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 (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`.

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

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

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

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

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. *Optional:* show that any expression built from `pure` and `<*>` can be transformed to a normal form in which a single pure function is applied to impure arguments:

$$\texttt{pure f} \texttt{ <*> } u_1 \texttt{ <*> } \ldots \texttt{ <*> } u_n$$