

Parser Combinators

Guide to Chapter 9 of Chiusano/Bjarnason
Advanced Programming

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

The yellow skills are more general.

BTW, none of the above is Scala-specific. Not even FP specific.

Key Concepts that appear in this chapter

- Algebraic design, algebra (type, operators, and laws)
- Full abstraction of a type
- Type constructor
- Higher-kind, higher-kinded polymorphism
- Structure-preserving map (the structure preservation law)
- Internal DSL, fluid interface

All of these are well hidden in the chapter (some not named explicitly), so make sure you identify them after class.

Agenda

1. Running Example: parsing JSON
2. Design patterns, concepts, and principles
3. Parsing JSON (review of the combinators)
4. Implementing a concrete parser (somewhat less important)
5. Parsing libraries in programming languages

Input data in JSON format

(this is an example in concrete syntax of JSON; basically a character string)

```
{  
  "Company name" : "Microsoft",  
  "Ticker" : "MSFT",  
  
  "Active" : true,  
  "Price" : 30.66,  
  "Shares outstanding" : 8.38e9,  
  "Related companies" :  
    [ "HPQ", "IBM", "YHOO", "DELL", "GOOG", ],  
}
```

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)))
```

Abstract Syntax for JSON

(the types of what we want to obtain from the input, using a parser)

```
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
```

Agenda

1. Running Example: parsing JSON
2. Design patterns, concepts, and principles
3. Parsing JSON (review of the combinators)
4. Implementing a concrete parser (somewhat less important)
5. Parsing libraries in programming languages

Algebraic Design

- **Algebraic design**: *design your **interface** first, along with associated **laws**. Use the types and laws to **refactor and evolve** the interface.*
- We are using types heavily, **designing the API with types**, compiling and trying expressions.
- Since **laws are properties**, they are **tests** (property tests). This is a form of test-driven development (TDD), or test-first development.

Algebraic Design, Full Abstraction, Higher Kinds

(the API/Interface first; separation of design & Implementation)

Demo: Check how I abstracted the solution to exercises in week 1, to test both empty solutions (your template) and my solutions (solved) with the same test-suite.

These types are **abstracted fully**;
We work without deciding how they are implemented.
We typecheck & compile!

```
trait Parsers[ParseError, Parser[+_]] {  
  def run[A] (p: Parser[A]) (input: String): Either[ParseError,A]  
  def char (c: Char): Parser[Char]  
  def string (c: String): Parser[String]  
  def or[A] (s1: Parser[A], s2: Parser[A]): Parser[A]  
  ...  
}
```

This is a higher kind (a type that is polymorphic in type constructors not int types!)

This is a type (variable)

This is a type constructor (variable).
This particular variable must be instantiated with a covariant type constructor.

Algebraic Design

(laws, aka tests)

```
forAll { (c: Char) => run (char(c)) (c.toString) == Right (c) }
```

```
forAll { (s: String) => run (string (s)) (s) == Right(s) }
```

```
forAll { (s1: String, s2: String) =>  
    val p = or (string(s1),string(s2))  
    run (p) (s1) shouldBe Right (s1)  
    run (p) (s2) shouldBe Right (s2) }
```

...

You can make such tests compile, before you have the implementation of parsers!

Map is **structure preserving**

Consider new API:

```
def many[A] (p: Parser[A]): Parser[List[A]]  
def map[A,B] (p: Parser[A]) (f: A=>B): Parser[B]
```

Example:

```
map (many (char ('a')) ( _.size) ← What does this parser produce?
```

Law [for map]:

```
map (p) (a => a) == p // for any parser p
```

This means that map is **structure preserving**

(it only changes values produced, so with identity there is no change at all).

Category theory (mathematics of structures) says: there exist interesting laws about types;
Laws that can be written knowing only structures, not implementations.

Agenda

1. Running Example: parsing JSON
2. Design patterns, concepts, and principles
3. Parsing JSON (review of the combinators)
4. Implementing a concrete parser (somewhat less important)
5. Parsing libraries in programming languages

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), unless implemented at compile time (parboiled!)
- 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
- Sometimes Internal DSLs are called fluid interfaces (although it seems that that term is a bit more narrow).

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]
```

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

Agenda

1. Running Example: parsing JSON
2. Design patterns, concepts, and principles
3. Parsing JSON (review of the combinators)
4. Implementing a concrete parser (somewhat less important)
5. Parsing libraries in programming languages

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)

```
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)))
```

Agenda

1. Running Example: parsing JSON
2. Design patterns, concepts, and principles
3. Parsing JSON (review of the combinators)
4. Implementing a concrete parser (somewhat less important)
5. Parsing libraries in programming languages

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

Conclusion

(what you need to get from this week)

- Algebraic design, algebra (type, operators, and laws)
- Full abstraction of a type
- Type constructor
- Higher-kind, higher-kinded polymorphism
- Structure-preserving map (the structure preservation law)
- Internal DSL, fluid interface
- ... and parser combinators 😊

This is the last chapter in the case study series. Hurray!