11 Applicative functors and monads

Exercise 11.1 (Warm-up: non-determinism and lists, `Evaluator.hs`). Recall the expression datatype shown in the lectures. Extend the type and its evaluator by non-deterministic choice.

```
data Expr
   = Lit Integer    -- a literal
   | Expr :+: Expr  -- addition
   | Expr :*: Expr  -- multiplication
   | Div Expr Expr  -- integer division
   | Expr :?: Expr  -- non-deterministic choice
```

The expression `e1 :?: e2` either evaluates to `e1` or to `e2`, non-deterministically. For example,

```
toss  ::  Expr
toss  =  Lit 0 :?: Lit 1
```

evaluates either to `0` or to `1`. To illustrate, here are some example evaluations involving `toss`.

```
 ⋙ evalN toss
[0,1]
 ⋙ evalN (toss :+: Lit 2 :*: toss)
[0,2,1,3]
 ⋙ evalN (toss :+: Lit 2 :*: (toss :+: Lit 2 :*: (toss :+: Lit 2 :*: toss)))
[0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15]
```

As you can see, the evaluator returns a list of all possible results.

```
evalN :: Expr → [Integer]
```

Haskell's list datatype is already an instance of `Functor`, `Applicative`, and `Monad`. So, all you have to do is to extend the interpreter in, say, applicative style by adding one equation for `:?:`.

Exercise 11.2 (Worked example: environments, `Evaluator.hs`).

1. Augment the expression datatype by variables.

   ```
   data Expr
      = Lit Integer    -- a literal
      | Expr :+: Expr  -- addition
      | Expr :*: Expr  -- multiplication
      | Div Expr Expr  -- integer division
      | Var String     -- a variable
   ```

   The values of variables are provided by an environment, a mapping from variable names to values. For simplicity, the environment is implemented by a list of name-value pairs, a so-called association list. (The standard prelude offers a function

   ```
   lookup :: (Eq key) ⇒ key → [(key, val)] → Maybe val
   ```

   for looking up a value by key.

   Defining a straightforward version of the evaluator for our extended expression language would get this environment as an additional parameter, i.e.

```
evaluate :: Expr → [(String, Integer)] → Integer
```

However, we want to reuse either the applicative or monadic version of the evaluator, and hence we have to find a suitable instance for the `f` in `Expr → f Integer`. This can be achieved by introducing the following newtype declaration:

```
newtype Environ a = EN { fromEN :: [(String, Integer)] →  a }
```

The corresponding type for the evaluator, say `evalR`, becomes

```
evalR :: Expr → Environ Integer
```

Some example applications of `evalR`,

```
 ⋙ (fromEN (evalR (Var "a" :+: Lit 1)) [("a", 4711), ("b", 0815)]
4712
 ⋙ (fromEN (evalR (Var "a" :*: Var "b")) [("a", 4711), ("b", 0815)]
3839465
 ⋙ (fromEN (evalR (Var "a" :*: Var "c")) [("a", 4711), ("b", 0815)]
0
```

The `fromEN` calls are necessary to get rid of the `EN` constructor. If a variable is not defined in the environment, it evaluates to `0`.

Like in the previous exercise, extend the original interpreter in applicative style by adding one equation, this time for `Var s`.
To get it working it is also necessary to provide appropriate instances of `Functor` and `Applicative` for type `Environ`.

2. Change the interpreter to support both lookup and non-determinism. The easiest way to achieve this is by introducing a newtype that incorporates both effects. A possible solution would be

   ```
   newtype EnvND a = EnN { fromEnN :: [(String, Integer)] →  [a] }
   ```

   Again, introduce appropriate instances for both `Functor` and `Applicative`. You also need to adjust the alternatives of the evaluator that have an effect. More specifically, copy the previous interpreter, name it `evalNR`, and adjust the alternatives for `Var` and `:?:`.

Exercise 11.3 (Programming, `Generate.hs`). For purposes of testing it is often useful to enumerate the elements of a datatype. If the datatype is finite and small, we may want to enumerate all of its elements. For other datatypes, we could choose to enumerate all elements of some specified size or up to some given size. Generating elements can be accomplished in a systematic manner using the applicative instance of Haskell's list datatype. For example,

```
bools  ::  [Bool]
bools  =  pure False ++ pure True

maybes :: [elem] → [Maybe elem]
maybes elems = pure Nothing  ++ (pure Just <*> elems)
```

Each definition is closely modelled after the corresponding datatype definition. Do you see how?

