12 — Parser Combinators

Einführung in die Funktionale Programmierung

Prof. Frank Puppe, Felix Herrmann, Alexander Gehrke

Sommersemester 2020

Lehrstuhl für Informatik VI, Uni Würzburg

Functional parsers

We learned lots of abstractions in the last weeks.

In this lecture, we will look at a more practical example and implement functional Parsers. We will see, how we can use our abstractions from before for that.

Types of parsers

So we have some data format we want to parse. What approaches could we use?

Existing parser:	if there is one, *use it*.
Very simple format:	maybe `String.split("\n")` is enough?
Regexes:	enough for some formats, but fragile and limited
Parser generated from grammar:	Commonly used, difficult to debug and reuse
Handwritten Recursive Parser:	most flexible and always possible, but tedious

Parser combinators

A functional approach to parsing are parser combinators. We model parsers as functions for small parts of the text, that can be composed into larger parsers with various combinators.

We will look at designing a library for this top-down, starting with the Algebra.

Basic operations

We start with the smallest unit we will parse, a character. We want a parser that can recognize a given character:

```
def char(c: Char): Parser[Char]
```

This method shall create a parser for a given char. The type `Parser[A]` will represent a parser which gives us an A when run on some input.

Running a parser

We said we want to represent parsers as functions. We need some string as input and produce a value of the result type:

```
String => A
```

But we want small parsers that we can compose, so most parsers will only process a part of the input.

Parsers as functions

Let's return the remaining input from our function. And while we are at it, we will also add error handling:

```
enum ParseResult[+A]: // similar to Either
  case Fail(remain: String, error: String) extends ParseResult[Nothing]
  case Done(remain: String, result: A)
```

So we can define our signature for running a parser:

```
trait Parser[+A]:
  def parse(input: String): ParseResult[A]
```

A simple identity law

We can define a simple law regarding parse for the character parser. If we input a single character's string representation, parsing it with a `char` parser should return the same char, without more input remaining:

```
char(c).parse(c.toString) == Done(remain = "", result = c)
```

String parser

We probably also want a parser that can read more than one character, without concatenating lots of character parsers. Let's add a String parser:

```
def string(s: String): Parser[String]
```

It has a similar law:

```
string(s).parse(s) == Done(remain = "", result = s)
```

Regex parser

For good measure, we also add a parser matching against a regex. We could also implement most of that functionality via combinators, but that's cumbersome if we don't need the parts of the regex.

```
def regex(r: Regex): Parser[String]
```

String parser

We can now write parsers recognizing strings:

```
string("a").parse("abc") == Done(remain = "bc", result = "a")
```

Just parsing constant strings is something we can also do with the `startsWith` method. Let's think about what operations we would like to make the parsers more useful.

Combinators

What if we want to recognize, if one of two strings is there? We add a combinator representing an "or" operation to our Parser:

```
def | [B>:A](pb: Parser[B]): Parser[B]
```

We can use it like this:

```
val orParser = string("a") | string("b")
orParser.parse("all") == Done("ll", "a")
orParser.parse("ball") == Done("all", "b")
```

Another commonly needed functionality is repetition of some matched pattern. So we want to add a combinator, that applies a parser until it doesn't match the remainder anymore, and then returns all matches.

We add such a combinator to our Parser:

```
def many: Parser[List[A]]
```

We can apply it to any parser:

```
orParser.many.parse("abbabcd") == Done("cd", List("a", "b", "b", "a", "b"))
```

Combinators

Another combinator we would like is chaining several parsers after each other. In parser libraries the operator ~ is commonly used for that:

```
def ~[B](next: => Parser[B]): Parser[(A, B)]
```

A usage example with some parsers of type `Parser[String]`:

```
val keyValueParser = keyword ~ whitespace ~ value
```

What would be the type of this parser?

Combinators

Another combinator we would like is chaining several parsers after each other. In parser libraries the operator ~ is commonly used for that:

```
def ~[B](next: => Parser[B]): Parser[(A, B)]
```

A usage example with some parsers of type `Parser[String]`:

```
val keyValueParser = keyword ~ whitespace ~ value
```

What would be the type of this parser?

`Parser[((String, String),String)]`

Typeclasses for our parser

We can now parse strings into parts that are also strings, but usually we'll want something else. For example, if we have a parser that accepts digits, we probably want a numeric value.

We are given a digit parser (accepting any number of digits 0-9), returning the digits as a string.

```
def digits: Parser[String]
```

We'd like a `Parser[Int]` instead. Which typeclass could we implement for Parser to help us here?

Typeclasses for our parser

We want a `Functor` to provide us with `map`:

```
def int: Parser[Int] = digits.map(_.toInt)
```

Are there more typeclasses, that would be useful for our Parser? Let's think about `Monad`. What parsers would a `flatMap` method allow us to write, that `map` could not?

```
def flatMap[A, B](fa: Parser[A])(f: A => Parser[B]): Parser[B]
```

Typeclasses for our parser

As `flatMap` allows us to use a different Parser based on the result of a previous one, we can use it to parse context sensitive grammars.

