Parser Combinators (Library Design)

This short set of exercises bears on building the Parsing API (Chapter 9)–without knowing the internal representation. We cannot actually run this code. But we can check whether formulations look reasonable, and whether they type check. We can achieve surprisingly much without spending a lot of time on implementing.

All exercises are to be solved by extending the file `Parsers.scala`—the file you should hand in.

Exercise 1. Write a *type declaration* for a parser `manyA` that recognizes zero or more consecutive `'a'` characters. Most parsers may fail, but not this one. It always succeeds with a number, possibly zero. For instance, for the input `"aa"` it succeeds with two, while for `"cadabra"` it succeeds with zero.

Make sure that both implementations type check (compile). For your convenience the test suite makes a few type checking checks, but no actual executions. Functional testing is delayed, until we can have parser implementations.

Exercise 2. Using `product`, *implement* the combinator `map2` and then use this together with `many` to *implement* `many1` (a parser that matches its argument 1 or more times). The type of `map2` is the same as in the other structures we have seen. The type of `many1` is provided below.[1]

```
def many1[A](p: Parser[A]): Parser[List[A]]
```

Exercise 3. Using `flatMap` write the parser that parses a single digit, and then as many occurrences of the character 'a' as was the value of the digit.

Your parser should be named `digitTimesA` and return the value of the digit parsed (thus one less the number of characters consumed). Examples:

> Parsing `0whatever` should result in `Right (0)`.
> Parsing `1awhatever` should result in `Right (1)`
> Parsing `3aaawhatever` should result in `Right (3)`
> Parsing `2aawhatever` should result in `Left (...)`

To parse the digits, you can make use of a new primitive, `regex`, which promotes a regular expression to a Parser. In Scala, a string `s` can be promoted to a `Regex` object (which has methods for matching) using the method call `s.r`, for instance, `"[a-zA-Z_][a-zA-Z0-9_]*".r`[2]

```
implicit def regex(r: Regex): Parser[String]
```

Exercise 4. Implement `product` and `map2` using `flatMap`.[3]

Exercise 5. Express `map` using `flatMap` and/or other combinators (`map` is not primitive if you have `flatMap`).[4]

[1] Exercise 9.1 [Chiusano, Bjarnason 2014]
[2] Exercise 9.6 [Chiusano, Bjarnason 2014]
[3] Exercise 9.7 [Chiusano, Bjarnason 2014]
[4] Exercise 9.8 [Chiusano, Bjarnason 2014]