Parser Combinators

Henk Erik van der Hoek

mail@henkerikvanderhoek.nl

December 15, 2014

What is a parser?

Parsing is the process of converting unstructured text into a structured data type. For example:

- Parsing programming code into an AST
- Parsing a network message (like a HTTP request)
- Parsing data in a specific file format (like JSON)
- Parsing a configuration file
- Etc.

What is a parser?

Parsing is the process of converting unstructured text into a structured data type. For example:

- Parsing programming code into an AST
- Parsing a network message (like a HTTP request)
- Parsing data in a specific file format (like JSON)
- Parsing a configuration file
- Etc.

In this presentation we will develop a parser for JSON.

Overview

- Part I: Write a parser combinator library
- Part II: Write a JSON parser using the library

Overview

- **Part I: Write a parser combinator library**
- Part II: Write a JSON parser using the library

A type for a parser

Let's start with a type called *Parser*:

```
case class Parser[+A](parse: String => List[(A,String)])
```

This states that a *Parser* is a function which takes a string as its first and only argument and returns a list of results. Failure is denoted by an empty list while a non empty lists denotes success.

A type for a parser

Let's start with a type called *Parser*:

```
case class Parser[+A](parse: String => List[(A,String)])
```

This states that a *Parser* is a function which takes a string as its first and only argument and returns a list of results. Failure is denoted by an empty list while a non empty lists denotes success.

This approach is known as the list-of-successors and was first described by Philip Wadler. There are more approaches to define the *Parser* type, but the list-of-successors approach is the easiest to understand.

Recognizing a prefix

The simplest parser is a parser which just recognizes if a string has a given prefix. For example: the *pHello* parser recognizes if the input starts with the prefix *Hello*.

```
object Parser {
  def pHello:Parser[Unit] =
    Parser { inp => inp.startsWith("Hello") match {
      case true => List((Unit,inp.stripPrefix("Hello")))
      case false => Nil
    }}
}
```

Recognizing a prefix

The simplest parser is a parser which just recognizes if a string has a given prefix. For example: the *pHello* parser recognizes if the input starts with the prefix *Hello*.

```
object Parser {
  def pHello:Parser[Unit] =
    Parser { inp => inp.startsWith("Hello") match {
      case true => List((Unit,inp.stripPrefix("Hello")))
      case false => Nil
    }}
}
```

Remember that *Parser* is defined as:

```
case class Parser[+A](parse: String => List[(A,String)])
```

Parsing a given character

The previous parser only recognized the input, but it didn't return a result.

Let's write a parser which recognizes if the input starts with a given character **and** gives this character as a result:

```
object Parser {
  def pChar(c:Char):Parser[Char] =
    Parser { inp => !inp.isEmpty && inp.head == c match {
      case true => List((inp.head,inp.tail))
      case false => Nil
    }}
}
```

Parsing a given character

Next we define a parser which successfully consumes the first character if and only if it satisfies a given predicate function:

```
object Parser {
  def pSatisfy(p:Char => Boolean):Parser[Char] =
    Parser { inp => !inp.isEmpty && p(inp.head) match {
      case true => List((inp.head,inp.tail))
      case false => Nil
    }}
}
```

Parsing a given character

Next we define a parser which successfully consumes the first character if and only if it satisfies a given predicate function:

```
object Parser {
  def pSatisfy(p:Char => Boolean):Parser[Char] =
    Parser { inp => !inp.isEmpty && p(inp.head) match {
      case true => List((inp.head,inp.tail))
      case false => Nil
    }}
}
```

We can use this function to redefine the *pChar* function:

```
object Parser {
  def pChar(c:Char):Parser[Char] = pSatisfy { x => x == c }
}
```

Parsing a single digit

In a similiar fashion we can define a parser to parse a single digit.

```
object Parser {
  def pDigit:Parser[Char] =
    pSatisfy { c => ('0' to '9').contains(c) }
}
```

Parsing a single digit

In a similiar fashion we can define a parser to parse a single digit.

```
object Parser {
  def pDigit:Parser[Char] =
    pSatisfy { c => ('0' to '9').contains(c) }
}
```

The result of this parser is a *Char*, but wouldn't it make more sense if this parser returned an *Integer*?

