12 Monads

Exercise 12.1 (*Warm-up*: From Maybe to Monad, `MaybeMonad.hs`).
The `Maybe` type should by now be very familiar. Consider the following function types (some of which we have seen before).

```
maybeMap   :: (a → b) → Maybe a → Maybe b
stripMaybe :: Maybe (Maybe a) → Maybe a
applyMaybe :: (a → Maybe b) → Maybe a → Maybe b
```

1. Give **non-trivial** implementations of these three functions. Again, a *trivial* function is one that always does the same thing no matter the input, so for example

   ```
   maybeMap :: (a → b) → Maybe a → Maybe b
   maybeMap _ _ = Nothing
   ```

   is *trivial*, and so not a correct solution for this exercise.

2. Now that we have seen `Functor` and `Monad`, the types above should look similar to operations from those type classes.

   If you didn't do so already in step 1, implement all three of the above functions making use of the fact that `Maybe` is an instance of `Monad` and `Functor`. I.e., use the function `fmap`, the bind operator `(>>=)` and/or *do-notation*.

   You may also use any function from the extensive list available in the `Control.Monad` module: https://hackage.haskell.org/package/base-4.16.0.0/docs/Control-Monad.html#g:4.

   To check your answers, change the types of the functions, replacing `Maybe` with a type variable `m` that is required to be an instance of `Monad`, e.g. you should be able to change

   ```
   maybeMap   :: (a → b) → Maybe a → Maybe b
   ```

   into

   ```
   monadMap   :: (Monad m) ⇒ (a → b) → m a → m b
   ```

   without needing to change anything (besides the name) of your definition.

(If you get stuck on this exercise, simply cheat using Hoogle.)

Exercise 12.2 (*Warm-up*: Do-notation, `Notation.hs`).
Do-notation can be very useful, but is just syntactic sugar for the bind-operator `(>>=)`, as shown during the lecture.

1. Rewrite the following *IO action* **using** do-notation. (You do not really need to know what `getZonedTime` or `formatTime` do, but you can probably guess.)

```
siri :: IO ()
siri =
  putStrLn "What is your name?" >>
  getLine >>= \name →
  getZonedTime >>= \now →
  putStrLn (name ++ formatTime defaultTimeLocale ", the time is %H:%M" now)
```

2. Rewrite the following function **without** do-notation, **using** the bind operator `(>>=)`.

```
mayLookup :: (Eq a) ⇒ Maybe a → [(a, b)] → Maybe b
mayLookup maybekey assocs = do
  key ← maybekey
  result ← lookup key assocs
  return result
```

   What does this function compute?

Exercise 12.3 (*Warm-up*: Applicatives and Monads, `ApplicativeMonad.hs`).
The type class `Applicative` is a super-class of `Monad`. That means that every monad is also an applicative functor, and we can use `fmap`, `⟨*⟩`, etc. on them as well. Consider this function:

```
liftMaybe2 :: (a → b → c) → Maybe a → Maybe b → Maybe c
liftMaybe2 f (Just x) (Just y) = Just (f x y)
liftMaybe2 _ _        _        = Nothing
```

1. Define this function **without explicit case distinctions**, using the fact that `Maybe` is an instance of `Applicative`. Check your definition by changing its name and type to:

```
liftA2 :: (Applicative m) ⇒ (a → b → c) → m a → m b → m c
```

2. Define this function again, but this time using the fact that `Maybe` is a monad (i.e. use `return`, `(>>=)` and/or *do-notation*). Check your definition by changing its name and type to:

```
liftM2 :: (Monad m) ⇒ (a → b → c) → m a → m b → m c
```

3. Test your `liftA2`/`liftM2` functions. Also try calling them on a *different* monad than `Maybe`:

```
⋙ liftM2 (++) (return "Hi, ") (putStr "name: " >> getLine) -- IO monad
⋙ liftM2 (++) ["Pol","Engelbert"] ["!", "?"]               -- list monad
```

   What do you expect to be the result?

