

Parser Combinators

Chapter 9 of Pilquist/Chiusano/Bjarnason

What do we learn from Chapter 9?

- How to **use** a parser combinator **library**?
- Specify a simple language (JSON) **using** a grammar and **regexes**
- **Design** an internal DSL for expressing grammars in Scala
- Implement a Program Expression Grammar
- Separating **design** from implementations

The yellow skills are more advanced,
but the blue ones are more often used.

Key Concepts in Chapter 9

- Algebraic design, algebra (type, operators, and laws)
- Full abstraction of a type
- Type constructor
- Higher-kind, higher-kinded polymorphism
- Structure-preserving map (the structure preservation law)
- Internal DSL

All of these are well hidden in the chapter (some not named explicitly), so make sure you identify them after class.

Input data in JSON format

(this is an example in concrete syntax of JSON; basically a character string)

```
{  
  "Company name" : "Microsoft",  
  "Ticker" : "MSFT",  
  
  "Active" : true,  
  "Price" : 30.66,  
  "Shares outstanding" : 8.38e9,  
  "Related companies" :  
    [ "HPQ", "IBM", "YHOO", "DELL", "GOOG", ],  
}
```

The Example in JSON's Abstract Syntax

(no longer a string, but a structured Scala object)

```
val json = JObject(Map(  
    "Shares outstanding" -> JNumber(8.38E9),  
    "Price" -> JNumber(30.66),  
    "Company name" -> JString("Microsoft"),  
    "Related companies" -> JArray(  
        Vector(JString("HPQ"), JString("IBM"),  
               JString("YHOO"), JString("DELL"),  
               JString("GOOG"))),  
    "Ticker" -> JString("MSFT"),  
    "Active" -> JBool(true)))
```

Abstract Syntax for JSON

(the types of what we want to obtain from the input, using a parser)

```
enum JSON
```

```
  case JNull
```

```
  case JNumber(get: Double)
```

```
  case JString(get: String)
```

```
  case JBool(get: Boolean)
```

```
  case JArray(get: IndexedSeq[JSON] )
```

```
  case JObject(get: Map[String, JSON] )
```

Menti!

Agenda

1. Running Example: parsing JSON
2. Design patterns, concepts, and principles
3. Parsing JSON (review of the combinators)
4. Implementing a concrete parser
5. Parsing libraries in programming languages

Algebraic Design

- **Algebraic design:** *design your interface first, along with associated laws. Use the types and laws to refactor and evolve the interface.*
- We are using types heavily, **designing the API with types**, compiling and trying expressions.
- Since **laws are properties**, they are **tests** (property tests). This is a form of test-driven development (TDD), or test-first development.

Algebraic Design, Full Abstraction, Higher Kinds

(the API/Interface first; separation of design & Implementation)

```
trait Parsers[ParseError, Parser[+_]]:  
  def char(c: Char): Parser[Char]  
  def string(c: String): Parser[String]  
  
extension [A](p: Parser[A])  
  def run(input: String): Either[ParseError, A]  
  def or(p2: Parser[A]): Parser[A]
```

This is a higher kind (a type that is polymorphic in type constructors not in types!)

This is a type (variable)

These types are **abstracted fully**;
We work without deciding how they are implemented.
We only typecheck & compile!

This is a type constructor (variable).
This particular variable must be instantiated with a covariant type constructor.

Menti x 3!

Algebraic Design

(laws, aka tests)

```
forAll { (c: Char) => char(c).run(c.toString) == Right(c) }
```

```
forAll { (s: String) => string(s).run(s) == Right(s) }
```

```
forAll { (s1: String, s2: String) =>
  val p = string(s1).or(string(s2))
  p.run(s1) == Right(s1) && p.run(s2) == Right(s2)
}
```

...

You can make such tests compile, before you have the implementation of parsers!

Map is structure preserving

Consider two new combinators:

extension [A] (p: Parser[A])

```
def many: Parser[List[A]]           // Kleene-star
```

```
def map[B](f: A => B): Parser[B]
```

Example:

```
char('a').many.map { _.size } ← What does this parser produce (menti) ?
```

Law:

```
p.map(a => a) == p // for any parser p
```

This means that map is **structure preserving**

(it only changes values produced, so with identity there is no change at all).