1. Apply the idea to generate cards, elements of type `Card`.

```
data Suit  =  Spades| Hearts | Diamonds | Clubs
data Rank  =  Faceless Integer | Jack | Queen | King
data Card  =  Card Rank Suit | Joker
```

2. Adapt the idea to recursive datatypes such as lists or trees.

```
data Tree elem = Empty | Node (Tree elem) elem (Tree elem)
```

   In case you need some inspiration, here are some example calls:

```
 ≫ lists bools 1
[[False],[True]]
 ≫ lists bools 2
[[False,False],[False,True],[True,False],[True,True]]
 ≫ trees (lists bools 2) 1
[  Node Empty [False,False] Empty,Node Empty [False,True] Empty,
   Node Empty [True,False] Empty,Node Empty [True,True] Empty]
 ≫ trees (lists bools 2) 2
[  Node Empty [False,False] (Node Empty [False,False] Empty),
   Node Empty [False,False] (Node Empty [False,True] Empty),
   Node Empty [False,False] (Node Empty [True,False] Empty),
   Node Empty [False,False] (Node Empty [True,True] Empty),
   ...
   Node (Node Empty [True,True] Empty) [True,False] Empty,
   Node (Node Empty [True,True] Empty) [True,True] Empty]
```

Exercise 11.4 (Lambda expressions, `LamdaEval.hs`).

In this exercise we again extend our expression language to turn it into a tiny programming language. The language allows you to introduce *anonymous functions* (i.e. λ-expressions), and to apply these to other expressions. This has led to the following datatypes.

```
data Expr
  =  Lit Integer      -- a literal
  |  Var String       -- a variable
  |  Bin Expr Op Expr -- binary operator
  |  Abs String Expr  -- a lambda expression
  |  Expr :@: Expr    -- an application
  deriving Show

data Op = Plus | Mult
  deriving Show

data Value = IntVal Integer | FunVal Env String Expr
  deriving Show

type Env = [(String,Value)]
```

Variable names are represented as strings. Variables can occur at two places: either as expressions on their own or as arguments of a λ-expression (represented as `Abs String Expr`). The values of variables are provided by an environment, implemented as a list of name-value pairs (type `Env`). The semantic function `eval` that assigns a meaning to expressions can now deliver two kinds of results: integer values and function values. Recall that a λ-expression is in *weak head normal form* and hence cannot be evaluated further. Consequently, evaluating `Abs` will immediately result in a function value. The `Env` component of `FunVal` is the environment in which the corresponding λ-expression was evaluated.

We can apply such a function value to an argument expression using `:@:`. To simplify evaluation we will use a call-by-value evaluation strategy, and hence the argument of a function will be reduced first before the function is applied.

You can use the following vanilla interpreter as starting point for the extensions/modifications you're supposed to implement.

```
applyIntOp :: Op → Value → Value → Value
applyIntOp op (IntVal v1) (IntVal v2) =
   case op of
      Plus → IntVal (v1 + v2)
      Mult → IntVal (v1 * v2)

eval0 :: Expr → Env → Value
eval0 (Lit i)        env = IntVal i
eval0 (Var v)        env = fromJust (lookup v env)
eval0 (Bin e1 op e2) env = applyIntOp op (eval0 e1 env) (eval0 e2 env)
eval0 (Abs v b)      env = FunVal env v b
eval0 (ef :@: ea)    env = let FunVal env var body = eval0 ef env
                               arg                 = eval0 ea env
                           in eval0 body ((var,arg):env)
```

The following example expression

$$12 + ((\lambda x \longrightarrow x \times 2)(4 + 2))$$

can be used to test your interpreter. Translating this example into our expression language gives:

```
myExpr = Bin (Lit 12) Plus ((Abs "x" (Bin (Var "x") Mult (Lit 2))) :@: Bin (Lit 4) Plus (Lit 2))
```

If you enter

```
⋙ eval0 myExpr []
```

you will receive the answer:

```
IntVal 24
```

1. Convert the evaluator to monadic style. You should hide the explicit environment in the same way as you did in the previous assignment.

2. Change the interpreter to support error handling. An error occurs if a variable is not present in the environment or if the expected value is not of the right type (i.e. one of the operands of a binary expression is a function value, or the first argument of an application is an integer value). `Maybe` seems handy when doing computations that may fail. The only problem

is that if a computation fails, you never know why your program failed, you only know that it failed! To remedy this, you may want to add some extra information when a computation fails, such as an error message. The `Either` type (defined in `Data.Either`) allows you to do this. It models a computation that might fail with an error message:

```
data Either e a = Left e | Right a
```

By convention `Right` is used as a 'correct' answer and `Left` is used to signal a 'incorrect' answer, or an error. Your implementation should give a clear error message in the event of an error.

Exercise 11.5 (The `RWS` monad, `RWS.hs`).

The State monad allows you to write functions that manipulate (in the background) some form of state that can be changed over time. Without using monads, the type of such a function would be something like

```
f :: ... → ST → (R, ST)
```

Here `R` is the main result of the computation, whereas `ST` incorporates a modifiable state used during the computation (see Hutton, page 168-172, for a detailed explanation.). For example, `ST` could hold the seed for a random number generator that changes each time a new random number is generated.

