

Abstraction patterns

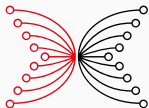
Haskell and Cryptocurrencies

Dr. Lars Brünjes, IOHK

Alejandro Garcia, IOHK

Dr. Andres Löb, Well-Typed LLP

2020-08-03



INPUT | OUTPUT

- Introduce `Monad` and `Applicative`.

Maybe

The `Maybe` type

```
data Maybe a = Nothing
              | Just a
```

The `Maybe` datatype is often used to encode failure or an exceptional value:

```
lookup :: (Eq a) => a -> [(a, b)] -> Maybe b
find    :: (a -> Bool) -> [a] -> Maybe a
```

Encoding exceptions using **Maybe**

Assume that we have a data structure with the following operations:

```
up, down, right :: Loc -> Maybe Loc  
update          :: (Int -> Int) -> Loc -> Loc
```

Given a location **l1**, we want to move up, right, down, and update the resulting position with using **update (+ 1)** ...

Each of the steps can fail.

Encoding exceptions using **Maybe** (contd.)

```
case up l1 of
  Nothing -> Nothing
  Just l2  -> case right l2 of
    Nothing -> Nothing
    Just l3  -> case down l3 of
      Nothing -> Nothing
      Just l4  -> Just (update (+ 1) l4)
```

Encoding exceptions using `Maybe` (contd.)

```
case up l1 of
  Nothing -> Nothing
  Just l2  -> case right l2 of
    Nothing -> Nothing
    Just l3  -> case down l3 of
      Nothing -> Nothing
      Just l4  -> Just (update (+ 1) l4)
```

Encoding exceptions using `Maybe` (contd.)

```
case up l1 of
  Nothing -> Nothing
  Just l2  -> case right l2 of
    Nothing -> Nothing
    Just l3  -> case down l3 of
      Nothing -> Nothing
      Just l4  -> Just (update (+ 1) l4)
```

In essence, we need

- a way to *sequence* function calls and use their results if successful
- a way to *modify* or *produce* successful results.

Encoding exceptions using `Maybe` (contd.)

```
case up l1 of
  Nothing -> Nothing
  Just l2  -> case right l2 of
    Nothing -> Nothing
    Just l3  -> case down l3 of
      Nothing -> Nothing
      Just l4  -> Just (update (+ 1) l4)
```

Sequencing:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
f >>= g = case f of
  Nothing -> Nothing
  Just x   -> g x
```

Encoding exceptions using `Maybe` (contd.)

```
up l1 >>=

\ l2    -> case right l2 of
  Nothing -> Nothing
  Just l3  -> case down l3 of
    Nothing -> Nothing
    Just l4  -> Just (update (+ 1) l4)
```

Sequencing:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
f >>= g = case f of
  Nothing -> Nothing
  Just x   -> g x
```

Encoding exceptions using `Maybe` (contd.)

```
up l1 >>=

\ l2    -> right l2 >>=

\ l3    -> case down l3 of
  Nothing -> Nothing
  Just l4  -> Just (update (+ 1) l4)
```

Sequencing:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
f >>= g = case f of
  Nothing -> Nothing
  Just x   -> g x
```

Encoding exceptions using `Maybe` (contd.)

```
up l1 >>=
  \ l2    -> right l2 >>=
    \ l3    -> down l3 >>=
      \ l4    -> Just (update (+ 1) l4)
```

Sequencing:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
f >>= g = case f of
    Nothing -> Nothing
    Just x   -> g x
```

Sequencing and embedding

```
up l1 >>=  
  \ l2 -> right l2 >>=  
    \ l3 -> down l3 >>=  
      \ l4 -> Just (update (+ 1) l4)
```

Sequencing and embedding

```
up l1 >>=  
  \ l2 -> right l2 >>=  
  \ l3 -> down l3 >>=  
  \ l4 -> return (update (+ 1) l4)
```

