Parser Combinators

Lennart Andersson

Revision 2012-09-16

2012 Scala seminar

Agenda

- Functional programming with Scala
- Domain specific language
- Implementation of parser combinators

Parser combinators

How I learned about it:

- Burge: *Recursive Programming Techniques*. Addison-Wesley, 1975.
- Hutton: *Higher-order functions for parsing*. Journal of Functional Programming 2 (3), 1992.
- Wadler: *Monads for functional programming*. Marktoberdorf Summer School on Program Design Calculi, Springer, 1992.
- Leijen, Meijer: *Parsec — Direct Style Monadic Parser Combinators for the Real World*, Technical Report UU-CS-2001-35, Utrecht University, 2001.

BNF Grammar and Parser program

$$
\begin{array}{lll}
\text{expr} & ::= & \text{term} \; (\; "+" \; \text{term} \; | \; "\text{-}" \; \text{term} \;)^* \\
\text{term} & ::= & \text{factor} \; (\; "*" \; \text{factor} \; | \; "/" \; \text{factor} \;)^* \\
\text{factor} & ::= & \text{number} \; | \; \text{variable} \; | \; "(" \; \text{expr} \; ")"
\end{array}
$$

BNF Grammar and Parser program

$$
\begin{array}{lcl}
\text{expr} & ::= & \text{term ("+" term } \mid \text{ "-" term)*} \\
\text{term} & ::= & \text{factor ("}*\text{" factor } \mid \text{ "/" factor)*} \\
\text{factor} & ::= & \text{number } \mid \text{ variable } \mid \text{ "(" expr ")"}
\end{array}
$$

$$
\begin{array}{l}
\textbf{def } \text{expr} = \text{term} \sim (''+'' \sim \text{term} \mid ''-'' \sim \text{term})* \\
\textbf{def } \text{term} = \text{factor} \sim (''*'' \sim \text{factor} \mid ''/'' \sim \text{factor})* \\
\textbf{def } \text{factor} = \text{number} \mid \text{variable} \mid ''('' \sim \text{expr} \sim '')''
\end{array}
$$

BNF Grammar and Parser program

$$\begin{array}{lll} \text{expr} & ::= & \text{term ("+" term \| "-" term)*} \\ \text{term} & ::= & \text{factor ("*" factor \| "/" factor)*} \\ \text{factor} & ::= & \text{number \| variable \| "(" expr ")"} \end{array}$$

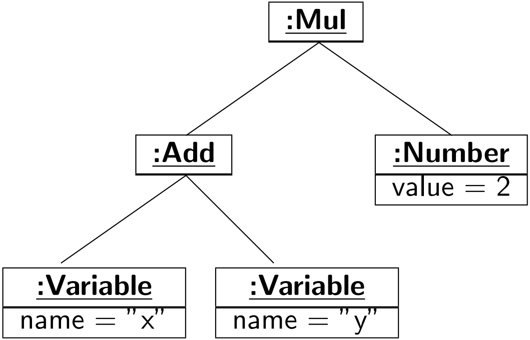
def expr = term ~ ("+" ~ term | "−" ~ term)∗
def term = factor ~ ("∗" ~ factor | "/" ~ factor)∗
def factor = number | variable | "(" ~ expr ~ ")"

expr("(x+y)∗2") −> ok

expr("(x+y)2") −> error

Build an abstract syntax tree

```
expr("(x+y)*2") ->
```



Expr classes

```
abstract class Expr

case class Number(value: Int) extends Expr
case class Variable(name: String) extends Expr
case class Add(expr1: Expr, expr2: Expr) extends Expr
case class Sub(expr1: Expr, expr2: Expr) extends Expr
case class Mul(expr1: Expr, expr2: Expr) extends Expr
case class Div(expr1: Expr, expr2: Expr) extends Expr
```

Result

```
abstract class Result[T]

case class Success[T](t: T, string: String) extends Result[T]

case class Failure[T]() extends Result[T]
```

Result with map

```
abstract class Result[T] {
  def map[U](f: T => U): Result[U]
}

case class Success[T](t: T, string: String) extends Result[T] {
  def map[U](f: T => U) = Success(f(t), string)
 }

case class Failure[T]() extends Result[T] {
  def map[U](f: T => U) = Failure[U]
}
```

