Parser Combinators

Guide to Chapter 9 of Chiusano/Bjarnason

What do we learn from this chapter?

- How to use a parser combinator library?
- Specify a simple language (JSON) using a grammar and regexes
- Design an internal DSL for expressing grammars in scala
- Separating design from implementations

Key Concepts that appear in this chapter

Algebraic design, algebra (type, operators, and laws)

Full abstraction of a type

Type constructor

Higher-kinded type, higher-kinded polymorphism

Structure-preserving map (the structure preservation law),

Input data in JSON format

(this is an example in concrete syntax of JSON)

```
{
  "Company name" : "Microsoft",
  "Ticker" : "MSFT",

  "Active" : true,
  "Price"  : 30.66,
  "Shares outstanding" : 8.38e9,
  "Related companies" :
    [ "HPQ", "IBM", "YHOO", "DELL", "GOOG", ],
}
```

Abstract Syntax for JSON

(this is what we want to obtain from input)

```
trait JSON
case object JNull extends JSON
case class JNumber (get: Double) extends JSON
case class JString (get: String) extends JSON
case class JBool (get: Boolean) extends JSON
case class JArray (get: IndexedSeq[JSON])
  extends JSON
case class JObject (get: Map[String, JSON])
  extends JSON
```

The Example in JSON's Abstract Syntax

(no longer a string, but a structured Scala object)

```
JObject(Map(
  "Shares outstanding" -> JNumber(8.38E9),
  "Price" -> JNumber(30.66),
  "Company name" -> JString("Microsoft"),
  "Related companies" -> JArray(
     Vector(JString("HPQ"), JString("IBM"),
            JString("YHOO"), JString("DELL"),
            JString("GOOG"))),
  "Ticker" -> JString("MSFT"),
  "Active" -> JBool(true)))
```

Parsing Combinators: TERMINALS for JSON

(We build a parser combinator language in which we can specify the translation)

```
val QUOTED: Parser[String] =
  """"([^"]*)"""".r
    .map { _ dropRight 1 substring 1}

val DOUBLE: Parser[Double] =
  """(\+|-)?[0-9]+(\.[0-9]+(e[0-9]+)?)?""".r
    .map { _.toDouble }

val ws: Parser[Unit] =
  "[\t\n ]+".r map { _ => () }
```

Parsing Combinators: JSON start symbol

```
lazy val json : Parser[JSON] =
   (jstring | jobject | jarray |
    jnull | jnumber | jbool) *| ws.?
```

- **|** is choice, **?** means optional
- ***|** is sequencing & ignore the right component when building AST
  (' x *| y ' is syntactic sugar for ' (x ** y) map { _._1 } '
- Laziness allows recursive rules (like in EBNF)

Turn terminal into AST leaves

```
val jnull: Parser[JSON] =
  "null" |* succeed (JNull)

val jbool: Parser[JBool] =
  (ws.? |* "true" |* succeed (JBool(true ))) |
  (ws.? |* "false"|* succeed (JBool(false)))

val jstring: Parser[JString] =
  QUOTED  map { JString(_) }

val jnumber: Parser[JNumber] =
  DOUBLE  map { JNumber(_) }
```

Parse complex values

```
lazy val jarray: Parser[JArray] =
  ( ws.? |* "[" |* (ws.? |* json *| ",").*
    *| ws.? *| "]" *| ws.? )
     .map { l => JArray (l.toVector) }

lazy val field: Parser[(String, JSON)] =
ws.? |* QUOTED *| ws.? *| ":" *| ws.? ** json *| ","

lazy val jobject: Parser[JObject] =
  (ws.? |* "{" |* field.* *| ws.? *| "}" *| ws.?)
    .map { l => JObject (l.toMap) }
```

Parser Combinators

(as an approach to parsing)

- Good for ad hoc jobs, parsing when regexes do not suffice
- Very lightweight as a dependency, no change to build process
- More expressive than generator-based tools (Turing complete)
- In standard libraries of many modern languages
- Error reporting weaker during parsing (but fpinscala does a good job)
- Usually slower than generated parsers (and use more memory)
- Typically no support for debugging grammars

Let’s analyze one combinator Expression

```
QUOTED *| ":" ** json *| "," // parser producing a field

QUOTED : Parser[String] // a parser producing a String
but implicit def operators[A](p:Parser[A])=ParserOps[A](p)
so operators(QUOTED) :ParserOps[String]

":" : String
but implicit def string (s: String): Parser[String]
so string (":") : Parser[String]

then (ParserOps[A])*| : Parser[B] => Parser[A]
So operators(QUOTED).*|(string(":")) : Parser[String]
```

The decoupling pattern

`trait Parsers[Parser[+_]]`
Contains all the combinators as (static) functions
transforming or constructing parsers of type `Parser[A]`

Also contains trait `ParserOps` & implicit conversions from `Parser`
`ParserOps` has methods that allow us using combinators infix

Type constructor `Parser[+A]` is abstract.
To implement the language we need to both implement a concrete `Parser` type, and the `Parsers` trait.

Running the parser

- We need to implement a Parsers.run method

```
def run[A] (p: Parser[A]) (input: String): Either[ParseError,A]
```

- Then we call a parser as follows:
  `run ("abra" | "cadabra") ("abra")`
  or `("abra" | "cadabra") run "abra"`
  (if we add a ParserOps delegation)

```
("abra" | "cada") run "abra" == Right("abra")
("abra" | "cada") run "Xbra" == Left(ParseError(...))
```

Parsing Libraries in Programming Languages

Java	
Parser Generators	ANTLR, JavaCC, Rats!, APG, …
Parser Combinators	Parboiled, PetitParser
Scala	
Parser Generators	? (parboiled2)
Parser Combinators	Scala parser combinators (previously Scalalib), parboiled2 (technically also a generator), fastparse
JavaScript	
Parser Generators	ANTLR, Jison
Parser Combinators	Bennu, Parjs And Parsimmon
C#	
Parser Generators	ANTLR, APG
Parser Combinators	Pidgin, superpower, parseq
C++	
Parser Generators	ANTLR, APG, boost meta-parse (?) , boost spirit (?)
Parser Combinators	Cpp-peglib, pcomb, boost meta-parse, boost spirit, Parser-Combinators