# 12 More Monads & Applicative functors

**Exercise 12.1** (*Warm-up:* Type instances, `FindDefs.hs`).
Give **non-trivial** function definitions that match the following types:

```
sayIO     :: IO Int → IO String
sayMaybe  :: Maybe Int → Maybe String
mpair     :: (Monad m) ⇒ m a → m b → m (a,b)
weirdBind :: (Monad m) ⇒ Maybe (m a) → (a → m b) → m (Maybe b)
```

For the purpose of this exercise, a *trivial* function is one that always returns the same result no matter the input, that does not terminate, or that produces a run time error when evaluated.

**Exercise 12.2** (*Warm-up:* Kinds, `Kinds.hs`).
Give the kind of each of the following types, type constructors and classes

```
data Pair a = Pair a a

data Tuple a b c = Tuple a b c

data MList m a = Nil | Cons (m (a, MList m a))

newtype Compose f g a = Compose { runCompose :: f (g a) }

class Funkytor f where
  funkymap :: (a → b) → (b → a) → f a b → f b a

class ToList a where
  toList :: a → [a]
```

You can check your answers with Ghci using the `:kind` command.

**Exercise 12.3** (*Warm-up:* Iteration in monads, `Replicate.hs`).
In last week's lecture, a `replicateM` function (which repeats a monadic action a given number of times, and collects the results 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 11.6, define `replicateM` using *do-notation* instead.

**Exercise 12.4** (*Mandatory*: State monad vs IO, `LCG.hs`, `RandomGen.hs`, `RandomState.hs`).
*(This exercise is used in Exercise 12.5, 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. `System.Random` also contains *pure* functions (i.e. without needing `IO`) for generating random numbers; however, in this exercise we will use a slightly simpler 'home-grown' version. In `LCG.hs`, we have implemented a *linear congruential generator.*[1]. This is an old-fashioned and simple method of generating pseudo-random numbers: we take a *seed value*, and every time we want a new pseudo-random number, the seed is updated using the function $x \mapsto (ax + b) \mod m$. In practice it is usually convenient to generate pseudo-random numbers within a certain range. The function `randomRange :: (Int,Int) → Seed → (Int,Seed)` implements this. It takes a lower and upper bound, and generates number only within that range:

```
>>> randomRange (1,6) (mkSeed 42)
(2,mkSeed 1250496027)
```

As you can see, apart from returning the number 2, `randomRange` also returns a new seed value: otherwise we would keep generating the same number over and over again.[2] This does mean that we need to pass this seed value (that gets updated constantly) around in every function that generates random numbers—this is rather cumbersome:

```
>>> let seed = mkSeed 42
>>> let (x,seed') = randomRange (1,6) seed
>>> ley (y,_)     = randomRange (1,6) seed'
>>> x+y
8
```

During the lecture, a *state monad* was defined, which allows stateful programming *without* needing the `IO` monad: (Also see Hint 2)

```
newtype State a = St (GlobalState → (a, GlobalState))
```

Together with the evaluation function `runState :: State a → GlobalState → (a, GlobalState)` that 'runs' the *state monad action* on a user-provided initial state. By using `GlobalState = Seed`, we can use this to put the random seed inside the state monad:

```
newtype RandomState a = St (Seed → (a, Seed))
```

This way, the seed value can remain in the background, 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*:

```
getState :: RandomState Seed
putState :: Seed → RandomState ()
```

Which retrieve (`getState`) the current seed value from the monad, or update it (with `putState`) to a new state.

[1] https://en.wikipedia.org/wiki/Linear_congruential_generator
[2] https://dilbert.com/strip/2001-10-25

1. Write a *state monad action* `getRandomRange :: (Int,Int) → RandomState Int` which generates a pseudo-random number using `randomRange`, but that keeps the random seed tucked away in the `RandomState` monad using `getState` and `putState`. *(Friendly note: you can get your code to compile without using `putState`, but your function will have a serious problem that you will notice later on.)*

   We want to use this function just like `randomRIO`, but in state monad actions such as:

   ```
   roll_2d6 :: RandomState Int
   roll_2d6 = do
     a ← getRandomRange (1,6)
     b ← getRandomRange (1,6)
     pure (a+b)
   ```

   Unlike `IO`, evaluating a value in the `RandomState` monad needs a bit more work, since we need to provide an initial seed:

   ```
     >> runState roll_2d6 (mkSeed 5)
     (6,mkSeed 554244747)
   ```

2. Write an *IO action* `runRandomStateIO :: RandomState a → IO a` which performs the computation of a `RandomState` action, but obtains an initial seed from Haskell's built-in random number generator (i.e. using `randomRIO`).

