11 Functors and Monads

Exercise 11.1 (*Warm-up:* Type instances, `FindDefs.hs`).
Give **non-trivial** function definitions that match the following types:

```
(?$)      :: Maybe (a → b) → Maybe a → Maybe b
pair      :: (Applicative f) ⇒ f a → f b → f (a,b)
apply     :: [a → b] → a → [b]
apply2nd  :: [a → b → c] → b → [a → c]
```

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.

For the function `pair`, note that it is *polymorphic* on the *kind* of Applicative. So it must work on arguments of type `Maybe a` and `Maybe b` to produce a an object of type `Maybe (a,b)`, but also on `[a]` and `[b]` to produce a list `[(a,b)]`, etc. The variable `t` gets substituted like any other type variable, except that instead of accepting complete types (like `Int` or `Maybe String`), it expects a *type constructor* like `Maybe` or `[]`.

But since you don't know what that type constructor will be, you have to rely on the `Applicative` class operations like `pure`, `⟨*⟩`, `<$>`, `liftA2`, etc.

Exercise 11.2 (*Warm up:* Functor recap, `FunctorRecap.hs`). We have already seen the `Functor` type class before in week 6. Do you still remember how it can be used? This exercise will give you some examples to practice with.

Consider the following container type, that holds two or three values:

```
data TwoOrThree a = Two a a | Three a a a
```

1. Give an instance of `Functor` for `TwoOrThree`.
2. What is the result of `fmap succ (Two 1 2)`?
3. If you have a list of `TwoOrThree`s, can you double the elements in each of them?

   ```
   doubleAll :: [TwoOrThree Int] → [TwoOrThree Int]
   ```

We can also make a slight variant of the container,

```
data TwoOrThree' a = Two' Int a | Three' Int a a
```

4. Give an instance of `Functor` for `TwoOrThree'`.
5. What is the result of `fmap succ (Two' 1 2)`? Why is this different than the result in part 2
6. Could `Int` be made an instance of `Functor`?

Exercise 11.3 (*Warm up:* Working with applicatives, `ApplicativeExpr.hs`).
Consider the following expressions:

```
("dr." ++) <$> Just "Sjaak"

pure (filter (\x→x>1)) <*> Just [1,2,3]

filter (>1) <$> Just [1,2,3]

mod <$> Just 7 <*> Just 5

replicate <$> [1,2,3] <*> ['a','b']
```

Predict what each of these expressions do. Check your answers using GHCi!
(Reminder: the function `replicate` has signature `Int → a → [a]`)

Exercise 11.4 (*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 11.5 (*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 11.6 (*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 1)*

Exercise 11.7 (*Mandatory:* Creating `Applicative` instances, `Result.hs`).
(This exercise is needed for the final part of Exercise 11.8, but you can do the first two parts of that exercise independently.)

The `Maybe` type is typically used in cases where it is not certain whether a computation will deliver a result—if it doesn't, `Nothing` can be returned. Examples are the expression evaluator of Exercise 4.6, or the standard function `lookup :: (Eq a) ⇒ a → [(a, b)] → Maybe b`. However just returning `Nothing` doesn't really tell us *why* a computation failed. So, we are going to introduce this variant on the `Maybe` type:

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

Here, the `Okay` constructor corresponds to the `Just` data constructor for `Maybe`, and `Error` corresponds to `Nothing`, except that we now have the ability to return (multiple) explicit error messages. Like `Maybe`, this type can be turned into an instance of `Functor` and `Applicative`.

1. Create the instance `Functor Result`. It should behave similar to the instance for `Maybe`: apply the given function to the value kept in `Okay`, and preserve error messages:

   ```
   >>> fmap reverse (Okay [1,2,3])
   Okay [3,2,1]
   >>> fmap reverse (Error ["list is empty", "not divisible by 5"])
   Error ["list is empty", "not divisible by 5"]
   ```

2. What is the type of `fmap` for the instance of `Functor` for `Result`?

3. Create an instance `Applicative Result`. The boilerplate for this starts with:

   ```
   instance Applicative Result where
     ...
   ```

   Complete this instance definition by defining the two minimally required functions, and specify what their types are.

