Property Testing

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 (which should be considerably easier), but about developing a testing framework. We write code to generate test cases and to state properties of these test-cases.

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.

The file in `src/main/scala/adpro/Exercise1.scala` is provided for background. You can ignore it if you are comfortable with our random generation exercises, or just skim through it as a refresher.

Hand in `src/main/scala/adpro/testing/Gen.scala`.

Exercise 1. 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.

Exercise 2. 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 3. 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

[1] Exercise 8.4 [Chiusano, Bjarnason 2014]

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

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

input to the next. This can be used to execute a series of consecutive generations, passing the `RNG` state around.

Exercise 4. 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 5. Use `flatMap` to implement a more dynamic version of `listOfN`:

```
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 6. 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 7. 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
```

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