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 (done last week), but about designing and developing a testing framework.

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/Gen.scala`.

Exercise 1. Recall 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 seed 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 `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]
```

[1] Exercise 8.4 [Chiusano, Bjarnason 2014]

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 numbers).[2]

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, which can be wrapped here.

Exercise 8. Implement a method `Gen[A].listOfN` that given an integer number `n` returns a list of length `n` containing `A` elements, generated by `this` generator.[3]

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

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

It is 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 `fill` to create a list of generators. To turn the list of generators into a generator of lists, use the `State`'s `sequence` method. 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 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`:

```
def listOf (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. Recall that `Prop` is defined as:

---

[2] Exercise 8.5 with some changes [Chiusano, Bjarnason 2014]
[3] Second part of Exercise 8.5 [Chiusano, Bjarnason, 2014]
[4] Exercise 8.6 [Chiusano, Bjarnason 2014] first part
[5] Exercise 8.6 [Chiusano, Bjarnason 2014] second part
[6] Exercise 8.7 [Chiusano, Bjarnason 2014]

```
case class Prop (run: (TestCases, RNG) =>Result)
```

It is basically a (wrapped) function (`run`) that given some test cases and a random seed will produce a result. 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
```

Hint: You can run the combined test suites using their `run` methods. You can check whether a result is a failure by calling the `isFalsified` method on a `Result` value.

There are no automated tests for this exercise, but make sure your solution compiles.

Exercise 13. This exercise explores the concept of type classes, which is important not only for testing and generators, but is a general extension mechanism used broadly in functional programming.

Reimplement functions `listOfN[A]` and `ListOf[A]` as functions in the `Gen` object (not in the class). The main challenge is to devise a type for the functions that uses an instance of a type class `Gen[A]` (or the evidence that `A` is `Generatable`) to create instances of `A`. The body of the functions can actually just delegate to solutions of exercises 8 and 10. We would like the following calls to compile, if instances of `Gen[Double]` and `Gen[Int]` are available:

```
Gen.listOfN[Double] (5)
Gen.listOf[Double]
```

The provided tests (look in `GenSpec.scala` for inspiration) only check whether you got the type signature right. You can create functionality tests easily by identifying the tests for exercises 8 and 10 and adapting them to this new set up. The functionality tests are not provided as they would fail to compile until you solve the exercise (which would break the experience for the entire set).

[7]Exercise 8.9 [Chiusano, Bjarnason 2014] first part