Agenda

1. Running Example: parsing JSON
2. Design patterns, concepts, and principles
3. Parsing JSON (review of the combinators)
4. Implementing a concrete parser
5. Parsing libraries in programming languages

Parsing Combinators: TOKENS for JSON

(We build a parser combinator language in which we can specify the translation)

```
val QUOTED: Parser[String] =
  regex("""([^\"]*)""".r)
    .map { _.dropRight(1).substring(1) }
```

```
val DOUBLE: Parser[Double] =
  regex("""(\+|-)?[0-9]+(\.[0-9]+((e|E)(-|\+)?[0-9]+)?)?""".r)
    .map { _.toDouble }
```

```
val ws: Parser[Unit] =
  regex("""\s+""".r).map { _ => () }
```

Parsing JSON start symbol

```
lazy val json : Parser[JSON] =  
  ws.? /* { jstring | jobject | jarray |  
    jnull | jnumber | jbool }
```

- `|` is choice, delegates to ‘or’
- `?` means optional, also known as ‘opt’
- `* |` is sequencing & drop the right side when building AST
 - (`'x * | y'` is a syntactic sugar for `'(x ** y).map { _.1 }'`)
 - (`'x | * y'` is a syntactic sugar for `'(x ** y).map { _.2 }'`)
- Laziness allows recursive rules (like in EBNF)

Turn terminals into AST leaves

```
val jnull: Parser[JSON] =  
  string("null") |* ws.? |* succeed(JNull)  
  
val jbool: Parser[JBool] =  
  (string("true") |* ws.? |* succeed(JBool(true))) |  
  (string("false") |* ws.? |* succeed(JBool(false)))  
  
val jstring: Parser[JString] =  
  { QUOTED *| ws.? }.map { JString(_) }  
  
val jnumber: Parser[JNumber] =  
  { DOUBLE *| ws.? }.map { JNumber(_) }
```

Parse complex values

[simplified to fit on a slide]

```
lazy val jarray: Parser[JArray] =  
{ char('[') |* ws.?  
|* (json *| char(',') |* ws.? ).*  
*| "]" *| ws.? }.map { l => JArray(l.toVector) }
```

```
lazy val field: Parser[(String, JSON)] =  
QUOTED *| ws.? *| char(':') *| ws.? ** json *| char(',') *| ws.?
```

```
lazy val jobject: Parser[JObject] =  
{ char('{') |* ws.? |* field.* *| char('}') *| ws.? }  
.map { l => JObject(l.toMap) }
```

Parser Combinators

(AKA PEGs = Program Expression Grammars)

- Good for ad hoc jobs, parsing when regexes does not suffice
- Very lightweight as a dependency, no change to build process
- More expressive than generator-based tools (Turing complete)
- In standard libraries of many modern languages
- Error reporting weaker during parsing (but fpinscala does a good job)
- Usually slower than generated parsers (and use more memory), unless implemented at compile time (but parboiled2!)
- Typically no support for debugging grammars

Internal Domain Specific Languages

(Parser Combinators are one example)

- Parser Combinators are a language (loosely similar to EBNF)
- Slogan: internal DSL is syntactic sugar of host language
- No external tools, pure Scala (or another host), no magic involved

Let's analyze an expression

```
QUOTED *| char(' :) ** json *| char(' ,') // parser producing a field
```

earlier:

```
QUOTED : Parser[String] // parser producing a String
```

but

```
extension (p: Parser[A]) def *|(p2: Parser[Any]): Parser[A]
```

so

```
ext1(p=QUOTED).*| (char(' :) ): Parser[String]
```

and also we have (an alias for product)

```
extension (p: Parser[A]) def **[B](p2: Parser[B]): Parser[(A, B)]
```

so

```
ext2(p=ext1(p=QUOTED).*| char(' :) ).product(json): Parser[(String, JSON)]
```

and the extension with *| already used above gives:

```
ext1(p=ext2(p=ext1(p=QUOTED).*| char(' :) ).product(json)).*|(char(' .'))  
: Parser[(String, JSON)]
```