This allows us to parse more complex languages. For example, we could have a file format with type annotations for values. Depending on the type annotation, we use another parser to parse the value.

Let's look at an example.

Typeclasses for our parser

We first parse a string, and then, depending on its content, we either parse the next token as an int or as a string:

```
val typedValue: Parser[Either[Int, String]] =
  for
    kw <- string("int") | string("string")          // parse the datatype
    _  <- whitespace                                // separate type and value
    value <- if kw == "int" then int.map(Left(_))   // parse as int
             else regex(".*".r).map(Right(_))       // parse as string
  yield value                                       // keep only value
```

```
typedValue.parse("int 34") == Done("", Left(34))
typedValue.parse("string 34") == Done("", Right("34"))
```

Typeclasses for our parser

To get a monad for `Parser`, we need an implementation for `pure`:

```
def pure[A](a: A): Parser[A]
```

How should the parser returned by `pure` behave?

Typeclasses for our parser

To get a monad for `Parser`, we need an implementation for `pure`:

```
def pure[A](a: A): Parser[A]
```

How should the parser returned by `pure` behave?

It should return the given value without consuming any input, i.e.

```
Monad[Parser].pure(a).parse(s) == Done(s, a)
```

Intermediate summary

Let's look at what we have until now:

`char(c)`	match the character `c`
`string(s)`	match the string `s`
`regex(r)`	match the regular expression `r`
`digits`	match numeric digits
`p1 | p2`	try `p1`, if it doesn't match try `p2`
`p1 ~ p2`	return tupled result of `p1` and `p2` if both match
`p.many`	apply p repeatedly and return a list of matches
`p.map(f)`	run the parser and transform its result
`p.flatMap(f)`	run the parser and change further parsing based on its result
`pure(a)`	returns a without consuming input

Which of these are primitive, i.e. can't be implemented via the others?

Intermediate summary

Let's look at what we have until now:

`char(c)`	match the character `c`
`string(s)`	match the string `s`
`regex(r)`	match the regular expression `r`
`digits`	match numeric digits
`p1 | p2`	try `p1`, if it doesn't match try `p2`
`p1 ~ p2`	return tupled result of `p1` and `p2` if both match
`p.many`	apply p repeatedly and return a list of matches
`p.map(f)`	run the parser and transform its result
`p.flatMap(f)`	run the parser and change further parsing based on its result
`pure(a)`	returns a without consuming input

Which of these are primitive, i.e. can't be implemented via the others?

Note: `string` can be built via `regex`, but implementing it directly is more efficient and easier

Adding more combinators

Our combinator `many` matches any number of occurences, including zero.

Write a combinator `many1`, which matches at least one element and returns a `NonEmptyList`!

```
def many1[A](p: Parser[A]): Parser[NonEmptyList[A]] =
```

Our combinator `many` matches any number of occurences, including zero.

Write a combinator `many1`, which matches at least one element and returns a `NonEmptyList`!

```
def many1[A](p: Parser[A]): Parser[NonEmptyList[A]] =
  (p ~ p.many).map((h, t) => NonEmptyList(h,t))
```

You may have noticed, that we added lots of functions to our API, but we never wrote any of the implementations for our primitives.

This way of designing an API helps uncoupling the representation of our data types from the algebra. We could keep everything else we implement private. As long as we fulfill the laws of our algebra, the implementation doesn't matter to the user.

Of course we still need an implementation somewhere, so we will take a look at it next.

Implementing our Parser API

Parsers as functions

We said in the beginning, that we want to see parsers as functions from String to some output. Our `Parser` trait has exactly one method:

```
trait Parser[+A]:
  def parse(input: String): ParseResult[A]
```

which allows implementing it by giving a function literal matching the signature:

```
def string(s: String): Parser[String] =
  input =>
    if input.startsWith(s) then Done(input.substring(s.length), s)
    else Fail(input, s"expected \"$s\"")
```

The `regex` parser is pretty similar.

Combinator implementations

The parser monad:

```
given Monad[Parser] with
  def pure[A](a: A): Parser[A] = input => Done(input, a)

  extension [A](fa: Parser[A])
    def flatMap[B](f: A => Parser[B]): Parser[B] =
      input => fa.parse(input) match
        case Done(rest, a) => f(a).parse(rest)
        case Fail(rest, msg) => Fail(rest, msg)
```

Combinator implementations

A pattern you will sometimes see in Scala libraries using operators (like our |, ~) is that the implementations are written as named methods, while the operator methods (either also a on the trait or as extensions) then delegate to these. This makes the intent of the operators clearer for people new to the library.

```
def | [B >: A](pb: Parser[B]): Parser[B] = Parser.or(this, pb)

def ~ [B](next: => Parser[B]): Parser[(A, B)] = Parser.andThen(this, next)
```

Combinator implementations

So what do the actual implementations look like?

```
def or[A](p1: Parser[A], p2: Parser[A]): Parser[A] =
  input =>
    p1.parse(input) match
      case Done(rest, out) => Done(rest, out)
      case Fail(_, _)      => p2.parse(input)
```

Apply the left parser first. If it matches, just return its result. If it doesn’t, discard its error and continue with the right parser.

