Domain Specific Languages
Part 2: Parser Combinators

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola

Domain Specific Languages (DSLs)
Parser Combinators: Introduction

A parser combinator is

- a high-order function accepting several parsers as input and returning a new parser;
- a parser is a function accepting strings as input and returning some structure, e.g., a parse tree.

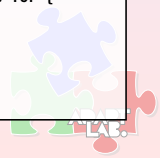
Parser combinators enable a recursive descent parsing strategy.

The basic idea

- parser combinators are building blocks for parsers that can be combined together
- a combinator framework eases to combine parsers to deal with sequential and alternative cases, repetition, optional terms, etc ....

Case study: the paycheck program, e.g.,

```
paycheck for employee "Buck Trends" is salary for 2 weeks minus deductions for {
   federal income tax            is   25. percent  of gross,
   state income tax              is    5. percent  of gross,
   insurance premiums            are 500. in gross currency,
   retirement fund contributions are  10. percent of gross
}
```



Domain Specific Languages (DSLs)
Case Study: the DSL Grammar.

```
paycheck       =   empl•gross•deduct
empl           =   paycheck•for•employee•employeeName
gross          =   is•salary•for•duration
deduct         =   minus•deductions•for•{•deductItems•}
employeeName   =   "•name•_•name•"
name           =    ...
duration       =   decimalNumber•weeksDays
weeksDays      =   week | weeks | day | days
deductItems    =   deductItem {•,•deductItem } | ε
deductItem     =   deductKind•deductAmount
deductKind     =   tax | insurance | retirement
tax            =   fedState•income•tax
fedState       =   federal | state
insurance      =   insurance•premiums
retirement     =   retirement•fund•contributions
deductAmount   =   percentage | amount
percentage     =   toBe•doubleNumber•percent•of•gross
amount         =   toBe•doubleNumber•in•gross•currency
toBe           =   is | are
decimalNumber  =    ...
doubleNumber   =    ...
```

nonterminals | terminals | alternatives | sequences | repetitions

Domain Specific Languages (DSLs)
Payroll DSL: A First Parser Combinator Version

```
package payroll.pcdsl

import scala.util.parsing.combinator._
import payroll._
import payroll.Type2Money._

class PayrollParserCombinatorsV1 extends JavaTokenParsers {
  def paycheck = empl ~ gross ~ deduct
  def empl = "paycheck" ~> "for" ~> "employee" ~> employeeName
  def gross = "is" ~> "salary" ~> "for" ~> duration
  def deduct = "minus" ~> "deductions" ~> "for" ~> "{" ~> deductItems <~ "}"
  def employeeName = stringLiteral              // stringLiteral from JavaTokenParsers
  def duration = decimalNumber ~ weeksDays      // decimalNumber from JavaTokenParsers
  def weeksDays = "weeks" | "week" | "days" | "day"
  def deductItems = repsep(deductItem, ",")
  def deductItem = deductKind ~> deductAmount
  def deductKind = tax | insurance | retirement
  def tax = fedState <~ "income" <~ "tax"
  def fedState = "federal" | "state"
  def insurance = "insurance" ~> "premiums"
  def retirement = "retirement" ~> "fund" ~> "contributions"
  def deductAmount = percentage | amount
  def percentage = toBe ~> doubleNumber <~ "percent" <~ "of" <~ "gross"
  def amount = toBe ~> doubleNumber <~ "in" <~ "gross" <~ "currency"
  def toBe = "is" | "are"
  def doubleNumber = floatingPointNumber   // floatingPointNumber from JavaTokenParsers
}
```

Domain Specific Languages (DSLs)
Some Combinators

Sequential Composition

- ~ is used when the results produced by the productions on the left and right of the ~ should be retained for further processing

```
def paycheck = empl ~ gross ~ deduct
```

- ~> is used when the result for the productions to the left is no longer needed

```
def empl = "paycheck" ~> "for" ~> "employee" ~> employeeName
```

- <~ is used when the result for the productions to the right is no longer needed

```
def tax = fedState <~ "income" <~ "tax"
```

Alternative Composition

- | expresses when two parsers are in alternative

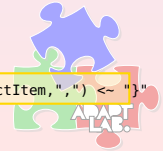
```
def weeksDays = "weeks" | "week" | "days" | "day"
```

Repetitive Composition

- rep/repsep match zero or more repetitions

```
def deduct = "minus" ~> "deductions" ~> "for" ~> "{" ~> repsep(deductItem,",") <~ "}"
```

There is an opt method for optional terms not used.

