Laboratorul 12 — Side effects & Applicative

```haskell
import System.Random
```

Random Numbers

Consider the following definition encapsulating the type of a random number generator.

```haskell
newtype MyRandom a = MyRandom { runRandom :: StdGen -> (a,StdGen) }
```

The idea is that in order to generate a random number you need a seed, and after you've generated the number you need to update the seed to a new value. In a non-pure language the seed can be a global variable so the user doesn't need to deal with it explicitly. But in a pure language the seed needs to be passed in and out explicitly - and that's what the signature of random describes.

Exercise 1

Consider the basic random number generator defined by

```haskell
randomPositive :: MyRandom Int
randomPositive = (MyRandom next)
```

This generates a positive random integer uniformly distributed over the interger range. Test MyRandom by runing things like

```
Main> runRandom randomPositive (mkStdGen 4)
```

Note that in order to run such a number generator we have to provide a StdGen as seed

Exercise 2

(a) Make `MyRandom` an instance of `Functor`.

(b) Use the `Functor` instance to define a function

```haskell
randomBoundedInt :: Int -> MyRandom Int
randomBoundedInt = undefined
```

producing a generator for positive numbers smaller than the given argument. Hint: rely on randomPositive

(c) Use the `Functor` instance, `randomBoundedInt`, and functions from `Data.Char` to produce a generator for characters in the range 'a'..'z'

```haskell
randomLetter :: MyRandom Char
randomLetter = undefined
```

Exercise 3

(a) Make `MyRandom` an instance of `Applicative`.

(b) Define a generator for pairs of integers smaller than 10 and letters in the range `'a'..'z'`

```
random10LetterPair :: MyRandom (Int, Char)
random10LetterPair = undefined
```

(c) Construct a 2 (decimal) digit random number in three steps. Starting with zero we:

- add a random integer in the range `[0,9]`
- multiply it by 10
- add another random integer in the range `[0,9]`

Parsing

Exercise 4

(a) Extend the `Exp` definition in `Exp.hs` to include substraction, division, and other operations

(b) Modify the parser to recognize these new operations

(c) Modify the evaluator to recognize these new operations

Exercise 5

(a) Add identifiers to `Exp`

(b) Define a parser for identifiers starting with a letter and followed by alphanumeric characters and/or "_"

(c) Modify the evaluator to incorporate that. Note that you will need an environment. Can you use the applicative instance from the class for functions with the same source