Parsing Combinators (JSON case study)

This week we want to complete an implementation of a concrete instance of `Parsers` and use this to implement a parser for JSON. The work happens at three levels:

- The abstract interface (trait) of parsers that is independent of low-level design choices. Check `trait Parsers` including the associated tests for the interface implementations (`Parsers.Laws`). These tests, unlike in the book, are written using Scalatest. TODO: IS THIS RELEVANT
- A simple implementation of parsers conforming to the above interface without back-tracking control (no labeling, no committing). Check `object MyParsers` and the associated types `Parser`, `ParseError`, `Location`, and `Result`
- An example of concrete parser for Json files, implemented using the library. Check `trait JSON` for the abstract syntax ADT and `class JSONParser` for the parser.

This setup, unlike in the text book, has the advantage that we can support you with automatic tests for the entire exercise. Be attentive to the three levels of work, throughout the work to achieve the best understanding. Different learning happens at each of the levels.

Hint: If an exercise appears in `MyParsers` then you should use the underlying representations in the solutions (it must be a basic function, otherwise we would have placed it under `Parsers`). Whenever an exercise appears inside `trait Parsers`, then abstract operators should be enough (a derived function). Finally, since the Json parser is implemented purely against the abstract interface, no access to the underlying representation is possible in the final exercises. If you reflect about it actively while solving solution, you will understand how the parser combinator algebra design is independent from the parser representation and state.

Hand-in: `Parsers.scala`

Alternative: If you are tired of solving small curated exercises, and would like to have a bit more freedom, you can ignore this exercise set, and implement a simple Json parser using Parboiled2. If you choose this route, put the entire parser in a single Scala file and hand it in.

Exercise 1. Study the implementation of the top-level type `Location`, top-level type `Parser`, and the structure `MyParser` in `Parsers.scala`. Also try to understand how `run` works with this implementation (this is the same variant that we discussed in the lecture.)

Implement the basic combinator `succeed` that takes an arbitrary value as an argument and produces a parser that always returns successfully with this value, not consuming any input.

Note: This exercise is solved in `MyParsers` because it is specific to our representation of parsers.

Exercise 2. Observe that `flatMap` is already implemented in the file for this representation of parsers. Express `map` in terms of `flatMap` and/or other combinators (`map` is not primitive if you have `flatMap`).[1]

Note: This exercise is solved in the trait `Parsers`. Since it uses only other operators, it is entirely independent of the representation chosen for parsers.

Some tests for this exercise require that Exercise 4 is completed (`string`). Also you may recall that you have already solved this exercise last week, so you can move over the solution and check whether it passes tests. Last week we have only checked whether it type checks.

Exercise 3. Using `map` and `flatMap` to implement the combinator `map2` for parsers.

[1]Exercise 9.8 [Chiusano, Bjarnason 2014]

Note: This exercise is solved in the trait `Parsers`. Since it uses only other operators, it is entirely independent of the representation chosen for parsers.

Some tests for this exercise require that Exercise 4 is completed (`string`). Like above, you may move the solution from last week and check whether it passes tests.

Exercise 4. We move our attention to object `MyParsers` again, where we implement basic concrete operations that need to use the underlying representation.

Implement parser `string` parser that matches a given string constant. A variant of the implementation is already presented in section 9.6.2 in the book but it requires a small adjustment for the representation of `Parser` used in this exercise set.

Once we have this parser, we can write more interesting test cases that are needed to test other combinators below (for instance `or` and `many` from the following exercises).

Exercise 5. Implement the combinator `or` that takes two parsers as arguments and tries them sequentially. The second parser is only tried, if the first one failed. This combinator needs to understand the underlying representation so we solve it in the `MyParsers` object.

Exercise 6. Use `succeed` and `map2` to implement the combinator `many`:

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

This combinator continues to parse using `p` as long as it succeeds and puts the results in a list. The last (the rightmost) parsed element is the head of the list after parsing.

Note: This exercise is solved in the trait `Parsers`. Since it uses only other operators, it is entirely independent of the representation chosen for parsers.