Result with map

```
abstract class Result[T] {
  def map[U](f: T => U): Result[U]
}

case class Success[T](t: T, string: String) extends Result[T] {
  def map[U](f: T => U) = Success(f(t), string)
 }

case class Failure[T]() extends Result[T] {
  def map[U](f: T => U) = Failure[U]
}

Success('1', "abc") map (_.toInt) -> Success(49, abc)

Failure[Char]() map (_.toInt) -> Failure()
```

trait Parsers

```
abstract class Parser[T] extends (String => Result[T]) {
  def acceptIf(predicate: T => Boolean): Parser[T] = ...
  def |(parser: Parser[T]): Parser[T] = ...
  def ~[U](parser: Parser[U]): Parser[(T, U)] = ...
  def map[U](f: T => U): Parser[U] = ...
  def * : Parser[List[T]] = ...
}
```

trait Parsers

```
abstract class Parser[T] extends (String => Result[T]) {
  def acceptIf(predicate: T => Boolean): Parser[T] = ...
  def |(parser: Parser[T]): Parser[T] = ...
  def ~[U](parser: Parser[U]): Parser[(T, U)] = ...
  def map[U](f: T => U): Parser[U] = ...
  def * : Parser[List[T]] = ...
}
```

```
char: Parser[Char]
digit: Parser[Char]

variable: Parser[Variable]
number: Parser[Number]
expr: Parser[Expr]
```

newParser(f)

```
def newParser[T](f: String => Result[T]) = new Parser[T] {
  def apply(string: String) = f(string)
}
```

newParser(f)

```
def newParser[T](f: String => Result[T]) = new Parser[T] {
  def apply(string: String) = f(string)
}
```

```
def all = newParser(string => Success(string, ""))
```

```
all.apply("abc") -> Success(abc, )
all("abc") -> Success(abc, )
```

```
def result[T](t: T) = newParser(string => Success(t, string))
```

```
result(1)("abc") -> Success(1, abc)
```

char: Parser[Char]

char(''abc'') => Success(a, bc)

char('''') => Failure()

char: Parser[Char]

char(''abc'') => Success(a, bc)

char('''') => Failure()

```
def char = newParser(string =>
  if (string.length > 0) Success(string.head, string.tail)
  else Failure()
)
```

The acceptIf method

```
def digit = char acceptIf(_.isDigit)

digit("123") -> Success(1, 23)
digit("abc") -> Failure()
```

The acceptIf method

```
def digit = char acceptIf(_.isDigit)

digit("123") -> Success(1, 23)
digit("abc") -> Failure()


def acceptIf(predicate: T => Boolean) = newParser(string =>
  this(string) match {
    case Success(t, string1) if predicate(t) => Success(t, string1)
    case _ => Failure[T]
  }
)
```

The | method

def alphanum = letter | digit

alphanum(''abc'') −> Success(a, bc)
alphanum(''123'') −> Success(1, 23)
alphanum(''∗'') −> Failure()

The | method

```
def alphanum = letter | digit

alphanum("abc") -> Success(a, bc)
alphanum("123") -> Success(1, 23)
alphanum("*") -> Failure()


def |(parser: Parser[T]) = newParser(string =>
  this(string) match {
    case Failure() => parser(string)
    case success => success
  }
)
```

The ∼ method

def twoChars: Parser[(Char, Char)] = char ∼ char

twoChars(''abc'') −> Success((a, b), c)
twoChars(''a'') −> Failure()

The ~ method

```
def twoChars: Parser[(Char, Char)] = char ~ char

twoChars(''abc'') −> Success((a, b), c)
twoChars(''a'') −> Failure()


def ~[U](parser: => Parser[U]) = newParser(string =>
  this(string) match {
    case Success(t, string1) => parser(string1) match {
      case Success(u, string2) => Success((t, u), string2)
      case Failure() => Failure[(T, U)]
    }
    case Failure() => Failure[(T, U)]
  }
)
```

The map method

digit(''1'') map(_.toInt) −> Success(49,)

def secondChar = (char ∼ char) map(pair => pair._2)
secondChar(''abc'') −> Success(b, c)

The map method

```
digit("1") map(_.toInt) -> Success(49, )

def secondChar = (char ~ char) map(pair => pair._2)
secondChar("abc") -> Success(b, c)

def map[U](f: T => U) = newParser(string => this(string).map(f))
```

The map method

digit("1") map(_.toInt) -> Success(49,)

def secondChar = (char ~ char) map(pair => pair._2)
secondChar("abc") -> Success(b, c)

def map[U](f: T => U) = newParser(string => **this**(string).map(f))

Exercise: Define ~> and <~ so that

(letter ~> digit)("a1") -> Success(1,)
(letter <~ digit)("a1") -> Success(a,)

The * method

def digits: Parser[List[Char]] = digit*

digits(''123'') −> Success(List(1, 2, 3),)
digits(''abc'') −> Success(List(), abc)