Reader and Writer monads can be thought of a specialized form of State with some constrains: you can only read from a reader and you can only write to a writer. Using these restricted versions of state you can be more specific about what you are trying to achieve. The reader monad (actually the reader applicative) was used in exercises 11.2 and 11.4 to pass on the lookup table. The writer monad appeared on the slides of the lectures to count the number of evaluation steps.

In this exercise we will look at the combination of these tree monads: the RWS Monad. This monad is captured by the following newtype:

```
newtype RWS r w s a = RWS { fromRWS :: r → s → (a, s, w) }
```

This type has four parameters, used for the reader, writer, state and result components of the computation, respectively. The partial application `RWS r w s` of this type has kind `* → *`, and is used to instantiate classes `Functor`, `Applicative` and `Monad`.

In order to put you on the way, we show you how the instance for `Functor` is defined. A general advice when defining instances for the other classes is to write down the types of the operations first. It can also be helpful to replace the newtype by its right-hand side while leaving out the dummy constructor `RWS`.

```
instance Functor (RWS r w s) where
-- fmap :: (a → b) → RWS r w s a → RWS r w s b
--      == (a → b) → (r → s → (a, s, w)) → (r → s → (b, s, w))
   fmap f (RWS rws) = RWS $ \r → \s → let (a,s',w) = rws r s in (f a, s', w)
```

The instances for `Applicative` and `Monad` cannot be (fully) polymorphic in `w`. This can, for instance, be seen if we try to define the function `pure` (from class `applicative`). This function requires us to create a value of type `w` out of nothing, which is obviously impossible. The common solution is to require that `w` inhabits class `Monoid` (by putting a `Monoid` context restriction on the instance type) which allows us to use `mempty` instead.

1. Define appropriate instances for both `Applicative` and `Monad`.

2. To access the `r` and `s` arguments of the monad and to update the `w` and `s` components we introduce the following operations:

```
ask :: (Monoid w) ⇒ RWS r w s r
ask = RWS $ \r → \s → (r, s, mempty)

get :: (Monoid w) ⇒ RWS r w s s
get = RWS $ \_ → \s → (s, s, mempty)

put :: (Monoid w) ⇒ s → RWS r w s ()
put s = RWS $ \_ → \_ → ((), s, mempty)

tell :: w → RWS r w s ()
tell w = RWS $ \_ → \s → ((), s, w)
```

   `ask` returns the current value of `r`, whereas `get` returns the current value of `s`. `put s` updates the state component with `s`, and `tell w` sets the writer component to `w`.

   Use these effect specific operations to handle variables correctly in the evaluator below. Moreover, extend the evaluator with a counter that counts how many literals appeared in the expression, and a writer that logs the names of all the variables. The counter should be implemented using the State component of `RWS`.

```
type Env   = [(String,Integer)]
type Log   = [String]
type Count = Int

evalRWS :: Expr → RWS Env Log Count Integer
evalRWS (Lit i)      = pure i
evalRWS (e1 :+: e2)  = pure (+)  <*> evalRWS e1 <*> evalRWS e2
evalRWS (e1 :*: e2)  = pure (*)  <*> evalRWS e1 <*> evalRWS e2
evalRWS (Div e1 e2)  = pure div  <*> evalRWS e1 <*> evalRWS e2
evalRWS (Var v)      = pure 0
```

Exercise 11.6 (Pencil and paper: applicative functor).

1. Recall the functor laws:

$$\texttt{fmap id} = \texttt{id} \tag{12}$$

$$\texttt{fmap (f} \circ \texttt{g)} = \texttt{fmap f} \circ \texttt{fmap g} \tag{13}$$

   Show that the applicative functor laws, repeated for reference below, imply the functor laws if we assume `fmap f m = pure f <*> m`.

$$\texttt{pure id <*> v} = \texttt{v} \tag{14}$$

$$\texttt{pure (.) <*> u <*> v <*> w} = \texttt{u <*> (v <*> w)} \tag{15}$$

$$\texttt{pure f <*> pure x} = \texttt{pure (f x)} \tag{16}$$

$$\texttt{u <*> pure x} = \texttt{pure (\f → f x) <*> u} \tag{17}$$

2. Show that the interchange law (17) is equivalent to the following property:

$$\texttt{pure f <*> u <*> pure x} = \texttt{pure (flip f) <*> pure x <*> u} \tag{18}$$

   where `flip` is defined `flip f x y = f y x`.

3. Recall the Monad laws:

$$\texttt{return x >>= f} = \texttt{f x} \tag{19}$$

$$\texttt{m >>= return} = \texttt{m} \tag{20}$$

$$\texttt{(m >>= f) >>= g} = \texttt{m >>= (\x → (f x >>= g))} \tag{21}$$

   Show that the derived operators `*>` are `>>` are equal if we assume
   `f <*> a = f >>= \\g → a >>= return.g`