(Note: the 'official' `liftA2` *and* `liftM2` *are defined in* `Control.Applicative` *and* `Control.Monad`*, respectively, and are functionally equivalent for monads. Also see Hint 3)*

Exercise 12.4 (*Warm-up:* Iteration in monads, `Replicate.hs`).
In the lecture, a `replicateM` function (which extracts a value from a monad a given number of times, and collects these as a list inside the monad), was defined as follows:

```
replicateM :: (Monad m) ⇒ Int → m a → m [a]
replicateM 0 mx = return []
replicateM n mx = (:) <$> mx <*> replicateM (n-1) mx
```

Now consider the expressions:

```
e1 = replicateM 4 getLine
e2 = replicateM 4 Nothing
e3 = replicateM 4 (Just 37)
e4 = replicateM 4 [0,1]
```

1. For each of these expressions, determine the monad instance being used.
2. Try to predict what they compute, and then check your answers.
3. Similar to Exercise 12.3, define `replicateM` using *do-notation* instead.

Exercise 12.5 (*Mandatory*: State monad vs IO, `RandomGen.hs`, `RandomState.hs`).
(This exercise is used in Exercise 12.6, but most parts of that exercise can be made independently from this one, and if you want, you can work on that exercise first.)

In the `System.Random` module, the random number generation of Haskell is defined. In Exercise 10.2, we already used the `randomRIO :: (Integer,Integer) → IO Integer` function to get random numbers (within a certain range) using the `IO` monad. But `System.Random` also contains *pure* functions for generating random numbers:

```
type StdGen
mkStdGen  :: Int → StdGen
randomR   :: (Integer,Integer) → (StdGen → (Integer, StdGen))
```

The type `StdGen` represents the state of a random number generator. (We don't need to know what it looks like internally.) The function `mkStdGen` creates such a random number generator using a seed value, while the function `randomR` computes a new random number, given a range and a random number generator state. And since we don't want to compute the same random number twice[1], it also returns a new random number generator state:

```
⋙ let gen = mkStdGen 5
⋙ let (x, gen')  = randomR (1,6) gen
⋙ let (y, gen'') = randomR (1,6) gen'
⋙ x+y
8                                  -- (your GHCi may produce a different result)
```

During the lecture, a *state monad* was defined, which allows stateful programming *without* needing the `IO` monad: (See Hint 1 if you are confused by this syntax.)

```
newtype State a = St { runState :: GlobalState → (a, GlobalState) }
```

[1] https://dilbert.com/strip/2001-10-25

By using `GlobalState = StdGen`, we can use this to put the random number generator inside the state monad:

```
newtype RandomState a = St { runState :: StdGen → (a, StdGen) }
```

Thus, we can *hide* the random number generator state in the monad, instead of having to keep it explicitly around as above. `RandomState` is a monad like `IO`, but instead of *IO actions* like `putStrLn`, `getLine`, and `removeFile`, we only get two (much safer) *state monad actions*:

```
get :: RandomState StdGen
put :: StdGen → RandomState ()
```

Which retrieve (`get`) the current random number generator state from the monad, or set it (`put`) to a new state. Compare this to the *IO actions* manipulating GHC's global random number generator:

```
getStdGen :: IO StdGen
setStdGen :: StdGen → IO ()
```

1. *(Optional)* Write an *IO action* `genRandIntegerIO :: (Integer,Integer) → IO Integer` which is equivalent to `randomRIO`, except that it uses `getStdGen`, `setStdGen`, and `randomR`.

2. Write a *state monad action* `genRandInteger :: (Integer,Integer) → RandomState Integer` which generates a random number using the `RandomState` monad instead of the `IO` monad.

   This can be used to define other state monad actions such as:

   ```
   roll_2d6 :: RandomState Integer
   roll_2d6 = do
     a ← genRandInteger (1,6)
     b ← genRandInteger (1,6)
     return (a+b)
   ```

   Unlike `IO`, evaluating a value in the `RandomState` monad needs a bit more typing, since we need to specify the initial state:

   ```
   >> runState roll_2d6 (mkStdGen 5)
   (8,<RNG state>)
   ```

3. Write an *IO action* `safeR :: RandomState a → IO a` which performs the computation of a `RandomState` action using the current global random number generator state, returning the result via the IO monad. This should change the state of the global random number generator, but nothing else.

   ```
    ⋙ setStdGen (mkStdGen 5)
    ⋙ safeR roll_2d6
   8
    ⋙ safeR roll_2d6
   3
   ```

   (Tip: `safeR` itself is an IO action, so it can perform actions like `setStdGen`, etc.)