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b  
f >>= g   = case f of  
             Nothing -> Nothing  
             Just x   -> g x  
  
return :: a -> Maybe a  
return x = Just x
```

Sequencing and embedding

```
up l1 >>=  
  \ l2 -> right l2 >>=  
    \ l3 -> down l3 >>=  
      \ l4 -> return (update (+ 1) l4)
```

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
f >>= g    = case f of  
              Nothing -> Nothing  
              Just x   -> g x
```

```
return :: a -> Maybe a
```

```
return x = Just x
```

```
(up l1) >>= right >>= down >>= return . update (+ 1)
```

Observation

Code looks a bit like imperative code. Compare:

```
up l1    >>= \ l2 ->  
right l2 >>= \ l3 ->  
down l3  >>= \ l4 ->  
return (update (+ 1) l4)
```

```
l2 := up l1;  
l3 := right l2;  
l4 := down l3;  
return update (+ 1) l4
```

- In the imperative language, the occurrence of possible exceptions is a side effect.
- Haskell is more explicit because we use the `Maybe` type and the appropriate sequencing operation.

A variation: **Either**

Compare the datatypes

```
data Either a b = Left a | Right b
data Maybe a    = Nothing | Just a
```

A variation: `Either`

Compare the datatypes

```
data Either a b = Left a | Right b
data Maybe a    = Nothing | Just a
```

The datatype `Maybe` can encode exceptional function results (i.e., failure), but no information can be associated with `Nothing`. We cannot distinguish different kinds of errors.

A variation: `Either`

Compare the datatypes

```
data Either a b = Left a | Right b
data Maybe a    = Nothing | Just a
```

The datatype `Maybe` can encode exceptional function results (i.e., failure), but no information can be associated with `Nothing`. We cannot distinguish different kinds of errors.

Using `Either`, we can use `Left` to encode errors, and `Right` to encode successful results.

Sequencing and returning for `Either`

We can define variants of the operations for `Maybe`:

```
(>>=) :: Either e a -> (a -> Either e b)
      -> Either e b
f >>= g = case f of
    Left  e -> Left e
    Right x -> g x
return :: a -> Either e a
return x = Right x
```

Simulating exceptions

We can abstract completely from the definition of the underlying `Either` type if we define functions to throw and catch errors.

```
throwError :: e -> Either e a  
throwError e = Left e
```

Simulating exceptions

We can abstract completely from the definition of the underlying `Either` type if we define functions to throw and catch errors.

```
throwError :: e -> Either e a
throwError e = Left e

catchError :: Either e a          ->      -- computation
              (e -> Either e a) ->      -- handler
              Either e a

catchError f handler = case f of
    Left  e -> handler e
    Right x -> Right x
```

State

Maintaining state explicitly

- We pass state to a function as an argument.
- The function modifies the state and produces it as a result.
- If the function does anything except modifying the state, we must return a tuple (or a special-purpose datatype with multiple fields).

This motivates the following type definition:

```
type State s a = s -> (a, s)
```


Using state

There are many situations where maintaining state is useful:

- using a random number generator

```
type Random a = State StdGen a
```

- using a counter to generate unique labels

```
type Counter a = State Int a
```

- maintaining the complete current configuration of an application (an interpreter, a game, ...) using a user-defined datatype

```
data ProgramState = ...  
type Program a = State ProgramState a
```

Example: labelling the leaves of a tree

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
labelTree :: Tree a -> State Int (Tree (a, Int))  
labelTree (Leaf x)    c  = (Leaf (x, c), c + 1)  
labelTree (Node l r) c1 =  
    let (ll, c2) = labelTree l c1  
        (lr, c3) = labelTree r c2  
    in (Node ll lr, c3)
```

Encoding state passing

```
\ s1 -> let (lvl , s2) = generateLevel      s1  
          (lvl', s3) = generateStairs lvl s2  
          (ms   , s4) = placeMonsters lvl' s3  
    in (combine lvl' ms, s4)
```

Encoding state passing

```
\ s1 -> let (lvl, s2) = generateLevel s1  
           (lvl', s3) = generateStairs lvl s2  
           (ms, s4) = placeMonsters lvl' s3  
in (combine lvl' ms, s4)
```

Encoding state passing

```
\ s1 -> let (lvl , s2) = generateLevel      s1  
          (lvl', s3) = generateStairs lvl s2  
          (ms   , s4) = placeMonsters lvl' s3  
        in (combine lvl' ms, s4)
```

Again, we need

- a way to *sequence* function calls and use their results
- a way to *modify* or *produce* successful results.

Bind and return for state

