Monad: **an abstraction; mechanism for performing computations that have side effects**
Why do we need them?

- They maintain referential transparency
- Example:

Violating referential transparency!

It doesn't take much to see the problem. Do we know what this will do:

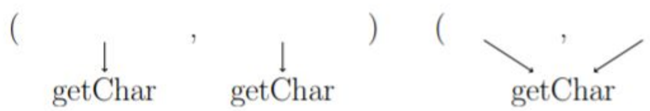
```
f1 = (primGetChar,primGetChar)
```

How about this:

```
f2 = let x = primGetChar in (x,x)
```

Violating referential transparency!

If we draw the graphs of `f1` and `f2` we can see the problem.



- IO brings in the concept of a token to reflect the state of the world beyond the program
    - Achieved by enforcing a structure onto all functions which perform IO operations
    - Can then hide all the plumbing into simple infix functions (e.g. >> to discard the result, >>= to pass it on)
    - `(>>=) :: IO a -> (a -> IO b) -> IO b`
    - `(>>) l r = l >>= (\_ -> r)`
- Adding return gives us a wrapper function (a -> IO a)

So a Monad is an abstraction that represents a computation, which has results.

- Provides return, and >>= (bind) at least
- Typically has primitive functions to make it useful e.g. getChar in IO
- Enables syntactic sugar of the 'do notation'
- Mechanical Translations of the do notation using >> and >>= ...

```
  do x
     y
=
  x >> do y

do a <- x
   y
=
  x >>= \a -> do y

do x = x
```

-

Other Monads

- Maybe

```haskell
return x = Just x

Nothing  >>= k = Nothing
(Just x) >>= k = k x
```

-
- Pre GHC 7.10 (March 2015) Monad class:

```haskell
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
```

    -
    - A fail implementation for Maybe might be: fail _ = Nothing
- State
    - Similar to the concept of a world from IO except provides controlled access
    - newtype State s a = State a -> (a,s)
        - Where s is the type of the state we carry around, and a is the result of our stateful computations
        - E.g. f :: State String Int is some computation maintaining a state of type string, and computing an Int
        - For convenience this is written as a record

```haskell
newtype State s a = State {
    runState :: s -> (a, s)
    }
```

            -
        - This becomes an instance of Monad with a few simple declarations

```haskell
instance Monad (State s) where
    return a = State (\s -> (a,s))
```

Combining two stateful actions looks like this:

```haskell
    m >>= k = State (\s -> let (a,s') = runState m s
                           in runState (k a) s')
```

            -
        - Not very useful if we can't access the state…

```haskell
get :: State s s
get = State $ \s -> (s,s)

put :: s -> State s ()
put s = State $ \_ -> ((),s)
```

            -
        - System.Random provides a random number generator (in IO monad because based off external param e.g. System.Time)
        - System.Random is a natural fit for state because it returns a generator on each generation to prevent repeated numbers...

I will create a set of State monad based wrappers for the Generator actions

```haskell
randomRState :: (Random a) => (a,a) -> State StdGen a
randomRState bounds = do g <- get
                         (x,g') <- return $ randomR bounds g
                         put g'
                         return x
```

Actually, this is a bit long winded:

```haskell
randomRState bounds = State (randomR bounds)
```

is sufficient!

So we wrap the generators in the state:

```haskell
randomState :: (RandomGen g, Random a) => State g a
randomState = State random

randomRState :: (RandomGen g, Random a) => (a,a) -> State g a
randomRState bounds = State (randomR bounds)
```

- randomState and randomRState represent instances of the state monad where the state held is the generator and the value held is a number (or a bounds)
- So what is the State monad actually?
    - The monadic instance is defined as *State s* which means in functions with a the type signature with m a in it, the m is *State s,* which allows State actions to be chained (m a -> m b for example) without changing the state
    - *State s a* and *m a* then is actually just *someFunc :: s -> (a,s)*
    - What is runState? *runState gives us access to our state constructor, i.e. to someFunc :: s -> (a,s)*
    - From: http://brandon.si/code/the-state-monad-a-tutorial-for-the-confused/

- Other 'Standard' Monads
    - []

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

- Why are lists an instance of Monad?
    - Compare to maybe: computation in the maybe monad can return something or nothing, computation in the list monad can return nothing ([]), something, or some **things**
    - Semantics of bind:
        - Type signature: [a] -> (a -> [b]) -> [b]
        - Implementation is given above
        - "pulls out the values from the list to give them to a function that produces a new list."
        - As type-def shows, values are given to a function which converts individual items into a list of bs, because of this, use concat (map f lst), as we want [b] not [[b]]

Monad Laws

- Left identity: `return a >>= f = f a`
- Right identity: `m >>= return = m`
- Associativity: `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

