Parser Combinators

Guide to Chapter 9 of Chiusano/Bjarnason

DOSPL

What do we learn from this mini-project?

- How does a parser work ?
- 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 implementationsin internal DSLs

Language Design

(and language use: a case study in parsing JSON)

Input data in JSON format

```
{
  "Company name" : "Microsoft Corporation",
  "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
```

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

Internal Domain Specific Languages

(Parser Combinators are one example)

- Parser Combinators are a language (loosely similar to EBNF)
- Slogan: internal DSL is syntactic sugar of host language
- No external tools, just pure Scala, no magic involved
- In the book we first define an abstract trait Parsers that contains the combinators: combinators are just typed functions
- In the last part we implement the trait (a bit on that later)
- Separating language design from language implementation
  (a nice decoupling, although testing is hard in initial design phases)

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 `QUOTED :ParserOps[String]`

```
”:” : String
```

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

so `”:” : Parser[String]`

then `(ParserOps[A])*| : Parser[B] => Parser[A]`

So `QUOTED *| ":" : Parser[String] // continue on the whiteboard`

What have we used to implement this DSL

- Polymorphic types (that check syntax of our programs), for instance:
  `(ParserOps[A]) *| : Parser[B] => Parser[A]`
- Function values: `type Parser[+A] = Location=>Result[A]`
- Implicits: `implicit def regex (r: Regex):Parser[String]`
- Calls to unary methods without period (infix ops are methods of ParserOps)
- `":" ** json` is really `":".**(json)`
  (which delegates to `Parsers.product(string (":"),json)`)
- Math symbols as names, eg: ?,|,*|,*|,*, etc
  (btw. Scala allows unicode identifiers, used in scalaz internal DSLs)
- Ability to drop parentheses on calls to nullary methods
  `ws.?` translates to `ws.?()` (which delegates to `Parsers.opt(ws)`)
- Used Scala’s parentheses (and other stuff) as elements of our DSL

Language Implementation

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(...))
```

Implementing **run**

```
type Parser[+A] = Location => Result[A]

def run[A] (p: Parser[A]) (input: String)
     : Either[ParseError,A] =
   p (Location(input,0)) match {
     case Success(a,n) => Right (a)
     case Failure(err,_) =>  Left (err)
   }
```

Implementing a concrete parser

(simplified slightly for presentation)

```
implicit def string(s: String): Parser[String] =
  loc =>
    if (loc.curr startsWith s)
      Success (s, s.size)
    else {
      val seen = loc.curr.substring (0,
         Math.min(loc.curr.size, s.size))
      Failure (s"expected '$s' but seen '$seen'")
    }
```

Implementing an operator/combinator

(slightly simplified for presentation, flatMap strikes back)

```
def flatMap[A,B] (p:Parser[A]) (f:A=>Parser[B])
    : Parser[B] =
  loc => p(loc) match {
    case Success(a,n) => f(a) (loc advanceBy n)
    case e@Failure(_,_) => e
  }
```

Implementing an operator/combinator

(slightly simplified for presentation)

```scala
def or[A] (s1: Parser[A], s2: => Parser[A])
     : Parser[A] =
  loc => s1 (loc) match {
    case Failure (e) => s2 (loc)
    case r => r
  }

def product[A,B] (p1: Parser[A],p2: =>Parser[B])
    :Parser[(A,B)] =
  flatMap (p1) (a => map (p2)(b => (a,b)))
```

Odds & Ends

Design Ideas / Tips

- Test combinators separately on very simple parsers
- Debugging the library using a JSON-sized parser is a nightmare
  (use scalatest/scalacheck – unit and property testing)
- Idea: Incorporate white space into tokens or combinators, to clean up the rules of all the ws.? calls (like when using a proper tokenizer), e.g.

  Then write: `QUOTED *| ":" ** json *| ","`

  Instead of:
  `ws.? |* QUOTED *| ws.? *| ":" *| ws.? ** json *| ",„`
- Our concern is language design more than detailed error reporting
  (if you want you can ignore error messages issues in the book)

Etc.

- We develop the library to learn language implementation (don’t skip the project)
- We use the library to parse an example piece of JSON to recall how to use a parser (and ensure that the implemetation is succesful).
- We test the library, because it is impossible to succeed otherwise and because we learn how to test language implementations.
- We will not use this parser later, so it does not have to be perfect when you are done, but make it parse at least the example from the book.