```
\ s1 -> let (lvl , s2) = generateLevel      s1
          (lvl', s3) = generateStairs lvl s2
          (ms  , s4) = placeMonsters lvl' s3
          in (combine lvl' ms, s4)
```

```
(>=>) :: State s a -> (a -> State s b) -> State s b
f >=> g  = \ s -> let (x, s') = f s in g x s'

return :: a -> State s a
return x = \ s -> (x, s)
```

Bind and return for state

```
                                generateLevel      >>= \ lvl ->  
\ s2 -> let (lvl', s3) = generateStairs lvl s2  
        (ms   , s4) = placeMonsters lvl' s3  
        in (combine lvl' ms, s4)
```

```
(>>=) :: State s a -> (a -> State s b) -> State s b  
f >>= g   = \ s -> let (x, s') = f s in g x s'  
  
return :: a -> State s a  
return x = \ s -> (x, s)
```

Bind and return for state

```
generateLevel      >>= \ lvl ->  
generateStairs lvl >>= \ lvl' ->  
\ s3 -> let (ms  , s4) = placeMonsters lvl' s3  
      in (combine lvl' ms, s4)
```

```
(>>=) :: State s a -> (a -> State s b) -> State s b  
f >>= g  = \ s -> let (x, s') = f s in g x s'  
  
return :: a -> State s a  
return x = \ s -> (x, s)
```


Bind and return for state

```
generateLevel      >>= \ lvl ->  
generateStairs lvl >>= \ lvl' ->  
placeMonsters lvl' >>= \ ms ->  
\ s4 ->    (combine lvl' ms, s4)
```

```
(>>=) :: State s a -> (a -> State s b) -> State s b  
f >>= g   = \ s -> let (x, s') = f s in g x s'  
  
return :: a -> State s a  
return x = \ s -> (x, s)
```

Bind and return for state

```
generateLevel      >>= \ lvl ->  
generateStairs lvl >>= \ lvl' ->  
placeMonsters lvl' >>= \ ms ->  
return (combine lvl' ms)
```

```
(>>=) :: State s a -> (a -> State s b) -> State s b  
f >>= g   = \ s -> let (x, s') = f s in g x s'  
  
return :: a -> State s a  
return x = \ s -> (x, s)
```

Observation

Again, the code looks a bit like imperative code. Compare:

```
generateLevel      >>= \ lvl ->  
generateStairs lvl >>= \ lvl' ->  
placeMonsters lvl' >>= \ ms ->  
return (combine lvl' ms)
```

```
lvl := generateLevel;  
lvl' := generateStairs lvl;  
ms := placeMonsters lvl';  
return combine lvl' ms
```

- In the imperative language, the occurrence of memory updates (random numbers) is a side effect.
- Haskell is more explicit because we use the `State` type and the appropriate sequencing operation.

“Primitive” operations for state handling

We can completely hide the implementation of `State` if we provide the following two operations as an interface:

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

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

```
inc :: State Int ()
inc = get >>= \ s -> put (s + 1)
```

Labelling a tree, revisited

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
labelTree :: Tree a -> State Int (Tree (a, Int))  
labelTree (Leaf x) c = (Leaf (x, c), c + 1)
```

```
labelTree (Node l r) c1 =  
  let (ll, c2) = labelTree l c1  
      (lr, c3) = labelTree r c2  
  in (Node ll lr, c3)
```

The old version, with tedious explicit threading of the state.