   ```
     >>> runRandomStateIO roll_2d6
     8
     >>> runRandomStateIO roll_2d6
     3
   ```

3. Haskell's standard library has a function `sequence :: (Monad m) ⇒ [m a] → m [a]`. We can try it out on the expression `sequence [roll_2d6, roll_2d6, roll_2d6]`. What is the type of this expression? What does it compute? How can you obtain its result? (See Hint 1)

4. *(Optional)* Write a function `multiEval :: [RandomState a] → RandomState [a]` which takes a list of state monad actions, and runs all of them using *the same initial seed.* This will uncover a challenge: after running *n* actions on the same seed, we also have *n* (possibly different) updated seeds; try to think of a way to properly update the state using `putState`, but only if you have time left.

   Note that Haskell's `sequence` function has a similar type signature. What is the difference in behaviour between `multiEval` and `sequence`?

5. *(Optional)* There are now two different Monads that have actions for generating random numbers: there is `randomRIO :: (Int,Int) → IO Int` and `getRandomRange :: (Int,Int) → RandomSta`
   Define a class `MonadRandom` with a function `getRandomR`, and make instances of this class for both `IO` and `RandomState`.

**Exercise 12.5** (*Mandatory:* Separating 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[3]. 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 Int | Dice Int
          | 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 = Int → m Int
   evalM :: Expr → DiceAction IO → IO Int
   ```

   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 Int` 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 Int
   ```

   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 Int
   evalRIO expr = evalM expr (\k→randomRIO (1,k))
   ```

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

   ```
   evalIO :: Expr → IO Int
   ```

   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.

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

   ```
   evalND :: Expr → [Int]
   ```

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

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

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

```
expectation :: (Fractional a) ⇒ Expr → a
expectation e = avg (evalND e)
  where
  avg :: (Fractional a) ⇒ [Int] → 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 12.4, define:

   ```
   evalR :: Expr → RandomState Int
   ```

   which resolves dice rolls through a random number generator *without* using IO, but using the *state monad* of Exercise 12.4 instead. You can test it with `runRandomStateIO` function defined in that same exercise.

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 [Int]`.

   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`)*

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 12.6** (*Extra:* List and IO as Applicative, `AskNames.hs`). When using `Applicative`, it is possible that functions that behave quite differently, but have some deeper similarities, can be expressed in remarkably similar ways.

1. Write the following function using operations from `Applicative` such as `<*>`, `pure`, `<$>`.

   ```
   generateNames :: [String] → [String] → [String]
   generateNames firstnames surnames = [ f ++ " " ++ l | f ← firstnames, l ← surnames ]
   ```

   A sample output would be:

   ```
    ⋙ generateNames ["Harry", "Ron", "Hermione"] ["Potter", "Weasley", "Granger"]
   ["Harry Potter","Harry Weasley","Harry Granger","Ron Potter","Ron Weasley"
   ,"Ron Granger","Hermione Potter","Hermione Weasley","Hermione Granger"]
   ```

2. Write the following IO action using operations from `Applicative` such as `<*>`, `pure`, `<$>`.

   ```
   getFullname :: IO String
   getFullname = do
     first   ← putStrLn "First name?" >> getLine
     surname ← putStrLn "Last name?"  >> getLine
     return (first ++ " " ++ surname)
   ```

3. Given the following function definition:

   ```
   makeName :: (Applicative t) ⇒ t String → t String → t String
   makeName first surname = pure (\f l→f ++ " " ++ l) <*> first <*> surname
   ```

   Try to implement `generateNames` and `getFullname` in terms of `makeName`. That is:

   ```
   generateNames f l = makeName arg1 arg2

   getFullName = makeName arg3 arg4
   ```

**Exercise 12.7** (*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 3) 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 12.8** (*Extra:* Proofs using monad laws).
In various places (such as Exercise 11.6), 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 3) 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.** You can think of the function `sequence` on monads to be defined as follows:

```
sequence :: (Monad m) ⇒ [m a] → m [a]
sequence []     = pure []
sequence (m:ms) = do
  x ← m
  xs ← sequence ms
  pure (x:xs)
```

**Hints to practitioners 2.** When parsing this syntax:

```
newtype State a = St { runState :: 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 3.** 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.