State

Chapter 6 shows how we can arrive at an elegant abstraction in library design. We

- start with an imperative generator of random numbers (no exercises about this);
- we make it pure (the `RNG` state);
- we observe that it is tedious to use;
- we change the view of random number generators to transformers (`Rand`); and,
- we remark that the main functions manipulating `Rand` can just as well manipulate any state transformers, not only random number generators.

Finally, we arrive at the `State` abstraction.

All exercises are to be solved by extending the file `Exercises.scala`. It contains several modules inside, to avoid name clashes between different abstractions. Please hand in the completed file.

Exercise 1. Write a function that uses `RNG.nextInt` to generate a random integer between 0 and `Int.maxValue`. Make sure to handle the corner case when `nextInt` returns `Int.MinValue`, which does not have a non-negative counterpart.[1]

```
def nonNegativeInt(rng: RNG): (Int, RNG)
```

Exercise 2. Write a function to generate a `Double` between `0` and `1`, not including `1`. You can use `Int.MaxValue` to obtain the maximum positive integer value, and you can use `x.toDouble` to convert an `x` of type `Int` to a `Double`.[2]

```
def double(rng: RNG): (Double, RNG)
```

Exercise 3. Write functions to generate `(Int, Double)`-pairs and `(Double, Int)`-pairs, where the integers are non-negative. You should be able to reuse the functions written above. Add explicit return type annotations to both functions in your solution.[3]

```
def intDouble(rng: RNG)
def doubleInt(rng: RNG)
```

Exercise 4. Write a function to generate a list of random integers. Add explicit return type annotation to this function.[4]

```
def ints(size: Int)(rng: RNG)
```

After you have solved Exercise 4: Notice that all functions written so far have the same format: `f[A] : RNG =>(A, RNG)`, except that in the last case this type has been curried with one additional parameter. This motivates the generalization of the interface from now on to:

```
type Rand[A] = RNG =>(A, RNG)
```

See Section 6.4 in the text book for more details about the `Rand[A]` type. We will develop an API for this type, like we did for `Option`. The API will allow us computing with random values, without explicitly carrying the generator state around.

[1] Exercise 6.1 [Pilquist, Chiusano, Bjarnason, 2023]
[2] Exercise 6.2 [Pilquist, Chiusano, Bjarnason, 2023]
[3] Exercise 6.3 [Pilquist, Chiusano, Bjarnason, 2023]
[4] Exercise 6.4 [Pilquist, Chiusano, Bjarnason, 2023]

Find the `Rand` type in the file and understand the `unit` and `map` implementations included, before you proceed.

Exercise 5. Use `map` to reimplement `double` for `Rand`. See Exercise 2 above.

Observe how, for `Option`, we used the higher order API to avoid using pattern matching, and how here we use it to avoid being explicit about the state (and also to avoid decomposing, a.k.a. pattern matching, the results of random generators into value and new state).[5]

Exercise 6. Implement `map2` with the signature shown below. This function takes two generators, `ra` and `rb`, and creates a generator producing values created by calling a function `f` to combine the values produced by `ra` and `rb`.[6]

```
def map2[A, B, C](ra: Rand[A], rb: Rand[B])(f: (A, B) =>C): Rand[C]
```

Exercise 7. Implement `sequence` for combining a `List` of `Rand` transitions into a single transition. Recall that we have already implemented `sequence` for `Option`—reuse the experience.

```
def sequence[A](ras: List[Rand[A]]): Rand[List[A]]
```

Use `sequence` to reimplement the `ints` function you wrote before. For the latter, you can use the standard library function `List.fill(n)(x)`[7] to make a list with `x` repeated `n` times.[8]

Exercise 8. Implement `flatMap`, and then use it to implement `nonNegativeLessThan`.[9]

```
def flatMap[A,B](f: Rand[A])(g: A =>Rand[B]): Rand[B]
```

Note: for `Option`, we used `map` to compose a partial computation with a total computation. In here we used `map` to compose a random generator with a deterministic function. Similarly for `flatMap`. Function, `Option.flatMap` was used to compose two partial computations. On `Rand` the `flatMap` function is used to compose two random generators.

Exercise 9. Read Section 6.5 before solving this exercise. Observe that everything we have done so far can just as well be done for other states than `RNG`. Generalize the functions `unit`, `map`, `map2`, `flatMap`, and `sequence`. Note that this exercise is split into two parts of the .scala file (search for "Exercise 9" twice).[10]

Exercise 10. We now connect the `State` and `LazyLists`. Recall from basics of computer science that automata and traces are intimately related: each automaton generates a language of traces. In our implementation automata are implemented using `State` and `LazyLists` can be used to represent traces.

Implement a function `stateToLazyList` that given a state object and an initial state, produces a lazyList of values generated by this `State` object (this automaton).

Exercise 11. Use `stateToLazyList` to generate a lazy stream of random integer numbers. Finally, obtain a finite list of 10 random values from this stream.

Notice, how concise is the expression to obtain 10 random values from a generator using streams. This is because all our abstractions compose very well.

[5] Exercise 6.5 [Pilquist, Chiusano, Bjarnason, 2023]
[6] Exercise 6.6 [Pilquist, Chiusano, Bjarnason, 2023]
[7] https://www.scala-lang.org/api/2.13.3/scala/collection/immutable/List$.html#fill[A](n:Int)(elem:=%3EA):CC[A]
[8] Exercise 6.7 [Pilquist, Chiusano, Bjarnason, 2023]
[9] Exercise 6.8 [Pilquist, Chiusano, Bjarnason, 2023]
[10] Exercise 6.10 [Pilquist, Chiusano, Bjarnason, 2023]