Labelling a tree, revisited

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
labelTree :: Tree a -> State Int (Tree (a, Int))
labelTree (Leaf x)    = get >>= \ c ->
                        inc >> return (Leaf (x, c))
labelTree (Node l r) =
  labelTree l >>= \ ll  ->
  labelTree r >>= \ lr  ->
  return (Node ll lr)
```

```
(>>) :: State s a -> State s b -> State s b
x >> y = x >>= \ _ -> y
```

(The same definition as for IO ...)

List

Encoding multiple results and nondeterminism

Get the length of all words in a list of multi-line texts:

```
map length  
  (concat (map words  
            (concat (map lines txts))  
          ))
```


Encoding multiple results and nondeterminism

Get the length of all words in a list of multi-line texts:

```
map length  
  (concat (map words  
            (concat (map lines txts))  
          ))
```

Embedding and sequencing for computations with many results *nondeterministic computations*:

Encoding multiple results and nondeterminism

Get the length of all words in a list of multi-line texts:

```
map length  
  (concat (map words  
            (concat (map lines txts))  
          ))
```

Embedding and sequencing for computations with many results *nondeterministic computations*:

- Embedding: a computation with exactly one result.
- Sequencing: performing the second computation on all possible results of the first one.

Defining bind and return for lists

```
(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f = concat (map f xs)
return :: a -> [a]
return x = [x]
```

We have to use `concat` in `(>>=)` to flatten the list of lists.

Using bind and return for lists

```
map length  
  (concat (map words  
            (concat (map lines txts)))))
```

```
txts    >>= \ t ->  
lines t >>= \ l ->  
words l >>= \ w ->  
return (length w)
```

Using bind and return for lists

```
map length  
  (concat (map words  
            (concat (map lines txts)))))
```

<code>txts</code>	<code>>>= \ t -></code>	<code>t := txts</code>
<code>lines t</code>	<code>>>= \ l -></code>	<code>l := lines t</code>
<code>words l</code>	<code>>>= \ w -></code>	<code>w := words w</code>
<code>return (length w)</code>		<code>return length w</code>

Using bind and return for lists

```
map length  
  (concat (map words  
            (concat (map lines txts)))))
```

<code>txts</code>	<code>>>= \ t -></code>	<code>t := txts</code>
<code>lines t</code>	<code>>>= \ l -></code>	<code>l := lines t</code>
<code>words l</code>	<code>>>= \ w -></code>	<code>w := words w</code>
<code>return (length w)</code>		<code>return length w</code>

- Again, we have a similarity to imperative code.
- Imperative language: implicit nondeterminism.
- Haskell: explicit by using the list datatype and `(>>=)`.

Intermediate Summary

At least four types with `(>>=)` and `return`:

- `Maybe`: `(>>=)` sequences operations that may fail and shortcuts evaluation once failure occurs; `return` embeds a function that never fails;
- `State`: `(>>=)` sequences operations that may modify some state and threads the state through the operations; `return` embeds a function that never modifies the state;
- `[]`: `(>>=)` sequences operations that may have multiple results and executes subsequent operations for each of the previous results; `return` embeds a function that only ever has one result.
- `IO`: `(>>=)` sequences the side effects to the outside world, and `return` embeds a function without any side effects.

Monads

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

- The name “monad” is borrowed from category theory.
- A monad is an algebraic structure similar to a *monoid*.
- Monads have been popularized in functional programming via the work of Moggi and Wadler.

Instances

```
instance Monad Maybe where
  ...
instance Monad (Either e) where
  ...
instance Monad [] where
  ...
newtype State s a = State {runState :: s -> (a, s)}
instance Monad (State s) where
  ...
```

Instances

```
instance Monad Maybe where
  ...
instance Monad (Either e) where
  ...
instance Monad [] where
  ...
newtype State s a = State {runState :: s -> (a, s)}
instance Monad (State s) where
  ...
```

The **newtype** for **State** is required because Haskell does not allow us to directly make a type **s -> (a, s)** an instance of **Monad**. (Question: why not?)

There are more monads

The types we have seen: `Maybe`, `Either`, `[]`, `State`, `IO` are among the most frequently used monads – but there are many more you will encounter sooner or later.

There are more monads

The types we have seen: `Maybe`, `Either`, `[]`, `State`, `IO` are among the most frequently used monads – but there are many more you will encounter sooner or later.

In fact, we have already seen one more! Which one?

There are more monads

The types we have seen: `Maybe`, `Either`, `[]`, `State`, `IO` are among the most frequently used monads – but there are many more you will encounter sooner or later.

In fact, we have already seen one more! Which one?

The generators `Gen` from QuickCheck form a monad. You can see it as an abstract state monad, allowing access to the state of a random number generator.

Monad laws

`return` is the unit of `(>>=)`

```
return a >>= f = f a  
m >>= return  = m
```

Associativity of `(>>=)`

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

Monad laws for **Maybe**

```
return a >>= f
= { Definition of (>>=) }
  case return a of
    Nothing -> Nothing
    Just x   -> f x
= { Definition of return }
  case Just a of
    Nothing -> Nothing
    Just x   -> f x
= { case }
  f a
```


Monad laws for **Maybe** (contd.)