   Note that the intent is that all error messages are preserved and combined. For example:

   ```
   >>> (*) <$> Okay 6 <*> Okay 7
   Okay 42
   >>> (++) <$> Okay [1,2,3] <*> Okay [4,5,6]
   Okay [1,2,3,4,5,6]
   >>> (++) <$> Okay [1,2,3] <*> Error ["invalid arguments"]
   Error ["invalid arguments"]
   >>> (*) <$> Error ["division by zero"] <*> Error ["not a number","unknown variable: x"]
   Error ["division by zero","not a number","unknown variable: x"]
   ```

Exercise 11.8 (*Mandatory*: Using applicative functors, `AST.hs`/`ASTInfix.hs` (your choice)).
In Exercise 4.6, we wrote an expression evaluator:

```
eval :: (Fractional a, Eq a) ⇒ Expr → a → Maybe a
```

for a data type `Expr` that could express addition, subtraction, multiplication and division, as well as integer constants and a *single* unknown variable x. So, `Expr` could represent a formula like "$2x + 1$". This data type could be implemented (your choice) using either prefix data constructors:

```
data Expr = Lit Integer | Var | Add Expr Expr | Mul Expr Expr | ...
```

or infix constructors:

```
data Expr = Lit Integer | Var | Expr :+: Expr | Expr :*: Expr | ...
```

To support multiple unknown variables (x, y, ...), we can extend this data type, replacing the `Var` constructor as follows:

```
type Identifier = String
data Expr = ... | Var Identifier | ...
```

We are going to modify `eval` so it supports this extension to `Expr`. You can use your solution to Exercise 4.6 as a starting point if you prefer, or use one of the two template versions.

1. Modify `eval` to support *multiple variables*, using the type:

   ```
   eval :: (Fractional a, Eq a) ⇒ Expr → [(Identifier,a)] → Maybe a
   ```

   The second argument to `eval` (which in Exercise 4.6 gave the value for x) is now an *association list* that associates variable names with values (we have seen *association lists* before, for instance when creating Huffman encodings in Exercise 7.6).

   For example (assuming prefix-constructors):

   ```
   let vars = [("x",5), ("y",37)]
   eval (Add (Var "x") (Var "y")) vars ⟹ Just 42.0
   eval (Add (Var "x") (Var "y")) [] ⟹ Nothing
   eval (Div (Var "z") (Lit 0)) vars ⟹ Nothing
   ```

2. Reduce the number of `case`-expressions needed in `eval` as much as possible by using the fact that `Maybe` is an instance of `Applicative`. So, rewrite it using the operations `⟨*⟩` and `<$>` and/or `pure`, as discussed in the lecture. Only one or two `case`-expressions should remain.

3. Replace `Maybe` with the `Result` type of Exercise 11.7:

   ```
   eval :: (Fractional a, Eq a) ⇒ Expr → [(Identifier,a)] → Result a
   ```

   So it can accurately report on all occurrences of these errors:

   - division by zero
   - variables without an associated value

   For example (assuming infix-constructors; the order of the errors does not matter):

   ```
   let vars = [("x",5), ("y",37)]
   eval (Var "x" :+: Var "y") vars ⟹ Okay 42.0
   eval (Var "x" :+: Var "y") [] ⟹ Error ["unknown variable: x","unknown variable: y"]
   eval (Var "z" :/: Lit 0) vars ⟹ Error ["division by zero", "unknown variable: z"]
   ```

   (If you used `Applicative` correctly in the previous step, this should not be a lot of work.)

Exercise 11.9 (*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 11.10 (*Extra*: Composing functors, `Compose.hs`).

Consider the type `[Maybe a]`. This has two layers of containers: it is a list of maybes of integers. You could also consider this combination of containers as a single new continer type,

```
data ListMaybe a = LM [Maybe a]
```

1. Give an instance of `Functor` for `ListMaybe`. Try to use the `Functor` instances of list and maybe.
2. Give an instance of `Applicative` for `ListMaybe`. Try to use the `Applicative` instances of list and maybe.
3. Give an instance of `Monad` for `ListMaybe`. Try to use the `Monad` instances of list and maybe as much as possible.
4. Test your implementation on the following cases

   ```
   fmap succ (LM [Nothing])
   fmap succ (LM [Just 1, Just 2, Nothing])
   (+) <$> LM [Nothing, Just 2, Just 3] <*> LM [Just 1, Nothing]
   liftM2 (+) (LM [Nothing, Just 2, Just 3]) (LM [Just 1, Nothing])
   ```

   The expected output is in the template file.

More generally, we can consider the composition of any two container types `f` and `g`

```
data Compose f g a = C (f (g a))
```

What we had before is a special case with `f=[]` and `g=Maybe`.

5. Give an instance of `Functor` for `Compose`.
6. Give an instance of `Applicative` for `Compose`.
7. Is it possible to give an instance of `Monad` for `Compose`? What problem do you run into?

Hints to practitioners 1. 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 (>>=).