Library Design for Parser Combinators

Today we train algebraic abstract design: note that all the exercises bear on building the API–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 (Exercises 1–5).

All exercises in the chapter make sense for the interested student, not just those listed below. It may take some gymnastics to get through all the exercises, because the text book changes the types and representations as-we-go in the chapter. This is best worked upon by reading the chapter sequentially. Treat these exercises as a micro-project, with a goal to both build a combinator library, and to build a working JSON parser using it.

All exercises are to be solved by extending the file `Parser.scala`—the only file you should hand in. Warning: the last exercise may be time consuming. It is fine not to hand it in, if you need to prioritize other deadlines. It is however a super useful and important programming experience to build a parser using parser combinators. You should do it, if you have never tried this before. (Alternatively, especially if you do not hand in, you can try to implement a JSON parser using Scala's standard parser combinators, or the parboiled2 library. The advantage is that you will learn one of the popular libraries, and you will be able to run the parser.)

Exercise 1. Write a type declaration for a parser `manyA` that recognizes zero or more `'a'` characters. For instances for `"aa"` the result should be `Right(2)`, for `""` and `"cadabra"` the result should be `Right(0)`.

Note that this week there is no tests, as we are using the type checker to test (so just continue to run the compiler after every solution). If you don't understand why we cannot write tests, try to write a test for the first exercise (and get it to run)—you will see that this is impossible because we would need to implement the `Parsers` trait first!

Exercise 2. Using `product`, implement the combinator `map2` and then use this to implement `many1` in terms of `many`.[1]

```
def map2[A,B,C](p: Parser[A], p2: Parser[B])(f: (A,B) =>C): Parser[C]
```

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

Make sure that both implementations type check (compile).

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.

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]
```

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

Exercise 4. Implement `product` and `map2` in terms of `flatMap`.[3]

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

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

Exercise 6. Implement a JSON parser using the combinators library (without being able to run it!).[5] Consult the book for the spec. It suffices to implement enough to parse the example in the beginning of Section 9.4.1. Place your implementation in a new object `JSON` in the bottom of the `Parsers.scala` (for the ease of grading, not for good practice...).

If you don't manage to complete the exercise, please include an example of a string that can be parsed by your parser (you think it can—because you cannot really run the parser).

[4] Exercise 9.8 [Chiusano, Bjarnason 2014]
[5] Exercise 9.9 [Chiusano, Bjarnason 2014]