070: Developing A Framework for Property Testing

The purpose is to practice API design on the property testing framework. You can choose to work through all exercises in Chapter 8 [Chiusano, Bjarnason 2015] instead. This version is a bit slower, and gives you a number of hints, for smoother experience.

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. We will *use* the testing framework in later exercises and projects.

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 `Gen.scala`. This week we use SBT for the first time (a build management tool), so start studying the question files with the README at the top-level. Finally, the comment on top of `Gen.scala` is probably useful, too.

Exercise 1. (Exercise 8.4 in [Chiusano, Bjarnason 2015])

Implement a test case generator `Gen.choose`. It should generate integers in the range `start` to `stopExclusive`.

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

This is best done in the `Gen` object in `Gen.scala`. The right place is already marked in the file for you. Test the generator in the REPL by generating three integer numbers from the range.

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. (Exercise 8.5 in [Chiusano, Bjarnason 2015] with some changes)

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).

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

Hint: The `State` trait already had `unit` implemented. How do you convert a random integer number to a random Boolean? Recall from Exercise 1, that we already implemented a random number generator for doubles.

The file `Exercise2.scala` contains some test cases for this exercise. It also has all the right imports set up so that you can use it for REPL testing all later exercises (just load `scala -i Exercise2.scala`).

Exercise 3. (Second part of Exercise 8.5 in [Chiusano, Bjarnason 2015])

Also 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`.

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`). Then 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 4. (Exercise 8.6 [Chiusano, Bjarnason 2015] first part)

Implement `flatMap`. 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]`:

```
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.

Exercise 5. (Easy; Exercise 8.6 [Chiusano, Bjarnason 2015] second part)

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

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

This version does not generate a list of a fixed size, but uses a generator of integers to generate the size first.

Exercise 6. (Easy; Exercise 8.7 [Chiusano, Bjarnason 2015])

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

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

Hint: Recall that we already have a generator that emulates tossing a coin. Which one is it? Use `flatMap` with it.

Exercise 7. Implement weighted, a version of union that accepts a weight for each Gen and generates values from each Gen with probability proportional to its weight.

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

Hint: Recall that we already have a generator of random double numbers from range (0;1); See Exercise 3. First translate weights to probabilities. Then use our generator of doubles to simulate an unfair coin with `flatMap`.

Exercise 8. (Relatively simple; Exercise 8.9 [Chiusano, Bjarnason 2015] first part)

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.

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