Parsing Combinators

Part I.

We first do a few warm-up exercises on building the Parsing API (Chapter 9)–without knowing the internal representation. We cannot actually run this code. The method followed by the book precludes testing at this stage (we are prototyping without sufficiently many details implemented). 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.

More and more tests start to work once you are in the second part of the set (starting with Exercise 8).

Hand-in: `Exercises.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. Your file should be prepared so that it embeds a json file as a string, and runs it in a main method – parsing. It should also contain at least 5 property-based tests (all passing). It will be tested by invoking `scala-cli run .` and `scala-cli test .`. Submit the file to learnIT and write to Andrzej on Teams that he should pass you—separately from the automated grading process..

Exercise 1. Implement `product` and `map2` using `flatMap`. Notice that this seems to mean that `map2` can be derived when we have `flatMap`. This will become explicit next week.[1]

Answer the questions: (1) Why the type signatures of `map2` and `product` do not declare the type parameter `A`? (2) Why is `map2` takes it second argument by-name?

Exercise 2. Use `succeed` and `map2` to implement the combinator `many`. 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. The tests for this exercise will fail until you implement exercises 8 and 9. You can jump in there and implement it right away (but it is a very different exercise as it requires understanding the concrete representation of parsers). Also you would need to implement `map`, in Exercise 3.

Exercise 3. Express `map` using `flatMap` and/or other combinators .[2]

Reflect a bit about this: We were able to derive `map2` and `map` from `flatMap`. The `map` function is not primitive if you have `flatMap`. This will be discussed in the following chapter, so start building the picture.

The tests for this exercise will fail as long as those for Exercise 2 fail (at least `succeed` has to be implemented).

Exercise 4. Use `many` and `map` to implement a parser `manyA` that recognizes zero or more consecutive `'a'` characters and returns the number of matched characters. For instance, for `"aa"` the result should be `Right(2)`, for `""` and `"cadabra"` the result should be `Right (0)`.

Note: This and all the previous exercise are solved in the trait `Parsers`. This is because we can write them using basic parser operations, and we do not have to know the underlying representation.

[1] Exercise 9.7 [Pilquist, Chiusano, Bjarnason, 2022]
[2] Exercise 9.8 [Pilquist, Chiusano, Bjarnason, 2022]

Exercise 5. Implement `many1`, a parser that matches its argument 1 or more times. Reflect on: Why `many1` is an extension method?[3]

Exercise 6. Using `map2` and `succeed`, implement the combinator `listOfN`:[4]

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

The tests will have to wait for Exercise 8.

Exercise 7. 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`[5]

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

The tests for this exercise will fail until `regex` is implemented in Exercise 10—you can do this right away, but it requires switching the mode of thinking from high-level to low-level, as `regex` is a basic operator and requires a concrete representation.

Part II.

Now, we shall complete an implementation of a concrete instance of `Parsers` and use this to implement a parser for JSON. This work happens at two levels:

- An implementation of parsers conforming to the above interface without back-tracking control (we follow the slicing model with labels, and committing, See Chapter 9). Check `object MyParsers` and the associated types `ParseError`, `Location`, `Parser`, `Result`, and `Sliceable` types.
- An example of concrete parser for Json files, implemented using the library. Check `enum JSON` for the abstract syntax ADT and `class JSONParser` for the parser.

This setup, is just slightly different from the book, which helps us to support you with automatic tests for the entire exercise.

Be attentive to the the 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 `Sliceable` 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.

[3]Exercise 9.1 [Pilquist, Chiusano, Bjarnason, 2022]
[4]Exercise 9.4 [Pilquist, Chiusano, Bjarnason, 2022]
[5]Exercise 9.6 [Pilquist, Chiusano, Bjarnason, 2022]

Exercise 8. Study the implementation of types `ParserError`, `Location`, `ParserState`, `Parser`, `Result`, and the structure `Sliceable` in `Exercises.scala`. As a warmup, answer (for yourself) the following question: why do we have to make the `Parser` type contravariant here?

Also try to understand how `run` works with this implementation (this is the same variant that we discussed in the lecture.) We first need to complete `Sliceable` so that we have a concrete implementation of `Parsers`. Revisit the trait `Parsers` on top of the file, and identify the abstract (unimplemented) members. These are the elements `Sliceable` has to provide.

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

Exercise 9. 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 10. 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:[6]

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

This combinator needs to understand the underlying representation so we place it in the `Sliceable` 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.)

Part III.

Exercise 11. 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 promoting string literals to parsers.

Exercise 12. 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`

[6] Exercise 9.6 [Pilquist, Chiusano, Bjarnason, 2022]

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

- `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` that parses an arbitrary JSON value is already implemented in the template file

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

Exercise 14. Open a Scala console and attempt to parse a simple Json file (for instance the example on top of `Exercises.test.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!

Exercise 15. (1) Explain why is it possible to place the laws inside the abstract trait `Parsers` and (2) Explain what is the advantage of placing the laws in the abstract trait.