The *map* combinator

Given a parser of type *Parser*[*A*] and a function from *A* to *B*, we wish to construct a parser of type *Parser*[*B*]:

```
case class Parser[+A](parse: String => List[(A,String)]) {
  def map[B](f: A => B):Parser[B] = ???
}
```

Exercise: Implement the *map* function.

The *map* combinator

Given a parser of type *Parser*[*A*] and a function from *A* to *B*, we wish to construct a parser of type *Parser*[*B*]:

```
case class Parser[+A](parse: String => List[(A,String)]) {
  def map[B](f: A => B):Parser[B] = ???
}
```

Exercise: Implement the *map* function.

```
case class Parser[+A](parse: String => List[(A,String)]) {
  def map[B](f: A => B):Parser[B] = Parser { inp =>
    parse(inp).map { case (a,rem) => (f(a),rem) }
  }
  def as[B](b: B):Parser[B] = map { Function.const(b) }
}
```

The *map* combinator

Given a parser of type $Parser[A]$ and a function from A to B, we wish to construct a parser of type $Parser[B]$:

```
case class Parser[+A](parse: String => List[(A,String)]) {
  def map[B](f: A => B):Parser[B] = ???
}
```

Exercise: Implement the *map* function.

```
case class Parser[+A](parse: String => List[(A,String)]) {
  def map[B](f: A => B):Parser[B] = Parser { inp =>
    parse(inp).map { case (a,rem) => (f(a),rem) }
  }
  def as[B](b: B):Parser[B] = map { Function.const(b) }
}
```

Note: The *map* function obeys the functor laws (we will use this fact later on):

Identity:	p.map { x => x } ≡ p
Composition:	p.map { x => f(x) }.map { x => g(x) } ≡ p.map { x => g (f(x)) }

The *map* combinator

Using this function we can parse a single digit and transform the result from *Char* to *Integer*:

```
object Parser {
  def pDigit:Parser[Integer] =
    pSatisfy { c => ('0' to '9').contains(c) }.map { c =>
      c.asDigit
    }
}
```

The *flatMap* combinator

The *flatMap* function is used to sequence two parsers where the second parser may depend on the result of the first parser:

```
case class Parser[+A](parse: String => List[(A,String)]) {
  def flatMap[B](f: A => Parser[B]):Parser[B] = ???
}
```

Exercise: Implement the *flatMap* function.

The *flatMap* combinator

The *flatMap* function is used to sequence two parsers where the second parser may depend on the result of the first parser:

```
case class Parser[+A](parse: String => List[(A,String)]) {
  def flatMap[B](f: A => Parser[B]):Parser[B] = ???
}
```

Exercise: Implement the *flatMap* function.

```
case class Parser[+A](parse: String => List[(A,String)]) {
  def flatMap[B](f: A => Parser[B]):Parser[B] = Parser { inp =>
    parse(inp).map { case (a,rem) => f(a).parse(rem) }.flatten
  }

  def sequence[B](p: => Parser[B]):Parser[B] =
    flatMap { Function.const(p) }
}
```

A parser that always succeeds

Let's write a parser which always succeeds without consuming any characters of the input string:

```
object Parser {
  def apply[A](a: => A):Parser[A] =
    Parser { inp => List((a,inp)) }
}
```

A parser that always succeeds

Let's write a parser which always succeeds without consuming any characters of the input string:

```
object Parser {
  def apply[A](a: => A):Parser[A] =
    Parser { inp => List((a,inp)) }
}
```

Specialized for the *Parser* type, the *apply* and the *flatMap* function obey the monad laws:

Left identity

`Parser(x).flatMap { x => f(x) }` $\equiv$ `f(x)`

Right identity

`mx.flatMap { x => Parser(x) }` $\equiv$ `mx`

Associativity

`mx.flatMap { x => f(x) }.flatMap { y => g(y) }`

$\equiv$

`mx.flatMap { x => f(x).flatMap { y => g(y) } }`

Implementing the Kleene cross operation

A *map* function obeying the functor laws and a *flatMap* function obeying the monad laws enables us to use the for-comprehension notation. This allows us to implement the Kleene cross operation quite elegantly:

```
object Parser {
  def pOneOrMore[A](p: => Parser[A]):Parser[List[A]] = for {
    a  <- p
    as <- pZeroOrMore(p)
  } yield a::as
}
```

Implementing the Kleene cross operation

A *map* function obeying the functor laws and a *flatMap* function obeying the monad laws enables us to use the for-comprehension notation. This allows us to implement the Kleene cross operation quite elegantly:

```
object Parser {
  def pOneOrMore[A](p: => Parser[A]):Parser[List[A]] = for {
    a  <- p
    as <- pZeroOrMore(p)
  } yield a::as
}
```

The Scala compiler desugars the for-comprehension to plain *map* and *flatMap* invocations:

```
object Parser {
  def pOneOrMore[A](p: => Parser[A]):Parser[List[A]] =
    p.flatMap { a => pZeroOrMore(p).map { as => a::as }}
}
```

Implementing the Kleene star operation

Let's assume we have another basic combinator *choice* which combines the result of two parsers. This allows us to write the Kleene star operation:

```
object Parser {
  def pZeroOrMore[A](p: => Parser[A]):Parser[List[A]] =
    pOneOrMore(p).choice(Parser(Nil))
}
```

Implementing the Kleene star operation

Let's assume we have another basic combinator *choice* which combines the result of two parsers. This allows us to write the Kleene star operation:

```
object Parser {
  def pZeroOrMore[A](p: => Parser[A]):Parser[List[A]] =
    pOneOrMore(p).choice(Parser(Nil))
}
```

Where, the type signature of *choice* is given by:

```
case class Parser[+A](parse: String => List[(A,String)]) {
  def choice[B >: A](p: => Parser[B]):Parser[B] = ???
}
```

Exercise: Implement the *choice* combinator

Implementing the Kleene star operation

Let's assume we have another basic combinator *choice* which combines the result of two parsers. This allows us to write the Kleene star operation:

```
object Parser {
  def pZeroOrMore[A](p: => Parser[A]):Parser[List[A]] =
    pOneOrMore(p).choice(Parser(Nil))
}
```

Where, the type signature of *choice* is given by:

```
case class Parser[+A](parse: String => List[(A,String)]) {
  def choice[B >: A](p: => Parser[B]):Parser[B] = ???
}
```

Exercise: Implement the *choice* combinator

```
case class Parser[+A](parse: String => List[(A,String)]) {
  def choice[B >: A](p: => Parser[B]):Parser[B] =
    Parser { inp => parse(inp) ++ p.parse(inp) }
}
```

Dealing with whitespace: parsing tokens

First we define a parser to parse solely whitespace:

```
object Parser {
  def pWhitespace = pZeroOrMore(pSatisfy { c => ' ' == c })
}
```

Dealing with whitespace: parsing tokens

First we define a parser to parse solely whitespace:

```
object Parser {
  def pWhitespace = pZeroOrMore(pSatisfy { c => ' ' == c })
}
```

Now, given a parser the *pToken* function returns another parser which parses the given parser **and** any additional trailing whitespace:

```
object Parser {
  def pToken[A](p: => Parser[A]):Parser[A] =
    for { a <- p; _ <- pWhitespace } yield a
}
```

Dealing with whitespace: parsing tokens

First we define a parser to parse solely whitespace:

```
object Parser {
  def pWhitespace = pZeroOrMore(pSatisfy { c => ' ' == c })
}
```

Now, given a parser the *pToken* function returns another parser which parses the given parser **and** any additional trailing whitespace:

```
object Parser {
  def pToken[A](p: => Parser[A]):Parser[A] =
    for { a <- p; _ <- pWhitespace } yield a
}

case class Parser[+A](parse: String => List[(A,String)]) {
  def run(inp: => String):List[(A,String)] =
    pWhitespace.sequence(this).parse(inp)
}
```

