

# Parser combinators

Georges Dubus

@georgesdubus

DIL

Avec des vrais morceaux de  
*functors, applicative functors et monads*

Mais pas besoin de comprendre

# Objectifs

- Intro aux Parser Combinators
- Utilise *scala-parser-combinators*, mais pas spécifique
- Aller-retours explications/code
- Des ponts avec des concepts de théorie des catégories

```
git clone https://github.com/madjar/talk-scala-parser-combinators
```

# So, what is a parser ?

```
abstract class Parser[+T] extends (Input => ParseResult[T])
```

```
type Input = Reader[Elem]
```

Grosso modo, une liste



Au choix



# So, what is a parser ?

```
abstract class Parser[+T] extends (String => ParseResult[T])
```

# Okay, so what is a parse result?

```
sealed abstract class ParseResult[+T]
```

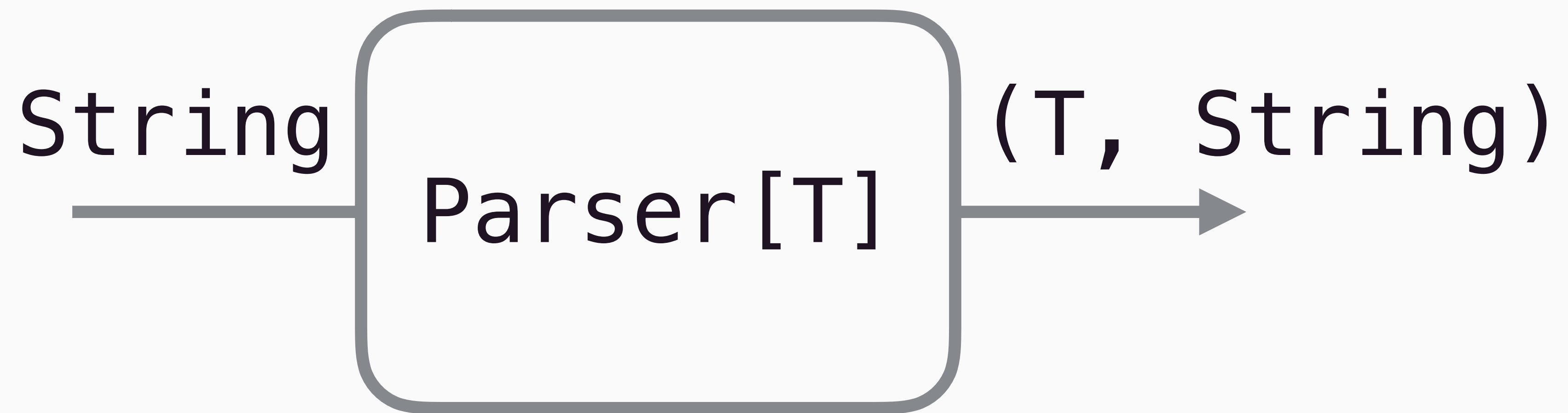
```
case class Success[+T](result: T, override val next: Input)  
  extends ParseResult[T]
```

```
case class Failure(val msg: String, override val next: Input)  
  extends ParseResult[T]
```

# When you remove the cruft

```
Parser[T] = String => Result[T]
```

```
Result[T] = Either[ ErrorMessage  
                   , (T, String) ]
```





*Parser*[*T*] est un parseur qui renvoie un *T*

*T* est un paramètre de type

*Parser* est un type paramétrique

# My first parser

```
object Hello extends Parsers {  
  type Elem = Char  
  def oneA = elem('a')  
}
```

```
>>> val result = Hello.oneA(new CharSequenceReader("axa"))  
>>> result.get  
'a'  
>>> result.next  
"xa"
```

# My first parser (less tedious version)

```
object HelloRegex extends RegexParsers {  
  def oneA = "a"  
}
```

```
>>> val result = HelloRegex.parse(oneA, "axa")  
>>> result.get  
'a'  
>>> result.next  
"xa"
```

# So, how do I get a (primitive) parser?

```
implicit def accept    (e: Elem)      : Parser[Elem]  
implicit def literal  (s: String)    : Parser[String]  
implicit def regex    (r: Regex)     : Parser[String]
```

```
def acceptIf           (p: (Elem) ⇒ Boolean)           : Parser[Elem]  
def acceptMatch[U]    (f: PartialFunction[Elem, U]) : Parser[U]  
def acceptSeq         (es: Iterable[Elem])             : Parser[List[Elem]]
```