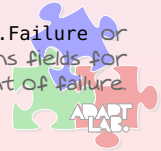


Domain Specific Languages (DSLs)
Parsing

To use the defined parser

```
val p = new PayrollParserCombinatorsV1
p.parseAll(p.paycheck, input) match {
   case p.Success(r,_) => ...
   case x => ...
}
```

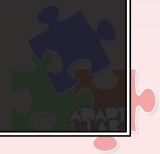
- parseAll is defined in a parent class it receives a parser (an invocation to paycheck in our case) and the input string to parse;
- if the parsing process is successful the result is an instance of p.Success[+T] a case class declared in the Parsers trait;
- the p prefix indicates that p.Success is a path-dependent type and permits to distinguish the result from two different parsers;
- the Success instance has two fields, the first is the result of the parse (of type T), the second is the remaining input to parse (normally empty);
- if the parse fails, the return instance is either a p.Failure or p.Error; both are derived from p.NoSuccess and contains fields for an error message and the unconsumed input at the point of failure.



Domain Specific Languages (DSLs)
Parsing (Cont'd)

Do you know which types have the parsers and the result?

```
scala> import scala.util.parsing.combinator._
scala> import payroll.pcdsl._

scala> val p = new PayrollParserCombinatorsV1

scala> p.empl
res0: p.Parser[String] = Parser (~>)
scala> p.weeksDays
res2: p.Parser[String] = Parser (|)
scala> p.paycheck
res3: p.Parser[p.~[p.~[String,p.~[String,String]],List[String]]] = Parser (~)

scala> p.parseAll(p.weeksDays, "weeks")
res4: p.ParseResult[String] = [1.6] parsed: weeks

scala> val input = """paycheck for employee "Buck Trends"
     | is salary for 2 weeks minus deductions for {}"""
input: java.lang.String =
paycheck for employee "Buck Trends" is salary for 2 weeks minus deductions for {}
scala> p.parseAll(p.paycheck, input)
res5: p.ParseResult[p.~[p.~[String,p.~[String,String]],List[String]]] =
              [2.46] parsed: ((("Buck Trends"~(2~weeks))~List())

scala>  val input = """paycheck for employe "Buck Trends"
     | is salary for 2 weeks minus deductions for {}"""
input: java.lang.String =
paycheck for employe "Buck Trends" is salary for 2 weeks minus deductions for {}
scala> p.parseAll(p.paycheck, input)
res6: p.ParseResult[p.~[p.~[String,p.~[String,String]],List[String]]] =
[1.14] failure: 'employee' expected but ' ' found
paycheck for employe "Buck Trends"
             ^
```



Domain Specific Languages (DSLs)
Giving a Semantics to the DSL

As we parse the DSL

- we had to look up the employee by name
- fetch his gross salary for the specified period and
- calculate the deductions

Once the parser finishes

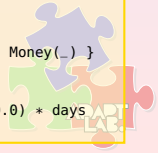
- we need to return a pair with the Employee instance and the completed Paycheck.

Domain Specific Languages (DSLs)
Giving a Semantics to the DSL


```
package payroll.pcdsl

import scala.util.parsing.combinator._
import payroll._
import payroll.Type2Money._

class UnknownEmployee(name: Name) extends RuntimeException(name.toString)

class PayrollParserCombinators(val employees: Map[Name,Employee]) extends JavaTokenParsers {
   var currentEmployee: Employee = null
   var grossAmount: Money = Money(0)

   /** @return Parser[(Employee, Paycheck)] */
   def paycheck = empl ~ gross ~ deduct ^^ {case e ~ g ~ d => (e, Paycheck(g, g-d, d))}

   /** @return Parser[Employee] */
   def empl = "paycheck" ~> "for" ~> "employee" ~> employeeName ^^ { name =>
     val names = name.substring(1, name.length-1).split(" ")
     val n = Name(names(0), names(1));
     if (! employees.contains(n)) throw new UnknownEmployee(n)
     currentEmployee = employees(n); currentEmployee
   }

   /** @return Parser[Money] */
   def gross = "is" ~> "salary" ~> "for" ~> duration ^^ {
     dur => grossAmount = salaryForDays(dur); grossAmount
   }

   def deduct = "minus" ~> "deductions" ~> "for" ~> "{" ~> deductItems <~ "}"
```