Actual parser libraries may have a more complex implementation to allow for better error handling. Here we only get the error message from the second parser if none of them matches.

Combinator implementations

```scala
def andThen[A, B](pa: Parser[A], next: => Parser[B]): Parser[(A, B)] = for
  a <- pa
  b <- next
yield (a,b)
```

We combine both parsers using a for-comprehension. This uses our monad in the background, it is equivalent to:

```scala
pa.flatMap(a => next.map(b => (a,b)))
```

So if the first parser parses something sucessfully, the second one is called with the remaining input and then both results are put into a tuple. If the first one fails, the second one isn't run at all (similar to Either).

Combinator implementations

```
def many[A](pa: Parser[A]): Parser[List[A]] = (
  for
    a <- pa
    tail <- many(pa)
  yield a :: tail
) | summon[Monad[Parser]].pure(Nil)
```

Our `many` combinator works recursively. We first match the given parser `p` once and then match `many(p)` again. We prepend the result of `p` to the list created by the recursive `many(p)` parser.

Our recursion needs some stopping condition, but this is included in the for-comprehension: if `p` fails to match, the recursive call won't happen. But we only want to stop matching when this happens, not return a `Fail` to outside. So we turn a failure from the comprehension into an empty list.

Using the parsers

So we've learned a lot about the API of our parsers, but haven't seen them used on some more practical example. Let's parse a contact list:

```
case class Contact(name:String, address: Address, phone: Option[Phone])
case class Address(street: String, number: Int, postCode: Int, city: String)
case class Phone(prefix: String, suffix:String)
```

Our input format will look like this:

```
Max Mustermann
Hublandstraße 123, 97074 Würzburg
01234/555555
```

The phone line may be missing, so our `phone` field is an `Option`. To make it easier, we assume that the street name does not contain spaces.

Address parsing

We start with the address:

```
def address: Parser[Address] =
```

Implement this parser, so that it parses the address line of our format:

```
address.parse("Hublandstraße 123, 97074 Würzburg") ==
  Done("", Address("Hublandstraße", 123, 97074, "Würzburg"))
```

You may use any parsers and combinators we defined until now. Additionally these two are given:

```
val whitespace = regex(raw"\h+".r) // matches all whitespace
val word       = regex(raw"\S+".r) // matches everything but whitespace
```

Address parsing — Solution

```
def address: Parser[Address] =
  for
    street   <- word
    _        <- whitespace
    number   <- int
    _        <- string(",") ~ whitespace
    postCode <- int
    _        <- whitespace
    city     <- word
  yield Address(street, number, postCode, city)
```

After this, the phone number parser itself is pretty straigtforward:

```
def phone: Parser[Phone] =
  for
    prefix <- digits
    _      <- char('/') // or string("/")
    suffix <- digits
  yield Phone(prefix, suffix)
```

But for parsing our contact, the phone number may be absent. We could write a parser specifically for optional phone numbers, but optional parts seem like a more common problem. We should create a combinator that turns a `Parser[A]` into a `Parser[Option[A]]`.

```
def opt[A](p: Parser[A]): Parser[Option[A]] =
```

Contact parsing: Optional parser

```
def opt[A](p: Parser[A]): Parser[Option[A]] =
  p.map(Some(_)) | summon[Monad[Parser]].pure(None)
```

or using the syntax extensions from the cats library

```
def opt[A](p: Parser[A]): Parser[Option[A]] =
  p.map(Some(_)) | None.pure[Parser]
```

Contact parsing

With two more helpers for dealing with line breaks:

```
val toLineEnd: Parser[String] = regex(raw"\V+".r)
val newline: Parser[String] = regex(raw"\v".r)
```

we can now combine everything to get our `Parser[Contact]`:

```
def contact: Parser[Contact] =
  for
    name  <- line // read until newline
    _     <- newline
    addr  <- address
    phone <- opt((newline ~ phone).map(_._2))
  yield Contact(name, addr, phone)
```

In the last for-line, we parse a newline and a phone number and make the whole thing optional (so if there is no phone number, we also don't need a newline).

Libraries

Parser combinators are a common concept in functional programming, and there are various Scala libraries implementing them. Here is a small selection (with Github repo names):

scala/scala-parser-combinators previously part of the standard library, now separate but still used

tpolecat/atto based on cats typeclasses, our combinators are a subset of this library's API

lihaoyi/fastparse parser library focused on speed