```
m >>= return
= { Definition of (>>=) }
  case m of
    Nothing -> Nothing
    Just x   -> return x
= { Definition of return }
  case m of
    Nothing -> Nothing
    Just x   -> Just x
= { case }
  m
```

Monad laws for `Maybe` (contd.)

Lemma

```
forall ((f :: a -> Maybe b)) .  
Nothing >=> f = Nothing
```

Proof

```
Nothing >=> f  
= { Definition of (>=>) }  
  case Nothing of  
    Nothing -> Nothing  
    Just x   -> f x  
= { case }  
  Nothing
```

Monad laws for `Maybe` (contd.)

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

Induction on `m`. Case `m` is `Nothing`:

```
(Nothing >=> f) >=> g
= { Lemma }
  Nothing >=> g
= { Lemma }
  Nothing
= { Lemma }
  Nothing >=> (\ x -> f x >=> g)
```

Monad laws for `Maybe` (contd.)

Case `m` is `Just y` :

```
(Just y >>= f) >>= g
= { Definition of (>>=) }
  (case Just y of
    Nothing -> Nothing
    Just x   -> f x) >>= g
= { case }
  f y >>= g
= { beta-expansion }
  (\ x -> f x >>= g) y
= { case }
  case Just y of
    Nothing -> Nothing
    Just x   -> (\ x -> f x >>= g) x
= { definition of (>>=) }
  Just y >>= (\ x -> f x >>= g)
```

Additional monad operations

Class `Monad` contains an additional method, with a default:

```
class Applicative m => Monad m where
  ...
  (>>) :: m a -> m b -> m b
  m >> n = m >>= \ _ -> n
```

The `do` notation we have introduced when discussing `IO` is available for all monads:

```
generateLevel      >>= \lvl ->  
generateStairs lvl >>= \lvl' ->  
placeMonsters lvl' >>= \ms ->  
return (combine lvl' ms)
```

```
do  
  lvl <- generateLevel  
  lvl' <- generateStairs lvl  
  ms <- placeMonsters lvl'  
  return (combine lvl' ms)
```

```
up l1    >>= \ l2 ->  
right l2 >>= \ l3 ->  
down l3  >>= \ l4 ->  
return (update (+ 1) l4)
```

```
do  
  l2 <- up l1  
  l3 <- right l2  
  l4 <- down l3  
  return (update (+ 1) l4)
```

Tree labelling, revisited once more

Using `Control.Monad.State` and `do` notation:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
labelTree :: Tree a -> State Int (Tree (a, Int))
labelTree (Leaf x) = do
  c <- get
  put (c + 1) -- or modify (+ 1)
  return (Leaf (x, c))
labelTree (Node l r) = do
  ll <- labelTree l
  lr <- labelTree r
  return (Node ll lr)
```

How to get at the final tree?

Running a stateful computation

```
evalState :: State s a -> s -> a
```

Running a stateful computation

```
evalState :: State s a -> s -> a
```

```
labelTreeFrom0 :: Tree a -> Tree (a, Int)  
labelTreeFrom0 t = evalState (labelTree t) 0
```

Running a stateful computation

```
evalState :: State s a -> s -> a
```

```
labelTreeFrom0 :: Tree a -> Tree (a, Int)  
labelTreeFrom0 t = evalState (labelTree t) 0
```

There's also

```
runState :: State s a -> s -> (a, s)
```

(which is just unpacking `State`'s `newtype` wrapper).