Exercise 12.6 (*Mandatory:* Seperating concerns with monads, `Dice.hs`).
In role playing games (such as *Dungeons & Dragons*), many different types of dice are used besides the familiar 6-sided ones[2]. For example, a 20-sided dice is the one most often needed in D&D. It is also common to roll multiple dice and add their results. For example, the abbreviation '2d8+1' means to compute the sum of two 8-sided dice, and adding 1. Sometimes, instead of adding dice, the maximum or minimum of two rolls is used. We can create a minimalistic expression language to capture this:

```
data Expr = Lit Integer | Dice Integer
          | Expr :+: Expr
          | Min Expr Expr | Max Expr Expr
```

Here `Dice k` denotes the result of rolling a k-sided die.

1. Create a function:

   ```
   type DiceAction m = Integer → m Integer
   evalM :: Expr → DiceAction IO → IO Integer
   ```

   which evaluates an expression inside the IO monad, with the results of dice rolls computed by the provided *dice action*. E.g., `evalM (Dice 8 :+: Dice 8 :+: Lit 1) (\k→randomRIO (1,k))` computes the value of '2d8+1' using the global random number generator.

   We repeat the tip from Exercise 4.6: start simple and work in steps; for example, first write an evaluator of the type `evalM :: Expr → IO Integer` which directly uses `randomRIO` to resolve dice rolls. Once that works, introduce the `DiceAction IO` argument.

2. Change the type of the function, removing the hardcoded use of the `IO` monad:

   ```
   evalM :: (Monad m) ⇒ Expr → DiceAction m → m Integer
   ```

   For this to work, you must make sure to remove all *IO actions* from `evalM` (there shouldn't be any left after the previous step!).

3. Using `evalM`, we can define the function `evalRIO`, which uses the global random number generator:

   ```
   evalRIO :: Expr → IO Integer
   evalRIO expr = evalM expr (\k→randomRIO (1,k))
   ```

   By implementing a different *dice action*, define a function:

   ```
   evalIO :: Expr → IO Integer
   ```

   that resolves any necessary dice rolls by asking the user to interactively enter the result of an actual, *physical* dice roll.

   You can use the `read` function to convert a string to an integer, but make sure the value entered is in the correct range.

[2]See https://en.wikipedia.org/wiki/Dice#Polyhedral_dice

4. Using `evalM` and an appropriate dice action, define:

   ```
   evalND :: Expr → [Integer]
   ```

   that lists *all possible outcomes* of an expression. (*Tip:* the type of the dice action in this case is `DiceAction []`, which is the type `Integer → [Integer]`)

   `evalND` can be used to compute an *expected value*:

   ```
   expectation :: (Fractional a) ⇒ Expr → a
   expectation e = avg (evalND e)
     where
     avg :: (Fractional a) ⇒ [Integer] → a
     avg xs = fromIntegral (sum xs) / fromIntegral (length xs)
   ```

   You can also compute a frequency table of all outcomes as in Exercise 5.8:

   ```
   histogram = map (\x→(head x,length x)) . group . sort . evalND
   ```

5. Using `evalM` and Exercise 12.5, define:

   ```
   evalR :: Expr → RandomState Integer
   ```

   which resolves dice rolls through a random number generator *without* using IO, but using the *state monad* of Exercise 12.5 instead. Ideally, it should be the case that: `safeR (evalR e)` computes the same value as `evalRIO e`.

6. Create a function:

   ```
   observed :: (Fractional a) ⇒ Int → Expr → IO a
   ```

   Which aggregates the results of (a large number of) simulated dice throws, and averages the results. You may want use a helper *IO action* of type `Int → Expr → IO [Integer]`.

   If everything is done properly, the results of `observed` should be close to that of `expectation`.

   (Optional: you can also use `RandomState` instead of `IO`—the solution will be mostly the same.)

7. Our expression language is quite limited; for instance, sometimes we need to *subtract* two results, or *divide* a result by a constant. Extend the definition of `Expr` and `evalM` accordingly. (You can choose to use infix or prefix constructors).

   Since we are working with integers, all results of divisions should be **rounded down**; you do *not* need to detect or prevent divisions by zero.

   If other functions we have defined so far are affected by this change, alter them as well.

