www.autoscout24.com

Scala School
Parser Combinators

16.02.2016 Matthew Lloyd

What's a Parser?

Takes in a String and a Grammar and Transforms the string to something else

```
parse(JsonGrammar, "{ name: 'Lemmy Kilmister' }") // JSONObject
parse(RomanNumeralGrammar, "VIII") // 8
parse(XMLGrammar, "<cars><car>...</car><car>...</car></cars>") //[XMLObject]
```

We are used to just seeing XML or JSON parsers

Scala gives us generic parsers

Think of them like super powerful Regular Expressions

What do Parsers look like in Scala?

In their most basic form you can match a string, and transform in to a value

```
def I: Parser[Int] = "I" ^^^ 1
def V: Parser[Int] = "V" ^^^ 5
def X: Parser[Int] = "X" ^^^ 10

parse(X, "X") match {
  case Success(n, next) => // n == 10
  case NoSuccess(err, next) =>
}
```

But I'm used to regular expressions, this seems like more work...

Well Regular Expressions can be treated as parsers in Scala!

```
def numStr: Parser[String] = "[0-9]+".r

parse(numStr, "3432") match {
  case Success(n, next) => n == "3432"
}
```

But that's still a string? How do I make it a number?

You can use the ^^ operator to transform it!

```
def numInt: Parser[Int] = "[0-9]+".r ^^ {
  numberStr => numberStr.toInt
}

parse(numInt, "3432") match {
  case Success(n, _) => n == 3432
}
```

So how would I say "Some thing" OR "Other thing" ?

Simple! Just use the | operator!

```
def numerals: Parser[Int] = I | V | X

parse(numerals, "V") match {
  case Success(n, _) => n == 5
}

parse(numerals, "I") match {
  case Success(n, _) => n == 1
}
```

Okay cool but that seems pretty limited?

Wait! You can join (compose) parsers together with ~

```
def VI1: Parser[Int ~ Int] = V ~ I
```

Okay cool but that seems pretty limited?

Wait! You can join (compose) parsers together with ~

```
def VI1: Parser[Int ~ Int] = V ~ I
```

Composed parsers results are joined together with the ~ type
Think of ~ as a tuple
Maybe this syntax makes it more understandable:

```
def VI2: Parser[~[Int, Int]] = V ~ I
```

What? No! That's not what I want! This is confusing!

The reality is we never use the ~ type

It's always transformed into other things using ^^

```
// "VI" -> 6
def VI: Parser[Int] = V ~ I ^^ {
  case vValue ~ iValue => vValue + iValue
}

// "XXX" -> 30
def XXX: Parser[Int] = X ~ X ~ X ^^ {
  case xValue1 ~ xValue2 ~ xValue3 => xValue1 + xValue2 + xValue3
}
```

But wait that seems silly? Can't I make it repeat?

Sure! By adding + or * on to the end of your parser

+ means 1 or more
* means 0 or more

```
def oneOrMoreXs: Parser[List[Int]] = X+
def zeroOrMoreXs: Parser[List[Int]] = X*
```

But I want a number not a list of numbers!

Well use ^^ to map the result!

```
def manyXs: Parser[Int] = (X+) ^^ { list => list.sum }
```

Be aware that sometimes you will need to place parentheses around parsers for it to make sense

But I only need 3 Xs, It should fail if there's more. Can I do that?

Yes! You can use the repN parser!

```
// "XXX" -> 30
def threeXs: Parser[Int] = repN(3, X) ^^ { list => list.sum }
```

It will match exactly 3 repetitions of the parser

But there's a bunch of junk I don't care about in the string. How do I ignore it?

So lets take the example of a JSON array,
where we don't care about the square brackets:

```
val dumbjArray: Parser[List[JsonValue]] = "[" ~ (jsonListEntry*) ~ "]" ^^ {
  case bracket1 ~ jsonList ~ bracket2 => jsonList
}
```

You use the squiggly arrow operators ~> and <~ !

```
val betterjArray: Parser[List[JsonValue]] = "[" ~> (jsonListEntry*) <~ "]"
```

But what if I have optional parts?

You can use the postfix ? operator!

```
def numeral: Parser[Int] = I | V | X
def str: Parser[String] = "I might add a numeral here:"

def group: Parser[(String, Option[Int])] = (str ~ (numeral?)) ^^ {
  case str ~ num => (str, num)
}
```

Adding ? after the parser means it is optional
The parser will not fail if it is not there
However it means the type is now Option[T]

How do I use them?

You'll need to add the Parser Combinator library to your build.sbt

It used to be part of the core language until it wasn't.

```
libraryDependencies ++= Seq(
  ws,
  specs2 % Test,
  "org.scala-lang.modules" %% "scala-parser-combinators" % "1.0.4"
)
```

How do I use them?

Create a class and have it extend RegexParsers

Inside of that class you now have access to all of the parser methods

```
class BasicRomanNumeralParser extends RegexParsers {

  def I: Parser[Int] = "I" ^^^ 1
  def V: Parser[Int] = "V" ^^^ 5
  def X: Parser[Int] = "X" ^^^ 10
  def L: Parser[Int] = "L" ^^^ 50
  def C: Parser[Int] = "C" ^^^ 100
  def D: Parser[Int] = "D" ^^^ 500
  def M: Parser[Int] = "M" ^^^ 1000

  def numerals: Parser[Int] = I | V | X | L | C | D | M

  def doParse(input: String): Option[Int] = parse(numerals, input) match {
    case Success(n) => Some(n)
    case NoSuccess(err, next) => None
  }

}
```

This is all very well and good, but what about real examples?

Okay then! Here's a (bad) JSON parser in 8 lines:

```
type JsonValue = Any
def jsonBool: Parser[Boolean] = ("true" ^^^ true) | ("false" ^^^ false)
def jsonNum: Parser[Int] = "[0-9]+" ^^ { _.toInt }
def jsonString: Parser[String] = "\"" ~> "[^\"]+".r <~ "\""
def jsonArray: Parser[List[JsonValue]] = "[" ~> ((jsonValue <~ ",")*) <~ "]"
def jsonObjEntry: Parser[(String, JsonValue)] = ("[^\"]+".r <~ ":") ~ jsonValue ^^
  { case key ~ value => (key, value) }
def jsonObj: Parser[Map[String, JsonValue]] = "{" ~> (jsonObjEntry*) <~ "}" ^^ { _.toMap }
def jsonValue: Parser[JsonValue] = jsonBool | jsonNum | jsonString | jsonArray | jsonObj
```