What did we use to build this DSL

- Polymorphic types (that check syntax of our programs), for instance:
`extension [A] (p: Parser[A]) *| : Parser[B] => Parser[A]`
- Function values: `type Parser[+A] = ParseState => Result[A]`
- We could've used automatic conversions (from string, char, regex)
- Calls to unary methods without period (infix ops are methods of ParserOps)
- `string(":") ** json` is really `string(":").**(json)`
(which delegates to the right extension)
- Math symbols as names, eg: ?, |, *|, *, etc
(btw. Scala allows unicode identifiers, used in scalaz/cats internal DSLs)
- No parentheses on nullary methods
`ws.?` translates to `ws.?()` (which delegates to `ws.opt()`)
- Used Scala's parentheses, braces, etc as elements of our DSL

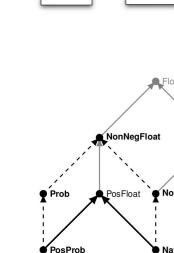
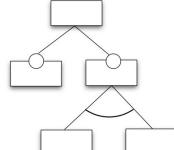
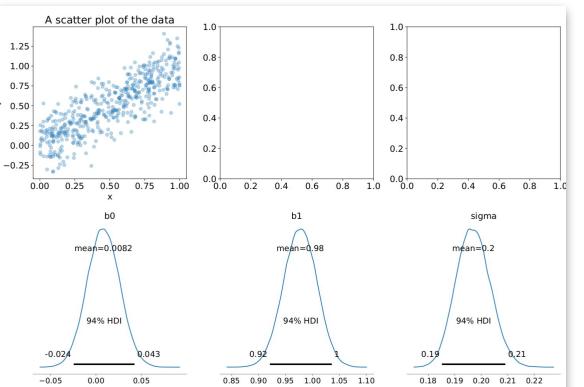
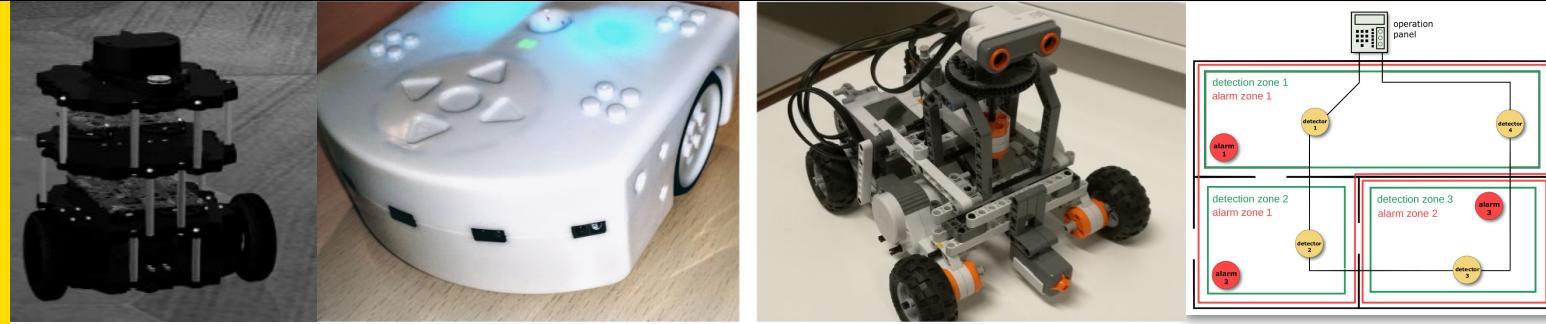


Domain-Specific Languages

Effective Modeling, Automation, and Reuse



Andrzej Wąsowski
Thorsten Berger



$$(\text{CONST}) \frac{c \text{ is a literal}}{\Gamma, \Delta \vdash c : \text{type-of}(c)}$$