# My first combinators

A combinator combines parsers to make parsers

`Parser[T] ~> Parser[U] => Parser[U]`

`Parser[T] <~ Parser[U] => Parser[T]`

```
def inParens = "[" ~> "[a-z]*".r <~ "]"
```

```
inParens("[la la]") == "la la"
```

# Diviser pour mieux régner

Faire des petits parseurs

Les combiner pour en faire des plus gros

# Exercise 1

Hello World => World

Hello Georges => Georges

Goodbye World => FAIL

```
sbt "~testOnly Step1Spec"
```

# Some nice combinators

## Repetition

`rep(Parser[T]) => Parser[List[T]]`

```
rep("l.".r).parse("lalalilo")  
== List("la", "la", "li", "lo")
```

Variantes : `parser.*` (ou `parser*`)



# Some nice combinators

Repetition with a separator

`repsep(Parser[T], Parser[Any]) => Parser[List[T]]`

```
repsep("l.".r, "->").parse("la->la->li->lo")  
== List("la", "la", "li", "lo")
```

# Some nice combinators

Non-empty repetition

`rep1(Parser[T]) => Parser[List[T]]`

`rep1sep(Parser[T], Parser[Any]) => Parser[List[T]]`

# Some nice combinators

Repeat N times

`repN(Int, Parser[T]) => Parser[List[T]]`

# Exercise 2

csv parser !

```
sbt "~testOnly Step2Spec"
```

# And if I don't want a string ?

```
def number = """"\d+""".r
```

```
scala> number("12345")  
res0: String = 12345
```

That is not a number !

# And if I don't want a string ?

Apply a function to a Parser

`Parser[T].map[U](f: (T) => U) => Parser[U]`

# And if I don't want a string ?

```
def number = """"\d+"""".r.map { x => x.toInt }
```

```
def number = """"\d+"""".r ^^ { x => x.toInt }
```

```
scala> number("12345")  
res0: Int = 12345
```

*map* means *Functor*



*map* means *Functor*

Functor[A]  
A => B      => Functor[B]

*map* means *Functor*

$\text{Functor}[A]$   
 $A \Rightarrow B \Rightarrow \text{Functor}[B]$

*Parser* is a *Functor*

# Choosing

# Elevator instructions

[illegible]

```
sealed trait Instruction
object Up extends Instruction
object Down extends Instruction
```

# Choosing

```
def up    = "(" . map(x => Up)
def down = ")"   ^^^ Down
```

```
scala> up.parse("(")
res0: Instruction = Up
```

# Choosing

`Parser[T] | Parser[T] => Parser[T]`

```
def instruction = up | down
```

# Choosing

```
def instruction = up | down
```

```
rep(instruction).parse("(((())())((((())"))  
== Seq(Up, Up, Up, Up, Down, Down, ...)
```

# Let's be recursive

Matching parenthesis

((toto)) should be parsed as toto

(toto should fail

```
def matching: Parser[String] =  
  "[a-z]*" | ("(" ~> matching <~ ")")
```

# Making complex parsers

A date

2015-02-03    =>    LocalDate(2015, 2, 3)



# Making complex parsers

`Parser[T].tuple(Parser[U])`       $\Rightarrow$       `Parser[(T, U)]`

Warning: does not exist

# Making complex parsers

```
"""\d{4}""".r.tuple("""\d{2}""".r.tuple("""\d{2}""".r))  
  .map { case (y, (m, d)) => new LocalDate(y, m, d) }
```

Warning: does not exist

# *Applicative Functor*

*Applicative functors are functors for which there is also a natural transformation that preserve the Cartesian product*

# *Applicative Functor*, translated

*Applicative functor*

=

*functor*

+

some function that merges values in the functor

$(\text{Functor}[A], \text{Functor}[B]) \Rightarrow \text{Functor}[(A, B)]$

# Some sugar on top

$(y, (m, d))$  is ugly

Let's make a type  $\sim [T, U]$  with values  $t \sim u$

Now we can write  $y \sim m \sim d$ , and pattern match on it

# Some sugar on top

$\text{Parser}[T] \sim \text{Parser}[U] \quad \Rightarrow \quad \text{Parser}[\sim[T, U]]$