- Quite intuitive when thought about

Applicative Functors

- Came along with GHC 7.10 (March 2015) to add extra abstraction
- The monad class was refactored to **Functor, Applicative, and Monad**
- **Functors**
    - Can apply to ‘wrapped’ values

      ```haskell
      class Functor f where
          fmap :: (a -> b) -> (f a -> f b)
      ```

      Taking `Maybe` as our example, it can be made an instance of `Functor`

      ```haskell
      instance Functor Maybe where
        fmap f Nothing  = Nothing
        fmap f (Just a) = Just (f a)
      ```
    - <$> = fmap (infix notation)
    - **Functor laws**

      ```haskell
      fmap id      = id
      fmap (g . h) = fmap g . fmap h
      ```
- **Applicatives**
    - More structure than a functor, less than a monad
    - Say we want to apply more args to fmap than one

      ```haskell
      fmap2 :: (a -> b -> c) -> f a -> f b -> f c
      fmap2 (+) (Just 1) (Just 2)        --
      ```
        - But why stop at 2…
    - **Applicative**

      ```haskell
      class Functor f => Applicative f where
          pure :: a -> f a
          (<*>) :: f (a -> b) -> f a -> f b
      ```

      we can now write `fmap2 g x y = pure g <*> x <*> y`
- **The 7.10 Haskell Monad**

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

    (>>) :: m a -> m b -> m b
    m >> k = m >>= \_ -> k

    return :: a -> m a
    return = pure

    fail :: String -> m a
    fail s = errorWithoutStackTrace s
  ```

It's good to know the following. Given that:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

There is a utility function in the Prelude called `liftM` which "lifts" a function into a monad:

```
liftM   :: (Monad f) => (a -> b) -> f a -> f b
liftM f m1              = do { x1 <- m1; return (f x1) }
```

in many (most) cases where you are creating a Monad instance `fmap = liftM` for your monad.

So the complete (modern) set of instances look like this...

```
instance Functor (State s) where
  fmap = liftM

instance Applicative (State s) where
  pure a = State (\s -> (a,s))

instance Monad (State s) where
  m >>= k = State (\s -> let (a,s') = runState m s
                         in runState (k a) s')
```

-

Classes in Haskell

- The type of equality
    - Naturally: (==) :: a -> a -> Bool
    - What does haskell say: (==) :: (Eq a) => a -> a -> Bool
- **Ad-Hoc polymorphism**
    - Equality is polymorphic [(==) :: a -> a -> Bool]
    - **However it is ad-hoc - there has to be a specific implementation for each type**
    - Contrast with (parametric) polymorphism of length:
        - length [] = 0
        - length (x:xs) = 1 + length xs
        - **This is guaranteed to work regardless of the list provided: no type dependency**
    - Ad-hoc polymorphism is ubiquitous
        - E.g. + is used to denote many different but related operators (a.k.a overloading)
        - Overloading is built into many languages, **but is a language feature of haskell called "type classes" - can create new instances**
    - **Defining (Type-) Classes in Haskell (Overloading)**
        - To define our own name/operator overloading, we:
            - **Specify** the **name/operator** involved (==)
            - **Describe its pattern** of use (e.g. a -> a -> Bool)
            - **Provide an overarching 'class' name for the concept (e.g. Eq)**
        - To use our operator with a given type (e.g. Bool) we **provide the implementation for that type** - instance of that type
    - **Equality Class**
        - `class Eq a where` <- introduce as a class characterising a type (a)
          `    (==) :: a -> a -> Bool` <- declares that a type of this class needs an implementation of (==) matching the signature provided

- ```
  instance Eq Bool where          --
    True  == True  = True         --
    False == False = True         --
    _     == _     = False        --
  ```
- The real equality class
    - ```
      class Eq a where                          --
        (==), (/=) :: a -> a -> Bool             --
        --minimum complete def: == or /=         --
        x == y = not (x /= y)                    --
        x /= y = not (x == y)                    --
      ```
    - Note: circular definitions of == and /=, so we only define one, and the other is derived
    - Might want to make both explicit for efficiency

- **How Haskell handles a class name/operator**
    - Views the symbol, notes its association to a class, then deduces the type of the args (assuming well-typed), verifies that it has an instance of the class for such type, and generates appropriate code for it
    - If not well typed?
        - ```
          No instance for (Eq MyType)
            arising from a use of `==' at ...
          Possible fix: add an instance declaration for (Eq MyType)
          ```
- **Standard (Prelude) Classes in Haskell**
    - Relation: Eq, Ord
    - Enumeration: Enum, Bounded
    - Numeric: Num, Real, Integral, Fractional, Floating, RealFrac, RealFloat
    - Textual: Show, Read
    - Categorical: Functor, Monad
          ```