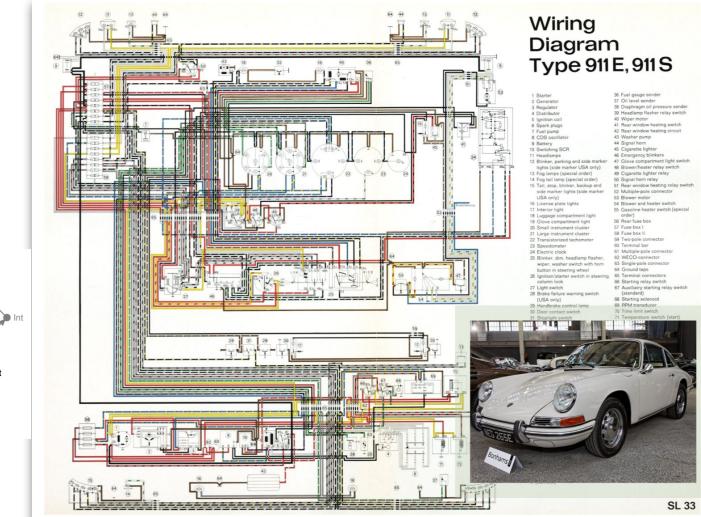
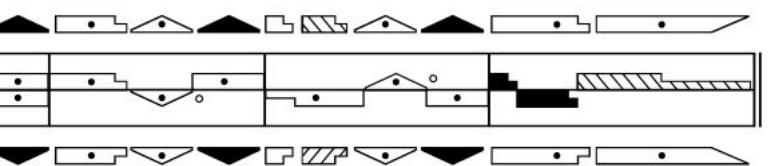
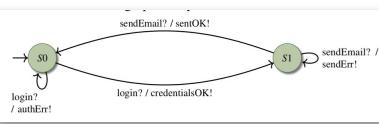
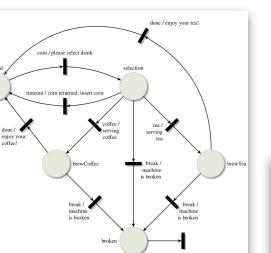
$$\forall m. \forall s. \text{initial}(m, s) \rightarrow \text{states}(m, s)$$

$$(\text{SSIMPLE}) \frac{t_1, t_2 \text{ simple} \quad t_1 \sqsubseteq^* t_2}{t_1 \leqslant t_2}$$

model → 'state' 'machine' String state* 'end'

`e → ('state' | 'initial') String transition*`

transition → 'input' String ('output' String)? 'target' String



Class	Source file	Lines	Methods	Statements	Invoked	Coverage	Branches	Invoked	Coverage	
Constraints	Constraints.scala	156	4	111	79	<div style="width: 71.17%;"><div style="width: 71.17%;"></div></div> 71.17 %	8	2	<div style="width: 25.00%;"><div style="width: 25.00%;"></div></div> 25.00 %	
FanParser	FanParser.scala	94	1	15	15	<div style="width: 100.00%;"><div style="width: 100.00%;"></div></div> 100.00 %	0	0	<div style="width: 100.00%;"><div style="width: 100.00%;"></div></div> 100.00 %	
Interpreter	Interpreter.scala	31	2	19	0	<div style="width: 0.00%;"><div style="width: 0.00%;"></div></div> 0.00 %	2	0	<div style="width: 0.00%;"><div style="width: 0.00%;"></div></div> 0.00 %	
Man	Main.scala	39	1	19	0	<div style="width: 0.00%;"><div style="width: 0.00%;"></div></div> 0.00 %	2	0	<div style="width: 0.00%;"><div style="width: 0.00%;"></div></div> 0.00 %	

```

12 // C1: All machines must have distinct names
13 val C1 = inv[Model] { M =>
14   M.getMachines.asScala.forall { m1 =>
15     M.getMachines.asScala.forall { m2 =>
16       m1 != m2 implies m1.getName != m2.getName
17     }
18   }
19 }
20
21 // C2: all states within the same machine must have distinct names
22 // Vm1 Vs1 Vn1 Vn2.
23 // ... s1 = s2  $\wedge$  states(m, s1)  $\wedge$  states(m, s2)  $\wedge$  name(s1, n1)  $\wedge$  name(s2, n2)  $\rightarrow$  n1 = n2
24 val C2 = inv[FiniteStateMachine] { m =>
25   m.getStates.asScala.forall { s1 =>
26     m.getStates.asScala.forall { s2 =>
27       s1 = s2 implies s1.getName != s2.getName
28     }
29   }
30 }
31
32 // Even shorter formulations of C2 using mdsebook.scala and wildcards
33
34 val C2a = inv[FiniteStateMachine] {
35   ..._getStates.asScala.forall { (s1: State, s2: State) =>
36     s1 = s2 implies s1.getName != s2.getName
37   }
38 }
39
40 val C2b = inv[FiniteStateMachine] {
41   ..._getStates.asScala.forall { _ != _Different {_.getName} }
42 }
43
44 // My favourite formulation of C2 in Scala
45
46 val C2c = ..._getStates.asScala.forall { _ != _Different {_.getName} }
```