Exercise 7. Use `many` and `map` to implement a parser `manyA` that recognizes zero or more consecutive `'a'` characters. For instance, for `"aa"` the result should be `Right(2)`, for `""` and `"cadabra"` the result should be `Right (0)`.

Note: This exercise is solved in the trait `Parsers`. Since it uses only other operators, it is entirely independent of the representation chosen for parsers. Recall that we solved it last week, so you can just move it over and test. Just be sure to understand the difference—between what we can test today, and what we did last week.

Exercise 8. Implement the `product` combinator that applies two parsers sequentially and returns both results as a pair.[2]

Note: This exercise is solved in the trait `Parsers`. Since it uses only other operators, it is entirely independent of the representation chosen for parsers. This can be done, for instance, using `map2` — check last weeks sheet again for a possible solution.

Exercise 9. The `many1` combinator matches arbitrary many occurrences, except for no occurrences at all (zero). Implement it using the combinators we already have:[3]

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

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

Note: This exercise is solved in the trait `Parsers`. Since it uses only other operators, it is entirely independent of the representation chosen for parsers. You can find a solution in the last week's set, which you can now test for functional correctness.

Exercise 10. Using `map2` and `succeed`, implement the combinator `listOfN`:[4]

```
def listOfN[A] (n: Int, p: Parser[A]): Parser[List[A]]
```

Note: This exercise is solved in the trait `Parsers`. Since it uses only other operators, it is entirely independent of the representation chosen for parsers.

Exercise 11. 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`.

Implement a new primitive, `regex`, which promotes a regular expression to a parser:[5]

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

This combinator needs to understand the underlying representation so we place it in the `MyParsers` object. (All the information how advance the character count is available only at the concrete level, and a regex needs to inform the parser how many characters it has consumed.)

Exercise 12. We will now start working on building a JSON parser using our primitives (See also Exercise 9.9 in the text book, and our lecture slides). We will implement all of these exercises in the bottom of the file, in the `JSON` object.

Our JSON parser depends only on the `Parsers` interface, so we should not need to access any underlying representations when implementing it.

The design has been changed slightly from the book website proposal, to facilitate testability, and to encourage modularity.

Implement:

- `QUOTED` – a `Parser[String]` that matches a quoted string literal, and returns the value of the string (without the syntactic quotes)
- `DOUBLE` – a `Parser[Double]` that matches a double number literal, and returns its numeric value
- `ws` – a `Parser[Unit]` that matches a non-empty sequence of white space characters

Note that for other tokens in our subset of JSON we do not need to implement explicit parsers. For fixed tokens, our library already allows writing string literals as parsers.

Exercise 13. After having implementing the tokens, we now implement the basic terminals of the grammar that construct the basic values in the abstract syntax ADT for `JSON`.

- `jnull` – matches the literal `null` and returns `JNull`
- `jbool` – matches literals `true` and `false` and returns `JBool`
- `jstring` – wraps the result of `QUOTED` in a `JString` value
- `jnumber` – wraps the result of `DOUBLE` in a `JNumber` value

Exercise 14. Finally, implement the non-terminal parsers for `JSON` values:

[4]Exercise 9.4 [Chiusano, Bjarnason 2014]
[5]Exercise 9.6 [Chiusano, Bjarnason 2014]

- `jarray` – parses an array literal: a comma-separated list of JSON values, surrounded by a pair of square brackets
- `field` – parses a JSON object field: a quoted field name, followed by a colon token, followed by a JSON value. It produces a field name–value pair, that will later be used to construct an object
- `jobject` – parses a JSON object: a comma-separated list of fields, surrounded by a pair of braces.
- `json` – parses an arbitrary JSON value

Note that due to mutual recursion, you will not be able to test this completely before you solve all four cases.

Exercise 15. Open a Scala console and attempt to parse a simple Json file (for instance the example on top of `ParsersSpec.scala`) by invoking the Json parser interactively.

You have not really learnt how to build a parser if you do not know how to run it!