   Test your functions! For example, the *expected value* of a single 8-sided dice roll is 4.5, but the sum of two 8-sided dice divided by 2, is slightly less at 4.25.

8. *(Optional)* Sometimes, a value is computed by throwing a number of dice and only adding the best *k* outcomes. For example, rolling four 6-sided dice and adding only the three best outcomes. Extend the data type `Expr` to support this as well by adding:

   ```
   data Expr = ... | SumBestOf [Expr] Int | ...
   ```

   and modify `evalM` accordingly.

Exercise 12.7 (*Extra*: Turning a container into a monad, `BtreeMonad.hs`).
Consider again the type of binary *leaf trees*:

```
data Btree a = Tip a | Bin (Btree a) (Btree a)
```

which is an instance of `Functor`:

```
instance Functor Btree where
  fmap f (Tip x)   = Tip (f x)
  fmap f (Bin l r) = Bin (fmap f l) (fmap f r)
```

1. Give an instance of `Applicative` for `Tree`.

2. Give an instance of `Monad` for `Tree`.

Exercise 12.8 (*Extra:* Applicatives vs Monads (*Challenging*), `Result.hs`).
Since `Applicative` is a super-class of `Monad`, every *monad* is also an *applicative functor*. But the reverse is not true.

In Exercise 11.5, we created an instance of `Applicative` for the `Result` data type, which is a more verbose version of the `Maybe` type:

```
data Result a = Okay a | Error [String]
```

and required this instance to collect *all* error messages:

```
⋙ Okay (+1) <*> Okay 5
Okay 6
⋙ Error ["illegal operation"] <*> Okay 5
Error ["illegal operation"]
⋙ Okay (+1) <*> Error ["not a number"]
Error ["not a number"]
⋙ Error ["illegal operation"] <*> Error ["not a number"]
Error ["illegal operation","not a number"]
```

Since `Maybe` is a monad, you may wonder if `Result` is too. If `Result` would also be a monad, then the monad laws (see Hint 2) require:

```
mf <*> mx  =  do { f ← mf; x ← mx; return (f x) }
           =  mx >>= (\f → mx >>= (\x → return (f x)))
```

And so, if `Result` is also a monad, we should get:

```
⋙ do { f ← Okay (+1); x ← Okay 5; return (f x) }
Okay 6
⋙ do { f ← Error ["illegal operation"]; x ← Okay 5; return (f x) }
Error ["illegal operation"]
⋙ do { f ← Okay (+1); x ← Error ["not a number"]; return (f x) }
Error ["not a number"]
⋙ do { f ← Error["illegal operation"]; x ← Error ["not a number"]; return (f x) }
Error ["illegal operation","not a number"]
```

Is it possible to create an instance of `Monad Result`, that will give us the above behaviour? Of course this instance would start with:

```
instance Monad Result where
  Okay value >>= k = k value
  Error msg  >>= ... = ...
```

If not, what needs to change about the definition of `<*>` so that `Result` *can* be turned into a monad? Do you think it is useful to do that, or is `Result` simply better off *not* being a monad?

Exercise 12.9 (*Extra:* Proofs using monad laws).
In various places (such as Exercise 12.3), we have seen that:

```
pure f <*> mx <*> my
```

is equivalent to:

```
do { x <- mx; y <- my; return (f x y) }
```

which is the *do-notation* for:

```
mx >>= (\x -> my >>= (\y -> return (f x y)))
```

Use the monad laws (see Hint 2) to prove this; in particular you will need the rules:

```
Monad-Applicative correspondence
  pure                  = return
  m1 <*> m2             = m1 >>= (\g -> m2 >>= (\x -> return (g x)))
Monad identity & associativity
  return x >>= f        = f x
  m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

Note: `<*>` is left-associative, so `pure f <*> mx <*> my = (pure f <*> mx) <*> my`.

Hints to practitioners 1. When parsing this syntax:

```
newtype State a = St { State :: GlobalState -> (a, GlobalState) }
```

Remember that this introduces a *record type*, and it is syntactic sugar for:

```
newtype State a = St (GlobalState -> (a, GlobalState))

runState :: State a -> (GlobalState -> (a, GlobalState))
runState (St f) = f
```

I.e. `runState` is the accessor function for the single value stored in the `newtype`.

Note that our definition of the `State` type and `MonadState` class is a little simplified—in that we hardcode the `GlobalState`. In the official Haskell libraries `Control.Monad.State`, this state is passed as an extra parameter, and we could use:

