

Prop: Design of a Property-Based Testing Library

The purpose is to practice API design on the property testing framework. The exercise may appear difficult. Note that this exercise is not about testing (considerably easier, done last week), but about designing and developing a testing framework. We write code to generate test cases and to state properties to be tested.

Also note that the experience is getting easier as you proceed with exercises. In the first two or so, you will likely experience quite some impedance, because you (likely) do not understand yet how this API is supposed to function. Like with any other library, once you learn its spirit, the later exercises become easier (and the early ones start to appear trivial). So do not give up too early!

We explain different source files as we need them. The file `State.scala` is the one developed in Chapter 6. We are not changing that file, just using it, this week.

Hand in `src/main/scala/adpro/testing/Gen.scala`. Handing in solutions other than your own is not permitted.

Exercise 1 [–]. We begin with simple warm up exercises, recalling the API of Chapter 6. `RNG` is the type of (R)andom (N)umber (G)enerators. Create a new Simple random number generator of type `RNG` and see it with 42.

Exercise 2 [–]. Get a pseudo random number (`x`) of the generator defined in the first exercise. Get the next random number and bind it to `y`.

Exercise 3 [–]. The book uses this simple generator to implement a series of various generators, including one for nonnegative integers (function `RNG.nonNegativeInt`) and for doubles (function `RNG.double`). Check briefly where they are in `State.scala`, and what are their types. It is slightly less important how they work.

The book wraps a random generator in a state pattern. The state pattern needs a transition function of type `RNG => (A, RNG)`. So the random number generator state will be the state of our automaton, and the automaton will generate outputs of type `A`. The function defines how to move from one state to another state producing an output. Define three automata over `RNG` as a state space, that produce integers, non-negative integers, and doubles respectively. The entries are marked in the exercise files.

After you have done the above, use the new generators (`s_random_int`, `s_nonNegativeInt`, `s_double`) to obtain the first random integer, the first non-negative integers, the first double. Use `rng1` as the initial seed for all three.

Exercise 4 [–]. Recall the function `state2stream` from an earlier homework. Use this function to create a stream of random double numbers, given a seed.

```
def randomDoubles (seed: RNG): Stream[Double]
```

Then use this function with the original `rng1` to create a `List` of 1000 random doubles. Then use some other random seed to create a different list of 1000 random doubles.

Exercise 5 [–]. Use `System.currentTimeMillis.toInt` (impure) to initialize your stream of numbers and obtain a different stream every time you ask for a stream of doubles. Use `randomDoubles` from the previous exercise.

This completes the warm-up exercises. Search for Exercise 6 in the Gen object.

Exercise 6. Implement a test case generator `Gen.choose`. It should generate integers in the range `start` to `stopExclusive`. Assume that `start` and `stopExclusive` are non-negative numbers.¹

```
def choose (start: Int, stopExclusive: Int): Gen[Int]
```

This is best solved in the Gen object in `Gen.scala`. The right place is already marked in the file.

Hint: Before solving the exercise study the type `Gen` in `Gen.scala`. Then, think how to convert a random integer to a random integer in a range. Then recall that we are already using generators that are wrapped in `State` and the state has a `map` function. The tests for this exercise will not pass until you have solved Exercise 9 (`flatMap`).

Exercise 7. Implement test case generators `unit` (always generates a constant value given to it in a parameter), `boolean` (generates randomly true, false), and `double` (generates random double numbers).²

Again suitable types have been prepared for you in `Gen.scala` (see in the Gen object).

Hints: (i) The `State` trait already had `unit` implemented. (ii) How do you convert a random integer number to a random Boolean? (iii) Recall from two weeks ago that we already implemented a random number generator for doubles.

Exercise 8. Implement a method `Gen[A].listOfN`, which given an integer number `n` returns a list of length `n` containing `A` elements, generated by `this` generator. The method type has been created for you in the Gen class in `Gen.scala`.³

Hint: The standard library has the following useful function (`List` companion object):

```
def fill[A] (n: Int) (elem: =>A): List[A]
```

It is of course possible to implement a solution without it, but the result is ugly (you need to replicate the behavior of `fill` inside `listOfN`). You can use it to create a list of generators. To turn the list of generators into a generator of lists, note that `State` has a method `sequence`, which allows to take a list of automata and execute their transitions as a sequence, feeding the output state of one as an input to the next. This can be used to execute a series of consecutive generations, passing the RNG state around.

Exercise 9. Implement `flatMap` for generators. Recall that `flatMap` allows to run another generator on the result of the present one (`this`). Note that in the type below the parameter `A` is implicitly bound, as this is meant to be a method of `Gen[A]`.⁴

```
def flatMap[B] (f: A =>Gen[B]): Gen[B]
```

Hint: Recall that `Gen` is essentially a wrapped `State` of special kind. We already have a method `flatMap` for states, which allows to chain execution of automata. The simplest (and probably the best) solution is to delegate to that method.

Exercise 10. Use `flatMap` to implement a more dynamic version of `listOfN`:

¹Exercise 8.4 [Chiusano, Bjarnason 2014]

²Exercise 8.5 with some changes [Chiusano, Bjarnason 2014]

³Second part of Exercise 8.5 [Chiusano, Bjarnason, 2014]

⁴Exercise 8.6 [Chiusano, Bjarnason 2014] first part

```
def listOfN (size: Gen[Int]): Gen[List[A]]
```

This version doesn't generate lists of a fixed size, but uses a generator of integers to pick the size first.⁵

Exercise 11. Implement `union`, for combining two generators of the same type into one, by pulling values from each generator with equal likelihood.⁶

```
def union[A] (g1: Gen[A], g2: Gen[A]): Gen[A]
```

Hint: We already have a generator that emulates tossing a coin (which one is it?). Use `flatMap`.

Exercise 12. Implement `Prop[A].&&` and `Prop[A].||` for composing `Prop` values. The former should succeed only if both composed properties (`this` and `that`) succeed; the latter should fail only if both composed properties fail.⁷

```
def &&(p: Prop): Prop
```

```
def ||(p: Prop): Prop
```

⁵Exercise 8.6 [Chiusano, Bjarnason 2014] second part

⁶Exercise 8.7 [Chiusano, Bjarnason 2014]

⁷Exercise 8.9 [Chiusano, Bjarnason 2014] first part