# Some sugar on top

```
""""\d{4}"""" . r ~ """"\d{2}"""" . r ~ """"\d{2}"""" . r  
^^ { case y ~ m ~ d => new LocalDate(y, m, d) }
```

This is map

# Exercise 3

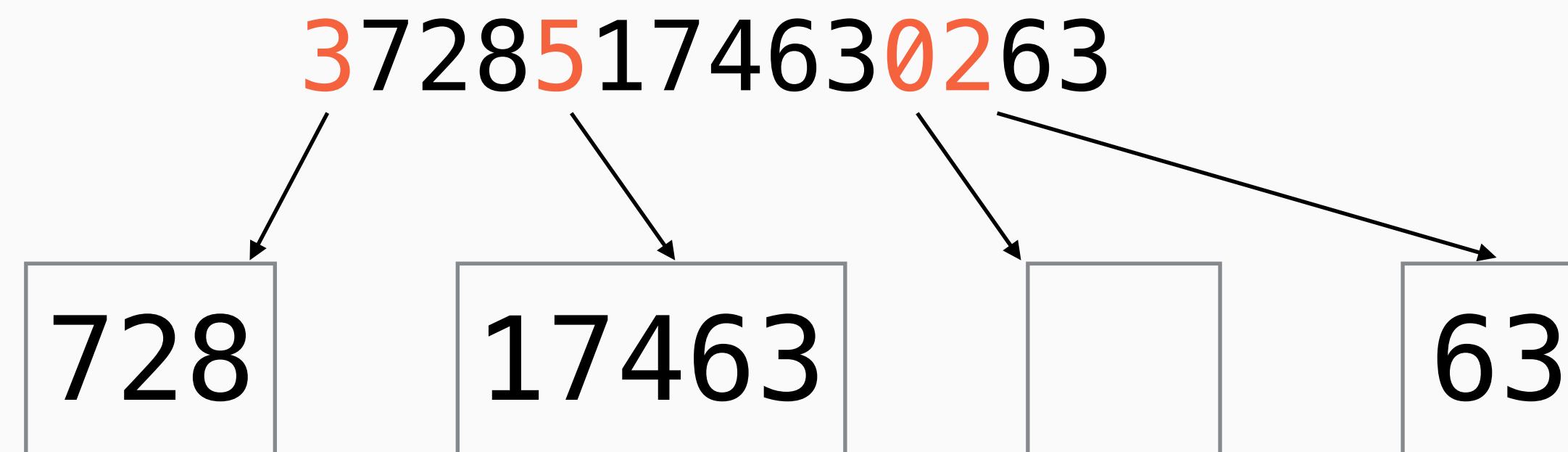
Expression parser !

```
sbt "~testOnly Step3Spec"
```



# Parsers depending on previous parsers

Format alacon : un chiffre annonce le nombre de chiffres à lire



# Parsers depending on previous parsers

`Parser[T].into(T => Parser[U]) => Parser[U]`

# Parsers depending on previous parsers

```
def digit = """"\d"""".r ^^ (_.toInt)
def cell = digit.into { n => repN(n, digit) }
def cells = rep(cell)
```

```
    parse(cells, "37285174630263")
== List(List(7, 2, 8),
        List(1, 7, 4, 6, 3),
        List(),
        List(6, 3))
```

# Parsers depending on previous parsers

Variantes

*digit*.into { n => repN(n, *digit*) }

*digit* >> { n => repN(n, *digit*) }

*digit*.flatMap { n => repN(n, *digit*) }

# *Monads*

# *Monads*

*Functors* that you can chain

$$\begin{aligned} \text{Monad}[A] + A &\Rightarrow \text{Monad}[B] \\ &= \\ \text{Monad}[B] \end{aligned}$$

# Syntactic sugar!

```
digit.into { n => repN(n, digit) }
```

```
for { n <- digit  
      result <- repN(n, digit)  
} yield result
```

# Syntactic sugar!

```
def product = "(" ~> (expr <~ "*" ) ~ expr <~ ")"  
^^ { case l ~ r => Product(l, r) }
```

```
for { _ <- "("  
      l <- expr  
      _ <- "*"   
      r <- expr  
      _ <- ")"  
} yield Product(l, r)
```



Final exercise

JSON!

```
sbt "~testOnly Step4Spec"
```

# Pour finir

- Perfs
- <https://github.com/madjar/talk-scala-parser-combinators>
- Questions ?