The * method

```
def digits: Parser[List[Char]] = digit*

digits("123") -> Success(List(1, 2, 3), )
digits("abc") -> Success(List(), abc)
```

```
def * : Parser[List[T]] =
  ((this ~ (this*)) map(pair => pair._1 :: pair._2)) |
    result(List())
```

Some useful parsers

```
def letters = letter*
def word = letters acceptIf(_.length > 0) map(_.mkString)
def variable = word map(Variable)
```

```
variable(''abc'') -> Success(Variable(abc), )
```

Some useful parsers

```
def letters = letter*
def word = letters acceptIf(_.length > 0) map(_.mkString)
def variable = word map(Variable)

variable("abc") -> Success(Variable(abc), )
```

Exercise:

```
def number = ...
```

Expr parsers

factor ::= number | variable | "(" expr ")"

Expr parsers

factor ::= number | variable | "(" expr ")"

```
def factor = number | variable | accept("(") ~> expr <~ accept(")")

implicit def accept(required: String) = newParser(string =>
  if (string.startsWith(required))
      Success(required, string.substring(required.length()))
  else Failure()
)
```

Expr parsers

$$\text{factor} ::= \text{number} \mid \text{variable} \mid \text{"("}\ \text{expr}\ \text{")"}$$

```
def factor = number | variable | accept("(") ~> expr <~ accept(")")

implicit def accept(required: String) = newParser(string =>
  if (string.startsWith(required))
      Success(required, string.substring(required.length()))
  else Failure()
)


def factor = number | variable | "(" ~> expr <~ ")"
```

Expr parsers

term ::= factor ("*" factor | "/" factor)*

Expr parsers

term ::= factor ("*" factor | "/" factor)*

def term1 = factor ~ ("*" ~ factor | "/" ~ factor)*

term1: Parser[(Expr, List[(String, Expr)])]

Expr parsers

term ::= factor ("*" factor | "/" factor)*

def term1 = factor ~ ("*" ~ factor | "/" ~ factor)*

term1: Parser[(Expr, List[(String, Expr)])]

```
def term = term1 map {
  case (factor1, list) => list.foldLeft(factor1) {
    case (factor2, ("*", factor3)) => Mul(factor2, factor3)
    case (factor2, ("/", factor3)) => Div(factor2, factor3)
  }
}
```

Expr parsers

term ::= factor ("*" factor | "/" factor)*

```
def term1 = factor ~ ("*" ~ factor | "/" ~ factor)*
```

term1: Parser[(Expr, List[(String, Expr)])]

```
def term = term1 map {
  case (factor1, list) => list.foldLeft(factor1) {
    case (factor2, ("*", factor3)) => Mul(factor2, factor3)
    case (factor2, ("/", factor3)) => Div(factor2, factor3)
  }
}
```

Exercise:

```
def expr = ...
```

How did I do it?

1. Straightforward translation from Haskell. The outcome was not as elegant as the Haskell version since you need to enter the object oriented part of Scala to use infix operators.
2. Change to infix operators.
3. I avoided foldLeft when parsing expressions since the context is quite complicated. I prefer using recursion instead of iteration, but since the arithmetic operators associate to the left there is another complication to handle.
4. Then I got the text book and found that Oderskys solution was very close to mine. I borrowed some from the book, switched to his operator identifiers and used the foldLeft solution from Scaladoc for RegexParsers.

Difficulties encountered

1. I was frustrated by the Scala type system. It can just derive types when it is trivial to do it. It doesn't even try when a function is recursive. I did not understand one error message: ´´Found: long string , expected: same string“.
2. I found it difficult to decide exactly where you need name parameters to avoid non-terminating execution. I am still not sure that may solution is correct. When you make abstractions you should not have think about execution details.
3. I avoided foldLeft when parsing expressions since the context is quite complicated. I prefer using recursion instead of iteration, but since the arithmetic operators associate to the left there is another complication to handle.
4. I had to ask Christian about pattern matching when Scala was not up to my expectations.

Summary

- What is parser combinator?
- How to use it.
- How to implement it.

Links

1. Andersson: EDAN40: Functional Parsing. Assignment.
2. Andersson: Parsing in Haskell. Tutorial.
3. Andersson: Parser combinators in Scala. Slides.
4. Andersson: Parser combinators in Scala. Implementation.
5. Hutton: Higher-order functions for parsing.
6. Wadler: Monads for functional programming
7. Leijen, Meijer: Parsec — Direct Style Monadic Parser Combinators for the Real World.