```
import Control.Monad.State
type RandomState a = State StdGen a
```

But, the official `State` type is defined in terms of a more generalized kind of state monad, which would take quite some time to explain—time which we rather spend on other aspects of Haskell! Rest assured that in all respects, our simplified state monad as used in this exercise set behaves exactly the same as the 'official one', and you can in fact use the above declarations instead of our `RandomState` module.

Hints to practitioners 2. Not everything is permitted as an instance of `Functor`, `Applicative` or `Monad`; there are certain laws that these should obey. For `Functor`, it is required (and users may assume) that:

```
fmap id      = id
fmap (f . g) = fmap f . fmap g
```

I.e., `fmap` applied to the identity function gives an identity for the functor; and `fmap` respects function composition. There are also (less intuitive) laws for `Applicative`, the most important of which to remember is:

```
pure f ⟨*⟩ x = f <$> x = fmap f x
```

and if the applicative functor is also a monad, the following should hold as well:

```
pure x     = return x
m1 ⟨*⟩ m2  = m1 >>= (\f → m2 >>= (\x → return (f x)))
           = do { f ← m1; x ← m2; return (f x) }
```

For monads, the laws are:

```
return x >>= f           = f x            (left-identity)
f >>= return             = f              (right-identity)
m >>= (\x → k x >>= h) = (m >>= k) >>= h  (associativity)
```

These start making more sense if we re-write them in do-notation:

```
do { x' ← return x; f x' }  = do { f x }
do { x ← f; return x }      = do { f }
do { x ← m; y ← k x; h y } = do { y ← do { x ← m; k x }; h y }
```

Remember that `return`, even in the `IO` monad, has **almost nothing** to do with the `return` statement in other programming languages: it just converts a pure value into an 'impure' value, and **does not** end execution. So in the case of the first monad law, `x` is put into the monad, and then immediately extracted again to be passed to `f`; hence why this should be the same as `f x`. For the second law, a value is extracted from a monad to be immediately put back into the same monad—so we might as well return the original.

To understand why the third law is called associativity, it is perhaps easiest to understand by rewriting it slightly; using the fact that `k = \x→k x`:

```
m >>= (\x → k x >>= h) = (m >>= \x → k x) >>= h
```

That is, this monad law ensures we don't really have to worry about the parenthesis in the above expression.

Hints to practitioners 3. The confusion between `Applicative` and `Monad` was a long-standing sore point in Haskell, Originally, these were separate classes with separate functions. Later, `Applicative` was made a super-class of `Monad` (see https://wiki.haskell.org/Functor-Applicative-Monad_Proposal). This was a good idea, but has caused a lot of redundancy in operations. For example:

```
pure   :: (Applicative f) ⇒ a → f a
return :: (Monad m)       ⇒ a → m a

fmap   :: (Functor f)     ⇒ (a → b) → f a → f b
liftM  :: (Monad m)       ⇒ (a → b) → m a → m b

liftA2 :: (Applicative f) ⇒ (a → b → c) → f a → f b → f c
liftM2 :: (Monad m)       ⇒ (a → b → c) → m a → m b → m c

(⟨*⟩)  :: (Applicative f) ⇒ f (a → b) → f a → f b
ap     :: (Monad m)       ⇒ m (a → b) → m a → m b
```

All these function pairs are equivalent for monads.

So when should you use `Applicative`, and when should you use `Monad`? A general rule is that when using `Monad`, computations have a distinct *sequencing* to them: the left-hand side of `(>>=)` can influence the outcome of the computation on the right-hand side. This is even more clear when using do-notation: `m >>= k = do { x ← m; y ← k x; return y }`.

On the other hand, with an `Applicative` that is *not* a `Monad`, the *sequencing* in an expression like `f <$> m1 ⟨*⟩ m2` is *not specified*—it can be that `m1` and `m2` are completely independent, or even be the case that `m2` influences the result of `m1`.

As a general rule, when what you want to write can be expressed using `Applicative`, use that instead of `Monad`. If you are going to need monad anyway, it doesn't really matter—and you will find many Haskell programs that freely mix `pure` and `(>>=)`.