List comprehensions

```
map length  
  (concat (map words (concat (map lines txts)))))
```

```
do  
  t <- txts  
  l <- lines t  
  w <- words l  
  return (length w)
```

Also *list comprehensions*:

```
[length w | t <- txts, l <- lines t, w <- words l]
```

More on `do` notation (and list comprehensions)

- Use it, the special syntax is usually more concise.
- Never forget that it is just syntactic sugar. Use `(>=)` and `(>>)` directly when it is more convenient.

And some things I've already said about `IO`:

- Remember that `return` is just a normal function:
 - Not every `do`-block ends with a `return`.
 - `return` can be used in the middle of a `do`-block, and it doesn't "jump" anywhere.

More on `do` notation (and list comprehensions)

- Use it, the special syntax is usually more concise.
- Never forget that it is just syntactic sugar. Use `(>=)` and `(>>)` directly when it is more convenient.

And some things I've already said about `IO`:

- Remember that `return` is just a normal function:
 - Not every `do`-block ends with a `return`.
 - `return` can be used in the middle of a `do`-block, and it doesn't "jump" anywhere.
- Not every monad computation has to be in a `do`-block. In particular `do e` is the same as `e`.
- On the other hand, you may have to "repeat" the `do` in some places, for instance in the branches of an `if`.

IO vs. other monads

The `IO` monad is special

- `IO` is a primitive type, and `(>>=)` and `return` for `IO` are primitive functions,
- there is no (politically correct) function `runIO :: IO a -> a`, whereas for most other monads there is a corresponding function, or at least some way to get an `a` out of the monad;
- values of `IO a` denote side-effecting programs that can be executed by the run-time system.

Effectful programming

- `IO` being special has little to do with it being a monad;
- you can use `IO` and functions on `IO` very much ignoring the presence of the `Monad` class;
- `IO` is about allowing real side effects to occur; the other types we have seen are entirely pure as far as Haskell is concerned, even though they capture a form of effects.

IO, internally

If you ask GHCi about `IO` by saying `:i IO`, you get

```
newtype IO a
  = GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
    -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
    -- Defined in 'GHC.Types'
```

So internally, GHC models `IO` as a kind of state monad having the “real world” as state!

Monadic operations

The advantages of an abstract interface

Several advantages to identifying the “monad” interface:

- Have to learn fewer names. Same `return` and `(>>=)` (and `do` notation) in many different situations.
- Useful derived functions that only use `return` and `(>>=)`. All these library functions become automatically available for every monad.

The advantages of an abstract interface

Several advantages to identifying the “monad” interface:

- Have to learn fewer names. Same `return` and `(>>=)` (and `do` notation) in many different situations.
- Useful derived functions that only use `return` and `(>>=)`. All these library functions become automatically available for every monad.
- There are many more monads than the ones we've discussed so far. Monads can be combined to form new monads.
- Application-specific code often uses just the monadic interface plus a few extra functions. As such, it is easy to switch the underlying monad of a large part of a program in order to accommodate a new aspect (error handling, logging, backtracking, ...).

Useful monad operations

```
liftM      :: (a -> b) -> IO a -> IO b
mapM       :: (a -> IO b) -> [a] -> IO [b]
mapM_      :: (a -> IO b) -> [a] -> IO ()
forM       :: [a] -> (a -> IO b) -> IO [b]
forM_      :: [a] -> (a -> IO b) -> IO ()
sequence   :: [IO a] -> IO [a]
sequence_  :: [IO a] -> IO ()
forever    :: IO a -> IO b
filterM    :: (a -> IO Bool) -> [a] -> IO [a]
replicateM :: Int -> IO a -> IO [a]
replicateM_ :: Int -> IO a -> IO ()
when       :: Bool -> IO () -> IO ()
unless     :: Bool -> IO () -> IO ()
```

Useful monad operations

```
liftM      :: Monad m => (a -> b) -> m a -> m b
mapM       :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_      :: Monad m => (a -> m b) -> [a] -> m ()
forM       :: Monad m => [a] -> (a -> m b) -> m [b]
forM_      :: Monad m => [a] -> (a -> m b) -> m ()
sequence   :: Monad m => [m a] -> m [a]
sequence_  :: Monad m => [m a] -> m ()
forever    :: Monad m => m a -> m b
filterM    :: Monad m => (a -> m Bool) -> [a] -> m [a]
replicateM :: Monad m => Int -> m a -> m [a]
replicateM_ :: Monad m => Int -> m a -> m ()
when       :: Monad m => Bool -> m () -> m ()
unless    :: Monad m => Bool -> m () -> m ()
```

Example: labelling a rose tree

```
data Rose a = Fork a [Rose a]
```

Each node has a (possibly empty) list of subtrees.

