Background and Motivation

Parser Combinators are very powerful tools to help parse strings using higher order functions. In a simplified manner, parser combinators take some parser(s) as an input and produces a parser as an output.

The reason as to why parser combinators are so powerful is that we can chain multiple parsers together. For example, suppose `parseHello` is responsible for parsing "hello" in to the number 5 and `parseNumbers` is responsible for parsing all numbers into a collective sum. Then by feeding the the output of `parseHello` into `parseNumbers`, we can get the numeric sum of all occurrences of "hello" in the string. While this may seem rather trivial, the power of parser combinators allows us to parse very complicated patterns just by chaining parsers and their rules together.

Building off the idea of parsing for parsing to compile a programming language, we often utilize the idea of context free grammars. In context free grammars, some production rules can generate various branches with different sets of non-terminals and/or terminals. If each parser represents as a production rule, parser combinators will allow us to chain and handle all cases of each of the parsers such that they construct the entire semantics of the grammar.

Since parser combinators imply functions and inputs for other functions, programming languages that support first class functions should be able to implement parser combinators. As such, Scala, a functional object-oriendted language that supports first class functions, should be able to support parser combinators.

Key Ideas and Contributions

Moors et al. provides an implementation of parser combinators in Scala. Their work provides support to express extended Backus–Naur form grammars and parser combinators in Scala. This means their library can support EBNF operators like the Kleene star, Kleene cross, etc., as well as the regular parser combinator operators.

They showed that Scala is able to provide parser combinators. But more importantly, these parsers and combinators are supported through an object-oriented style. So all one requires to use the parsers is extending the various traits/classes in the provided library. In this way, any parser written will have the support of combinators that Moors et al. has implemented.

Pushing further, the library also contains added support for type specific parsing. By inheritance of the Parser superclass, the Scala parser library also supports regular expressions for parsing and added support for Java specific parsing. This work is now implemented as the Scala Standard Parser Combinator Library, which was once part of the Scala standard library.

Evaluation

For the following evaluation, a small example of using Scala parser combinators can be accessed here.

To evaluate the Scala Parser Combinators library, I decided to present a small demo on parsing comma delimited strings into matrices. Instead of building various parsing rules (like string $\rightarrow$ list), I utilize the provided functions, which highlights the hybrid functional object-oriented style of Scala. Furthermore, I utilize the `RegexParser` and its support for regular expression.

We begin with our input. A common way of providing a matrix as a string of values is often separating the values by commas and rows by carriage returns.

"1, 2, 3, 4
5, 6, 7, 8
9, 10, 11, 12"

The aim of parsing such a string is to provide a matrix data structure, or simply a `List[List[Int]]`. What we must consider is when the parser could go wrong. Any non-numeric character (besides the comma) would cause the parser to throw an error. When an error occurs, we must decide how to handle it. In the demo, we return an empty matrix if the parser encounters an error at any point.

```
def parse(s: String): List[List[Int]] = parseAll(matrix, s) match {
  case Success(res, _) => res
  case _ => List[List[Int]]()
}
```

Figure 1: Parse case matching results

With that in mind, we move onto the construction of each parser. In Scala, each parser will have a defined output type `Parser[T]`. This type will optionally become the input to another parser through combinators, so each parser must handle their corresponding input type. We define 3 different parsers that we will combine together.

First, we consider transforming a String into an integer for each instance of an integer in the entire string. Thus, we have a parser `cell = num ^^ {_.toInt}` that matches any digits in a string and casts it into an integer via mapping. Here `num` is a regular expression to capture any sequence of digits, thereby capturing the integer. The `RegexParser` provides support to parse via regular expression pattern matching instead of creating our creating our own numeric parser.

Second, we must consider a sequence of numbers delimited by a comma and transform it into a List. This will produce each individual row. So we can consider a parser that repeatedly tries to parse numbers using `num` and then parses a comma when one is encountered. Fortunately, the library provides support for such a common case. `repsep(p, s)` takes a parser `p` and a separator parser `s`. It repeatedly uses parser `p` until it fails, which it will then try parsing `s` once. Each value parsed in `p` will become an element in a List. Therefore, `row = repsep(cell, ",")` will produce a list/row of numbers.

Finally, we must consider each line in the string and parse it as a row in a matrix. We utilize the same function above to assist in obtaining a list of lists. So running `matrix = repsep(row, EOL)` will result in a list of list of integers, where `EOL` is the system's line separator character.

```
def cell: Parser[Int] = num ^^ {_.toInt}
def row: Parser[List[Int]] = repsep(cell, ",")
def matrix: Parser[List[List[Int]]] = repsep(row, EOL)
```

Figure 2: Combining parsers

With this, we can pass in any string with the correct format and it will parse and produce a matrix. As a visual test, we can see that the matrix is indeed printed from the output in Figure 3. The matrix indeed holds integer values and negative values are parsed as expected.

Roadblocks

When using the `RegexParser`, whitespaces are ignored. So it became problematic when using the carriage return to separate lines. The flag can be overwritten and turned on to enable whitespaces to be parsed as normal.

For this example, the definition of whitespaces had to be amended to only include spaces and tabs. If we don't ignore whitespaces, any other whitespace besides carriage return would cause the parser to throw and error. Since carriage return is of importance, we must redefine the whitespace definition in an override.

```
def main(args: Array[String]): Unit = {
  val s = "412, 615, -415" + EOL +
    "123, -84, 1" + EOL +
    "56, 78, 15"

  val m = parse(s)
  println(m)
  println(m(0)(2) < 0)
}
List(List(412, 615, -415), List(123, -84, 1), List(56, 78, 15))
true
```

Figure 3: Sample parsing runner and output

```
override protected val whiteSpace = """ \t""".r
```

Figure 4: Whitespace override

Thoughts

Overall, using the library feels seamless. The library seems pretty intuitive to use, but further reading on the various subclasses would be needed. For further exploration, it would be a nice challenge to write a parser for more complex files. The matrix parser can be adapted to parse CSV files as those are fairly similar in structure. Parsing token streams into ASTs with function combinators could be easier through usage of combinators to handle all cases.