Domain Specific Languages (DSLs)
Giving a Semantics to the DSL (Cont'd)


```
   /** "stringLiteral" provided by JavaTokenParsers   *
   * @return Parser[String]                           */
   def employeeName = stringLiteral

   /** * "decimalNumber" provided by JavaTokenParsers *
   * @return Parser[Int]                              */
   def duration = decimalNumber ~ weeksDays ^^ {
     case n ~ factor => n.toInt * factor
   }

   def weeksDays = weeks | days
   def weeks = "weeks?".r ^^ { _ => 5 }
   def days = "days?".r ^^ { _ => 1 }

   /** @return Parser[Money] */
   def deductItems = repsep(deductItem,",")^^{items => items.foldLeft(Money(0)){_ + _}}

   def deductItem = deductKind ~> deductAmount
   def deductKind = tax | insurance | retirement
   def tax = fedState <~ "income" <~ "tax"
   def fedState = "federal" | "state"
   def insurance = "insurance" ~> "premiums"
   def retirement = "retirement" ~> "fund" ~> "contributions"
   def deductAmount = percentage | amount

   def percentage = toBe ~> doubleNumber <~ "percent" <~ "of" <~ "gross" ^^ {
     percentage => grossAmount * (percentage / 100.)
   }

   def amount = toBe ~> doubleNumber <~ "in" <~ "gross" <~ "currency" ^^ { Money(_) }
   def toBe = "is" | "are"
   def doubleNumber = floatingPointNumber ^^ { _.toDouble }
   def salaryForDays(days: Int) = (currentEmployee.annualGrossSalary / 260.0) * days
}
```



Domain Specific Languages (DSLs)
Giving a Semantics to the DSL (Cont'd)

Notes on the DSL

- The parser uses a map (`Name`) of known employees for simplicity;
- `currentEmployee` and `grossAmount` respectively store the employee the parser is processing and they gross salary for the pay periods;
- this parser version is an evolution of the previous one which take in consideration what should be the final result, e.g.,

```
def paycheck = empl ~ gross ~ deduct ^^ {case e~g~d => (e, Paycheck(g, g-d, d))}
```

  will return a `Pair` with the `Employee` and the computed `Paycheck`.

- `^^` combinator, `p1^^f1` applies `f1` to the result of `p1` when it succeeds

```
def empl = "paycheck" ~> "for" ~> "employee" ~> employeeName ^^ {
  name =>
    val names = name.substring(1, name.length-1).split(" ")
    val n = Name(names(0), names(1));
    if (! employees.contains(n)) throw new UnknownEmployee(n)
    currentEmployee = employees(n); currentEmployee
}
```

- `weeks` and `days` ignore the parsed string; they just return a multiplication factor used to determine the total days in the `duration` production rule

Domain Specific Languages (DSLs)
Giving a Semantics to the DSL (Cont'd)


```
import payroll._
import payroll.Type2Money._
import payroll.pcdsl._

object PayRollBuilder {
  def main(args: Array[String]) = {
    val buck = Employee(Name("Buck", "Trends"), Money(80000))
    val jane = Employee(Name("Jane", "Doe"), Money(90000))
    val employees = Map(buck.name -> buck, jane.name -> jane)

    val p = new PayrollParserCombinators(employees)

    args.foreach { filename =>
      val src = scala.io.Source.fromFile(filename)
      val lines = src.mkString
      p.parseAll(p.paycheck, lines) match {
        case p.Success(Pair(employee, paycheck),_) =>
          print(format("%s %s: %s\n", employee.name.first, employee.name.last, paycheck))
        case x => print(x.toString)
      }
      src.close()
    }
  }
}
```

Domain Specific Languages (DSLs)
Giving a Semantics to the DSL (Cont'd)

Considering the following

- 2 correct programs in the new DSL

```
paycheck for employee "Jane Doe"
is salary for 2 weeks minus deductions for {}
```

```
paycheck for employee "Buck Trends"
is salary for 2 weeks minus deductions for {
        federal income tax is 25. percent of gross,
        state income tax is 5. percent of gross,
        insurance premiums are 500. in gross currency,
        retirement fund contributions are 10. percent of gross
}
```

- and the wrong (inexistent employee) program

```
paycheck for employee "John Doe"
is salary for 2 weeks minus deductions for {}
```

They behave as follows

```
[16:29]cazzola@surtur:~/lp/scala/>scala PayRollBuilder test1.pr test2.pr test3.pr
Jane Doe: Paycheck($3461.54,$3461.54,$0.00)
Buck Trends: Paycheck($3076.92,$1346.15,$1730.77)
payroll.pcdsl.UnknownEmployee: Name(John,Doe)
  at payroll.pcdsl.PayrollParserCombinators$$anonfun$empl$4.apply(payroll-pc.scala:24)
```

References

- Martin Odersky and Matthias Zenger.
  Scalable Component Abstractions.
  In Richard P. Gabriel, editor, Proceedings of 19th ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'05), pages 41–57, San Diego, CA, USA, October 2005. ACM Press.
- Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black.
  Traits: Composable Units of Behaviour.
  In Luca Cardelli, editor, Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03), Lecture Notes in Computer Science 2743, pages 248–274, Darmstadt, Germany, July 2003. Springer.
- Venkat Subramaniam.
  Programming Scala.
  The Pragmatic Bookshelf, June 2009.
- Dean Wampler and Alex Payne.
  Programming Scala.
  O'Reilly, September 2009.