Parser combinators

Alexander Podkhalyuzin

October 26, 2012

Parser Combinators

Parser Combinators
- Arithmetic expressions
- Scala implementation
- Tilde method
- Rest of grammar
- How to use Parser
- Parse tree is unusable...
- So Complex example...
- Regex parser
- What’s else?

Implicit conversions and parameters

Questions?

Arithmetic expressions

Imagine the following language:

```
expr ::= term {”+” term |”-” term}
term ::= factor {”*” factor |”/” factor}
factor ::= floatingPointNumber |”(” expr ”)”
```

What you need to implement this arithmetic language?

Scala implementation

This is very similar to formal grammar

```
import scala.util.parsing.combinator._
class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep("+"~term | "-"~term)
  def term: Parser[Any] = factor~rep("*"~factor | "/"~factor)
  def factor: Parser[Any] = floatingPointNumber | "("~expr~")"
}
```

Scala implementation

Parser Combinators
- Arithmetic expressions
- Scala implementation
- Tilde method
- Rest of grammar
- How to use Parser
- Parse tree is unusable...
- So Complex example...
- Regex parser
- What's else?

Implicit conversions and parameters

Questions?

This is very similar to formal grammar

```
import scala.util.parsing.combinator._
class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep("+"~term | "—"~term)
  def term: Parser[Any] = factor~rep("*"~factor | "/"~factor)
  def factor: Parser[Any] = floatingPointNumber | "("~expr~")"
}
```

We can learn that very simple rules can turn formal grammar to working example in Scala.

Tilde method

Semantic is very simple, it just joins two parsers, and result is ~[A, B], which contains both parts of this two parts.
In such case, when you don’t need result from one of this two parts, you can use method $\sim>$or $<\sim$.

Rest of grammar

- | method have the same meaning
- * should be replaced by ‘rep‘ method invocation
- Optional part of the grammar should be replaced by ‘opt‘ method invocation

Parser Combinators
- Arithmetic expressions
- Scala implementation
- Tilde method
- Rest of grammar
- How to use Parser
- Parse tree is unusable...
- So Complex example...
- Regex parser
- What's else?

Implicit conversions and parameters

Questions?

How to use Parser

This is very simple, just use method ‘parseAll‘:

```
object ParseExpr extends Arith {
  def main(args: Array[String]) {
    println(”input : ” + args(0))
    println(parseAll(expr, args(0)))
  }
}
```

Parser Combinators
● Arithmetic expressions
● Scala implementation
● Tilde method
● Rest of grammar
● How to use Parser
● Parse tree is unusable...
● So Complex example...
● Regex parser
● What’s else?
Implicit conversions and parameters
Questions?

Parse tree is unusable...

Our parse tree contains only ~ class objects and strings. This is not useful data object, to do some action on it, for example evaluation.
To replace it by our own data, use method ‘^^‘.

Parse tree is unusable...

That's our Arith example:

```
import scala.util.parsing.combinator._
class Arith extends JavaTokenParsers {
  import Arith._
  def convertToBinary: PartialFunction[~[Expr, List[~[String, Expr↩
       ]]], Expr] = {
    case left ~ list =>
      if (list.isEmpty) left
      else list.foldLeft(left) {
        case (res, op ~ right) => BinaryExpression(res, op, right)
      }
  }
  def expr: Parser[Expr] = term~rep("+"~term | "-"~term) ^^ ↩
       convertToBinary
  def term: Parser[Expr] = factor~rep("*"~factor | "/"~factor) ^^ ↩
       convertToBinary
  def factor: Parser[Expr] = (floatingPointNumber ^^ {case s => ↩
       LiteralExpression(s.toDouble)}) | ("("~>expr<~literal(")"))
}
```

Parse tree is unusable...

```
object Arith {
  sealed trait Expr {
    def eval: Double
  }
  case class BinaryExpression(left: Expr, op: String, right: Expr) ↩
       extends Expr {
    def eval: Double = {
      op match {
        case "+" => left.eval + right.eval
        case "-" => left.eval - right.eval
        case "*" => left.eval * right.eval
        case "/" => left.eval / right.eval
      }
    }
  }
  case class LiteralExpression(number: Double) extends Expr {
    def eval: Double = number
  }
  case class ParenthesisedExpression(expr: Expr) extends Expr {
    def eval: Double = expr.eval
  }
}
```

So Complex example...