Parsing symbols

Given a string the *pSym* function returns a parser which parses the given string **and** any additional trailing whitespace:

```
object Parser {
  def pSym(str:String):Parser[String] = {
    def go(xs:String):Parser[String] = xs.isEmpty match {
      case true => Parser ("")
      case false => for {
        a <- pChar(xs.head)
        as <- go(xs.tail)
      } yield a + as
    }

    pToken(go(str))
  }
}
```

Parsing symbols

Given a string the *pSym* function returns a parser which parses the given string **and** any additional trailing whitespace:

```
object Parser {
  def pSym(str:String):Parser[String] = {
    def go(xs:String):Parser[String] = xs.isEmpty match {
      case true => Parser ("")
      case false => for {
        a <- pChar(xs.head)
        as <- go(xs.tail)
      } yield a + as
    }

    pToken(go(str))
  }
}
```

We can parse specific keywords using this function: *pSym*("*protected*")

Enclosing a parser

Quite often we wish to enclose a parser with a symbol on the left and one on the right, but we wish to only keep the result of the enclosed parser:

```
object Parser {
  def pEnclose[L,A,R](
    l: => Parser[L],
    p: => Parser[A],
    r: => Parser[R]
  ):Parser[A] = for { _ <- l; a <- p; _ <- r } yield a
}
```

Enclosing a parser

Quite often we wish to enclose a parser with a symbol on the left and one on the right, but we wish to only keep the result of the enclosed parser:

```
object Parser {
  def pEnclose[L,A,R](
    l: => Parser[L],
    p: => Parser[A],
    r: => Parser[R]
  ):Parser[A] = for { _ <- l; a <- p; _ <- r } yield a
}
```

Using this combinator we define a function *pBraces*:

```
object Parser {
  def pBraceL = pSym("{")
  def pBraceR = pSym("}")

  def pBraces[A](p: => Parser[A]):Parser[A] =
    pEnclose(pBraceL, p, pBraceR)
}
```

Parsing a separated list

The function *pSepBy*(*p*, *q*) parses zero or more occurrences of p, separated by q and returns a list of values returned by p.

```
object Parser {
  def pSepBy1[A,B](
    p: => Parser[A],
    q: => Parser[B]
  ):Parser[List[A]] = for {
    a  <- p
    as <- pZeroOrMore (q.sequence(p))
  } yield a::as

  def pSepBy[A,B](
    p: => Parser[A],
    q: => Parser[B]
  ):Parser[List[A]] = pSepBy1(p, q).choice(Parser(Nil))
}
```

Overview

- Part I: Write a parser combinator library
- **Part II: Write a JSON parser using the library**

Parsing JSON

We shall use our small parser combinator library to write a parser for JSON. A JSON value is always one of the following values:

- Null
- Boolean
- Number
- String
- Array
- Object

A JSON data type

Given the following algebraic data type:

```scala
sealed trait JsonValue
case class JsonNull() extends JsonValue
case class JsonBoolean (value: Boolean) extends JsonValue
case class JsonNumber (value: Integer) extends JsonValue
case class JsonString (value: String) extends JsonValue
case class JsonArray (value: List[JsonValue]) extends JsonValue
case class JsonObject (value: Map[String,JsonValue]) extends ..
```

A JSON data type

Given the following algebraic data type:

```
sealed trait JsonValue
case class JsonNull() extends JsonValue
case class JsonBoolean (value: Boolean) extends JsonValue
case class JsonNumber (value: Integer) extends JsonValue
case class JsonString (value: String) extends JsonValue
case class JsonArray (value: List[JsonValue]) extends JsonValue
case class JsonObject (value: Map[String,JsonValue]) extends ..
```

we wish to write a function *pJsonValue*:

```
object JsonParser {
  def pJsonValue:Parser[JsonValue]
}
```

Parsing JSON

To parse a JSON null we define a function *pJsonNull*:

```
object JsonParser {
  def pJsonNull:Parser[JsonNull] = pSym("null").as(JsonNull())
}
```

