11 Applicative functors

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 t) ⇒ t a → t b → t (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 `product`, 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:* Working with functors, `FMapExpr.hs`).
While the *name* 'functor' sounds very abstract and mathematical, it essentially just embodies an operation that you have been using since Exercise 1.5: namely, that of `map` to 'lift' an operation to a different type. Except that it is now called `fmap`. So, whenever you see `Functor`, think: 'the type class that allows you to use `fmap`'.

Now, consider the following expressions:

```
fmap (\x→x+1) [1,2,3]

fmap ("dr." ++) (Just "Sjaak")

fmap toLower "Marc Schoolderman"

fmap (fmap ("dr." ++)) [Nothing, Just "Marc", Just "Twan"]
```

For each of these expression:

- Describe what they compute.
- Determine the `Functor` instance used for each `fmap` occurrence.
- Determine the *type* of each `fmap` occurrence *(Note: this will be completely determined by your answer to the previous point.)*

Check your answers using GHCi! (See Hint 1 on checking your answer for the last two questions.)

Exercise 11.3 (*Warm-up*: Implementing your own functor, `TreeMap.hs`).
In Exercise 4.3, we introduced binary trees as follows:

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

and we said: "Like `[a]`, the type of a list, `Tree a` is a polymorphic type: it stores elements of type `a`. Thus, we can have a tree *of* strings, a tree *of* integers, a tree *of* lists of things, and even trees *of* trees ...". I.e., `[]` and `Tree` are of the same *kind*.

Of course, we have `map` on lists, and so it is not unreasonable to also want that operation for *binary search trees* as well (it was already defined for `Btree`, the type of *leaf trees*, in the lecture). We are going to make a new instance of `Functor` for this.

1. Create an instance of `Functor` for the kind of `Tree`. You can define `fmap` directly using the recursive design pattern for `Tree`. *(In particular, you do not have to define a function `mapTree` first!)* What is the type of `fmap` here?

   Remember the boiler plate for writing instances:

   ```
   instance Functor Tree where
     -- fmap :: ???
     fmap f Leaf          = ...
     fmap f (Node x lt rt) = ...
   ```

2. Test your `fmap` instance on some example trees. For example `fmap (+1) (fromAscList [1,2,3])` should produce the same tree as `fromAscList [2,3,4]`.

3. `fmap` applied to a *binary search tree* (as defined in Exercise 4.3) is not guaranteed to result in a binary search tree. Try to find a binary search tree `tree` and lambda-expression so that the result of `fmap (\x→...) tree` is no longer a binary search tree. What additional requirement should hold for a function `f` to make sure that `fmap f` *does* preserve the requirements for binary search trees? Discuss whether you think the `Functor` instance for `Tree` is a good idea.

Exercise 11.4 (*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 11.5 (*Mandatory:* Creating `Applicative` instances, `Result.hs`).
(This exercise is needed for the final part of Exercise 11.6, 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 4.7, 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 (in comments) 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 11.6 (*Mandatory*: Using applicative functors, `AST.hs`/`AST2.hs` (your choice)).
In Exercise 4.7, 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 4.7 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 4.7 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
   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 11.5:

   ```
   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
   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 11.7 (*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 f) ⇒ f String → f String → f 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 arg1 arg2
   ```

Hints to practitioners 1. As an example, in the first expression of Exercise 11.2:

```
fmap (\x→x+1) [1,2,3]
```

The `Functor` instance used is that for lists (`[]`), since the second argument to `fmap` is a list. And so, looking at the type of `fmap`:

```
(a → b) → f a → f b
```

and using `f = []`, we get:

```
(a → b) → [a] → [b]
```

(we can in fact also write `(a → b) → [] a → [] b`.)

We can check that this is correct by using a type annotation:

```
⋙ (fmap :: (a → b) → [a] → [b]) (\x→x+1) [1,2,3]
[2,3,4]
```

Of course, in this case `fmap` is simply the same as plain old `map`.