more recently from Category Theory.
- We shall see that the monad concept goes beyond I/O and has much wider utility.

Monads in Haskell

Monads in Haskell are represented by a type class:

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

Since `>>` can be defined in terms of `>>=` we usually only need to provide instances for `return` and `>>=`.

The fourth member of the class, `fail`, is an error handling operation which takes an error message and causes the chain of functions to fail, perhaps by using `error` to halt the program

Monads in Haskell, 7.10 onwards

In fact the declaration of the Monad class in Haskell now has the form:

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

- The annotation `(m ::  * -> *)` simply says that `m` is a type-constructor, not a type (View `*` as representing a type argument or result).
- We shall ignore this for this course, as it has no effect on what is to come.

do is *syntactic sugar*

There is a mechanical translation from the do-notation form to the combinator form, which we can summarize:

```
do { a1 ; a2 ; .. ; an }
  ⇝ a1 >> do {a2 ; .. ; an }

do { x <- a1 ; a2 ; .. ; an }
  ⇝ a1 >>= \ x -> do {a2 ; .. ; an }

do a          ⇝      a
```

Note that above we show the full Haskell syntax for do-notation with explicit }, ; and }, rather than relying on the offside-rule.

The Monad laws

In order to retain the semantics that we want, any implementation of a monad is required to follow these rules:

```
(return v >>= f)    ==  f v

f >>= return        ==  f

(x >>= f) >>= g     ==  x >>= (\ a -> f a >>= g)
```

These laws are not checked by the compiler.

Any monad?

Any implementation of a monad?

Yes, monads represent something fundamental in computation, the idea of connecting two computations by sequencing them.

There are more monads than just `IO a`.

For example, another monad which we have already seen is `Maybe`!

Using the `Maybe` monad

Imagine a function:

```
f dict = case (lookup "foo" dict) of
           Nothing  ->  Nothing
           Just x   ->  case (lookup "bar" dict) of
                          Nothing  ->  Nothing
                          Just y   ->  Just (x,y)
```

We can clean this up because "Maybe" is a monad!

```
f dict
 = do x <- lookup "foo" dict
      y <- lookup "bar" dict
      return (x,y)
```

Let's think about how we can define `>>=` and `return` so that this code behaves like the code above.

The relevant definitions?

- We make the `Maybe` Type constructor an instance of the `Monad` class:
  `instance Monad Maybe where`
- To return a result we wrap it with `Just`:
  `return x = Just x`
- In bind, if a previous function returns `Nothing` we simply propagate this:
  `Nothing >>= f = Nothing`
- If a previous function returned `Just` something we apply the (monadic) function to it:
  `(Just x) >>= f = f x`
- If we want to report an error (fail), we produce `Nothing`:
  `fail s = Nothing`
- All of this is in the standard prelude

Maybe forms a monad?

- It represents the type of computations that may succeed or fail.
- More specifically, it combines actions by trying the first, and applying the second if the first succeeded (produced a `Just` result).
- `Maybe a` is the type of short-circuiting computations which can produce an `a`.
- There are no "side-effects" here — so monads are not just a way to hide those.

What's actually happening?

Let's *desugar* the monadic program and translate it into ordinary functions.

```
f dict
 = do x <- lookup "foo" dict
      y <- lookup "bar" dict
      return (x,y)
```

```
f dict = lookup "foo" dict >>= (\ x ->
         lookup "bar" dict >>= (\ y ->
         Just (x, y) ) )
```

What's actually happening?

What's going to happen if the first lookup fails?

```
f dict = Nothing >>= (\ x ->
         lookup "bar" dict >>= (\ y ->
         Just (x, y) ) )
```

```
f dict = Nothing
```

How about the second?

```
f dict = lookup "foo" dict >>= (\ x ->
         Nothing >>= (\ y ->
         Just (x, y) ) )
```

```
f dict = lookup "foo" dict >>= (\ x -> Nothing )
```

```
f dict = Nothing
```

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 ]
  ```