Parsing JSON

To parse a JSON null we define a function *pJsonNull*:

```
object JsonParser {
  def pJsonNull:Parser[JsonNull] = pSym("null").as(JsonNull())
}
```

To parse a JSON number we define a function *pJsonNumber*:

```
object JsonParser {
  def pJsonNumber:Parser[JsonNumber] =
    pToken(pDigits.map { n => JsonNumber(n) })
}
```

Parsing JSON

To parse a JSON null we define a function *pJsonNull*:

```
object JsonParser {
  def pJsonNull:Parser[JsonNull] = pSym("null").as(JsonNull())
}
```

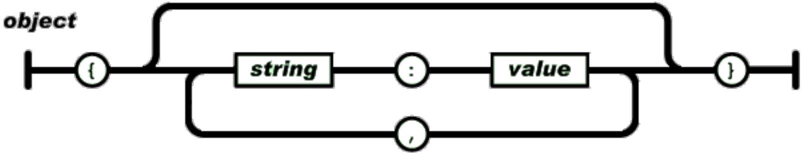
To parse a JSON number we define a function *pJsonNumber*:

```
object JsonParser {
  def pJsonNumber:Parser[JsonNumber] =
    pToken(pDigits.map { n => JsonNumber(n) })
}
```

To parse a JSON string we define a function *pJsonString*:

```
object JsonParser {
  def pJsonString:Parser[JsonString] =
    pString.map { str => JsonString(str) }
}
```

Parsing JSON



To parse a JSON object we define a function *pJsonObject*:

```
object JsonParser {
  def pJsonObject:Parser[JsonObject] = {
    val pPair = for {
      k <- pString
      _ <- pColon
      v <- pJsonValue
    } yield (k,v)

    pBraces(pSepBy(pPair, pComma)).map
      { xs => JsonObject(xs.toMap) }
  }
}
```

Parsing JSON

Lab: Implement *pJsonBoolean* and *pJsonArray*.

```
git clone https://github.com/henkerik/parsers.git
git checkout lab-json
```

Parsing JSON

Lab: Implement *pJsonBoolean* and *pJsonArray*.

```
git clone https://github.com/henkerik/parsers.git
git checkout lab-json
```

```
object JsonParser {
  def pJsonBoolean:Parser[JsonBoolean] = {
    val pTrue = pSym("true").as(JsonBoolean(true))
    val pFalse = pSym("false").as(JsonBoolean(false))
    pTrue choice pFalse
}
```

Parsing JSON

Lab: Implement *pJsonBoolean* and *pJsonArray*.

```
git clone https://github.com/henkerik/parsers.git
git checkout lab-json
```

```
object JsonParser {
  def pJsonBoolean:Parser[JsonBoolean] = {
    val pTrue = pSym("true").as(JsonBoolean(true))
    val pFalse = pSym("false").as(JsonBoolean(false))
    pTrue choice pFalse
}
```

To parse a JSON array we define a function *pJsonArray*:

```
object JsonParser {
  def pJsonArray:Parser[JsonArray] =
    pBrackets(pSepBy(pJsonValue, pComma)).map
      { xs => JsonArray(xs) }
}
```

Parsing JSON

So, finally, Let's implement our last function *pJsonValue*:

```
object JsonParser {
  def pJsonValue:Parser[JsonValue] =
      pJsonObject.choice(pJsonNumber)
                 .choice(pJsonArray)
                 .choice(pJsonBoolean)
                 .choice(pJsonNull)
                 .choice(pJsonString)
}
```

Parsing JSON

So, finally, Let's implement our last function *pJsonValue*:

```
object JsonParser {
  def pJsonValue:Parser[JsonValue] =
      pJsonObject.choice(pJsonNumber)
                 .choice(pJsonArray)
                 .choice(pJsonBoolean)
                 .choice(pJsonNull)
                 .choice(pJsonString)
}
```

Now we can parse a JSON string:

```
object Test extends App {
  val result = pJsonParser.run
    ("""{ "boolean": true, "number": 100 }""")

  println(result)
}
```