Let's observe example for parsing complex number.

Parser Combinators
● Arithmetic expressions
● Scala implementation
● Tilde method
● Rest of grammar
● How to use Parser
● Parse tree is unusable...
● So Complex example...
● Regex parser
● What's else?
Implicit conversions and parameters
Questions?

Regex parser

In complex example we saw new regex parser. It takes any matched string, so it's quite simple:

```
object MyParsers extends RegexParsers {
  val ident: Parser[String] = """[a-zA-Z_]\w*""".r
}
```

What's else?

Obviously Scala standard parser combinators are not the best among all libraries, you can try something different, for example ‘parboiled‘, it looks very similar:

What's else?

```
class SimpleCalculator extends Parser {
  def Expression: Rule1[Int] = rule {
    Term ~ zeroOrMore(
      "+" ~ Term ~~> ((a:Int, b) => a + b)
    | "-" ~ Term ~~> ((a:Int, b) => a - b)
    )
  }
  def Term = rule {
    Factor ~ zeroOrMore(
      "*" ~ Factor ~~> ((a:Int, b) => a * b)
    | "/" ~ Factor ~~> ((a:Int, b) => a / b)
    )
  }
  def Factor = rule { Number | Parens }
  def Parens = rule { "(" ~ Expression ~ ")" }
  def Number = rule { oneOrMore("0" - "9") ~> (_.toInt) }
}
```

Implicit conversions and parameters

Why we need them?

Actually this is main Scala feature. This is the main thing for building powerful and flexible DSLs.

- Use implicit conversions to construct something called extension method: ”[a-z]”.r
- Custom static type cast
- Use implicit parameters to make method usage simpler for defined default things, and flexible to make custom implementation: method sorted and implicit parameter Ordering[T]
- Use combination of implicit conversion and implicit parameters to build complex DSLs like parser combinators

Extension methods

Old style way is to create RichObject:

```
class RichString(s: String) {
  def isEmail: Boolean = {...}
}
implicit def str2richStr(s: String): RichString = new RichString(s)

if (”string”.isEmail) {...}
```

Extension methods

Scala 2.10 introduced ”implicit classes”. This is more proper and simple syntax construction to add extension methods in your code:

```
implicit class RichString(s: String) {
  def isEmail: Boolean = {...}
}

if ("string".isEmail) {...}
```

There are some requirements to implicit classes, for example you can’t declare top-level class implicit, because it’s as usual only syntax sugar, compiler generates old style implicits for you.

Runtime object...

What about created object on runtime, just to get working extension methods. It’s performance problem, and all languages, which have extnsion method feature don’t have such problem.
Use value classes introduced in Scala 2.10:

```
implicit class RichString(val s: String) extends AnyVal {
  def isEmail: Boolean = {...}
}

if ("string".isEmail) {...}
```

So now it’s completely the same like extension, but still you can use hierarchies for your implicit types.

Parser Combinators

Implicit conversions and parameters

- Why we need them?
- Extension methods
- Runtime object...
- Implicit type cast

Questions?

Implicit type cast

Why we need it?

```
val i: BigInt = BigInt(1)

val button = new JButton
button.addActionListener(
  new ActionListener {
    def actionPerformed(event: ActionEvent) {
      println("pressed!")
    }
  }
)
```

Implicit type cast

But it can be much simpler:

```
val i: BigInt = 1

val button = new JButton
button.addActionListener(
  (_: ActionEvent) => println("pressed!")
)
```

It’s much more readable.

Implicit resolution rules

- There should be only one implicit conversion in scope. In case of ambiguity most specific alternative will be chosen
- For conversion compiler searches among implicit functions, function values, function objects
- No chained conversion
- You can choose any name for implicits, however the same shadowing rules can be applied for implicit search too

Object scope rules

It's not really useful, in case if we need to know what to import, if we want to convert Int to BigInt. So Scala have rules for implicit scopes.
For type From $\Longrightarrow$ To compiler collects all object related to this type. Then all implicit conversion from such objects used in implicit search resolution.

Questions?