Agenda

1. Running Example: parsing JSON
2. Design patterns, concepts, and principles
3. Parsing JSON (review of the combinators)
4. Implementing a concrete parser (somewhat less important)
5. Parsing libraries in programming languages

Running the parser

- We need to implement a Parsers.run method

```
extension [A] (p: Parser[A])  
  def run(input: String): Either[ParseError, A]
```

- Then we call a parser as follows

```
(string("abra") | string("cadabra")).run("abra")
```

```
(string("abra") | string("cada")).run("abra") == Right ("abra")
```

```
(string("abra") | string("cada")).run("Xbra") == Left  
(ParseError(...))
```

Implementing a simple run

```
type Parser[+A] = Location => Result[A]

extension [A](p: Parser[A])
  def run (input: String): Either[ParseError, A] =
    p (Location(input, 0)) match
      case Success(a, n) => Right(a)
      case Failure(err, _) => Left(err)
```

Implementing a concrete parser

(simplified slightly for presentation, exercises use a more advanced representation)

```
def string(s: String): Parser[String] = loc =>
  if loc.curr.startsWith(s) then
    Success(s, s.size)
  else
    val seen = loc.curr
      .substring (0, min(loc.curr.size, s.size))
    Failure(s"expected '$s' but seen '$seen'")
```

Implementing an operator/combinator

(slightly simplified for presentation, flatMap strikes back)

```
extension [A](p: Parser[A])
def flatMap[B](f: A => Parser[B]): Parser[B] = loc =>
  p(loc) match
    case Success(a, n) => f(a)(loc.advanceBy(n))
    case e @ Failure(_, _) => e
```

Implementing an operator/combinator

(slightly simplified for presentation, a more complex variant used in the exercises)

```
extension [A](p: Parser[A])
  def or(p2: => Parser[A]): Parser[A] =
    loc => p(loc) match
      case Failure(_) => p2(loc)
      case r => r
```

Agenda

1. Running Example: parsing JSON
2. Design patterns, concepts, and principles
3. Parsing JSON (review of the combinators)
4. Implementing a concrete parser (somewhat less important)
5. Parsing libraries in programming languages

Parsing Libraries

Java

Parser Generators ANTLR, JavaCC, Rats!, APG, ...

Parser Combinators Parboiled, PetitParser

Scala

Parser Generators ? (parboiled2)

Parser Combinators Scala parser combinators (previously Scalalib), parboiled2 (technically also a generator), fastparse

JavaScript

Parser Generators ANTLR, Jison

Parser Combinators Bennu, Parjs, and Parsimmon

C#

Parser Generators ANTLR, APG

Parser Combinators Pidgin, superpower, parseq

C++

Parser Generators ANTLR, APG, boost meta-parse (?) , boost spirit (?)

Parser Combinators Cpp-peglib, pcomb, boost meta-parse, boost spirit, Parser-Combinators

Conclusion

(what you need to get from this week)

- Algebraic design, algebra (type, operators, and laws)
- Full abstraction of a type
- Type constructor
- Higher-kind, higher-kinded polymorphism
- Structure-preserving map (the structure preservation law)
- Internal DSL, fluent interface
- ... and parser combinators ☺