Example: labelling a rose tree

```
data Rose a = Fork a [Rose a]
```

Each node has a (possibly empty) list of subtrees.

```
labelRose :: Rose a -> State Int (Rose (a, Int))  
labelRose (Fork x cs) = do  
  c <- get  
  put (c + 1)  
  lcs <- mapM labelRose cs  
  return (Fork (x, c) lcs)
```

Questions

What do you think these will evaluate to:

```
replicateM 2 [1..3]
mapM return [1..3]
sequence [[1, 2], [3, 4], [5, 6]]
mapM
  (flip lookup [(1, 'x'), (2, 'y'), (3, 'z')]) [1..3]
mapM
  (flip lookup [(1, 'x'), (2, 'y'), (3, 'z')]) [1, 4, 3]
evalState (replicateM_ 5 (modify (+ 2))) >> get) 0
```

A common pattern

Let's once again look at tree labelling:

```
labelTree :: Tree a -> State Int (Tree (a, Int))
labelTree (Leaf x) = do
  c <- get
  put (c + 1) -- or modify (+ 1)
  return (Leaf (x, c))
labelTree (Node l r) = do
  ll <- labelTree l
  lr <- labelTree r
  return (Node ll lr)
```

We are returning an application of (constructor) function `Node` to the results of monadic computations.

A common pattern (contd.)

do

$r_1 \leftarrow \text{comp}_1$

$r_2 \leftarrow \text{comp}_2$

...

$r_n \leftarrow \text{comp}_n$

return (f r_1 r_2 ... r_n)

A common pattern (contd.)

```
do  
  r1 <- comp1  
  r2 <- comp2  
  ...  
  rn <- compn  
  return (f r1 r2 ... rn)
```

This isn't type correct:

```
f comp1 comp2 ... compn
```

A common pattern (contd.)

```
do
  r1 <- comp1
  r2 <- comp2
  ...
  rn <- compn
  return (f r1 r2 ... rn)
```

This isn't type correct:

```
f comp1 comp2 ... compn
```

But we can get close:

```
f <$> comp1 <*> comp2... <*> compn
```

Monadic application

We need a function that's like function application, but works on monadic values:

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf mx = do
  f <- mf
  x <- mx
  return (f x)
```

Monadic application

We need a function that's like function application, but works on monadic values:

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf mx = do
  f <- mf
  x <- mx
  return (f x)
```

Types supporting `return` and `ap` have their own name:

```
class Functor f => Applicative f where
  pure  :: a -> f a           -- like return
  (<*>) :: f (a -> b) -> f a -> f b -- like ap
infixl 4 <*>
```


Functor and Applicative in terms of Monad

```
instance Monad T where ...
```

Requires superclass instances for `Functor` and `Applicative`:

```
instance Functor T where  
  fmap = liftM
```

```
instance Applicative T where  
  pure = return  
  (<*>) = ap
```

Example

```
labelTree :: Tree a -> State Int (Tree (a, Int))
labelTree (Leaf x)  = do
  c <- get
  put (c + 1) -- or modify (+ 1)
  return (Leaf (x, c))
labelTree (Node l r) =
  Node <$> labelTree l <*> labelTree r
```

Exercise: Convince yourself that this is type correct.

Lessons

- The abstraction of monads is useful for a multitude of different types.
- Monads can be seen as tagging computations with effects.
- While `IO` is impure and cannot be defined in Haskell, the other effects we have seen can be modelled in a pure way:
 - exceptions via `Maybe` or `Either`;
 - state via `State`;
 - nondeterminism via `[]`.
- The monad interface offers a large number of useful abstractions that can all be applied to these different scenarios.
- All monads are also **applicative functors** and in particular **functors**. The `(<$>)` and `(<*>)` operations are also useful for structuring effectful code in Haskell.