Grok: optimistic low-level parsing in Scala

Rex Kerr · UC San Francisco
Outline

1. Why "easy" parsing is not easy (psst, it's the errors)
Part One
Parsing is Hard (?!)

Background

• I am a neuroscientist (studying aging in simple model organisms)
A typical day in research

• I wonder if older worms have defects in sequencing behavior?
• Let's compute transition probabilities for behaviors
• Let's read behavioral data
• Okay, parse the data, analyze...
• (wait, wait, wait)
• Stack trace? From the parser?! !*&%(*@&!%
• That's not supposed to be the hard part!!
Why is parsing *deceptively* hard?

It's easy:

• Input is complex  <- Want focus here!!
• But it's like language, which we humans are specialized for!
• And we're good at explaining what we want, what the rules are.
• *We can do this!*

But it's not!

• Any rule can go wrong and needs handling.
• We are (Rex is) *not* so good at intuiting that.
• Need to focus on errors.  <- Argh!
An example: head and last of a list.

```python
def ht[A](xs: List[A]) = (xs.head, xs.last)
```

Easy and wrong.

```python
def ht[A](xs: List[A]) = {
    for (h <- xs.headOption;
         t <- xs.lastOption)
    yield (h,t)
}
```

Correct but not so easy.
Parsing is like this, but a hundredfold worse.

What can we do?
Part Two
Crafting a solution to the parsing problems
What do we want when parsing?

- Focus on transformation, not on errors
- Capture errors for rapid debugging / resolution
- Reasonably fast: rare to need hand-rolling
- Make it hard to make a mistake
Testbed: the world's simplest* language

<item> := "y" | "n"

* Simplest non-degenerate language. The empty language is simpler.
The Java Way: what goes wrong?

```java
def parseEx(c: Char) =
    if (c == 'y') true
    else if (c == 'n') false
    else throw new IllegalArgumentException("Only y and n are allowed")
```

Unfortunately...

```java
def countEx(s: String): Int = {
    var n, i = 0
    while (i < s.length) {
        n += (if (parseEx(s charAt i)) 2 else -1)
        i += 1
    }
    n
}
```

Did we forget to catch the exception? Will anyone ever remember?
Failure cases are really, really slow!
The Monadic Way: what goes wrong?

```python
def parseOpt(c: Char) =
    if (c == 'y') Some(true)
    else if (c == 'n') Some(false)
    else None
```

Unfortunately...

```python
def countOpt(s: String): Int = {
    var n, i = 0
    while (i < s.length) {
        parseOpt(s charAt i) match {
            case Some(x) => n += (if (x) 2 else -1)
            case _ =>
        }
        i += 1
    }
    n
}
```

Boilerplate grows *linearly with number of options!*
The Monadic Way: what goes wrong?

Success cases are slow!
Can we rescue the Monadic Way?

class Wiggle(src: Dance, id: Int, i0: Int, iN: Int, dist: Double, phase: Option[Double])

What we want:

new Wiggle(parseDance, parseInt, parseInt, parseInt, parseDouble, parseDoubleOption)

What we've got:

for {
    src <- parseDance; id: <- parseInt;
    i0: <- parseInt; iN: <- parseInt; dist: <- parseDouble
} yield new Wiggle(src, id, i0, iN, dist, parseDouble)

• Detailed option/error types infect entire call stack
• Loads of nuisance variables
Can we rescue the Monadic Way?

Maybe we can abstract away the boilerplate?

\[(A, B) \Rightarrow Z \Rightarrow (E[A], E[B]) \Rightarrow E[Z]\]

This is *possible*, but we must write builders:

```javascript
object Foo {
    def apply[A,B](a: A, b: B) = new Foo(a, b)
}
```

And we must abstract over arity:

\[(A, B, C) \Rightarrow Z \Rightarrow (E[A], E[B], E[C]) \Rightarrow E[Z]\]
\[(A, B, C, D) \Rightarrow Z \Rightarrow (E[A], E[B], E[C], E[D]) \Rightarrow E[Z]\]
\[(A, B, C, D, F) \Rightarrow Z \Rightarrow (E[A], E[B], E[C], E[D], E[F]) \Rightarrow E[Z]\]
Can we rescue the Monadic Way?

And (at least with implicit conversions) over certainty:

\[
\begin{align*}
((A, B, C) \Rightarrow Z) &\Rightarrow ((E[A], E[B], C) \Rightarrow E[Z]) \\
((A, B, C) \Rightarrow Z) &\Rightarrow ((E[A], B, E[C]) \Rightarrow E[Z]) \\
((A, B, C) \Rightarrow Z) &\Rightarrow ((A, E[B], E[C]) \Rightarrow E[Z])
\end{align*}
\]

And then finally you can:

\[
\text{magic(Foo.apply _)(parseDance, parseInt, parseInt, parseInt, parseDouble, parseDouble)}
\]

And after all that work you still have sizable performance penalties.
Let's back up.

What ways do we have to deal with error conditions?

• Return values (incl. monads).
  Slow on success, boilerplate, error logic mixed in everywhere.

• Exceptions.
  Slow on failure, can escape, records code context not data context.

• Error codes.
  You will forget to check the error code. Seriously, you will forget.

• Goto.
  Just...no.
Well, okay. What about error codes?

They're fast, just like we thought. But we will forget to check them.
What about goto?

- C has a frightfully dangerous `setjmp/longjmp` library call.
- C++ has exceptions.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception setup</td>
<td>0</td>
</tr>
<tr>
<td>Exception invoked</td>
<td>6800</td>
</tr>
<tr>
<td><code>setjmp</code> (setup)</td>
<td>18</td>
</tr>
<tr>
<td><code>longjmp</code> (invoked)</td>
<td>47</td>
</tr>
</tbody>
</table>

But the JVM doesn't have `setjmp/longjmp`. *Whew!*
How close can we get?

**Stackless exceptions**

- Worse than exceptions: *no idea* where an uncaught one came from.

```scala
import scala.util.control.Breaks._
def test() = {
  var n = 0
  for (i <- Iterator.from(1)) {
    if (n > 10) break
    n += i
  }
  println(n)
}
test()
```

What is the output in the REPL?

scala.util.control.BreakControl

*sigh*

- But compiler already inserts these to `return` from inside closures, and you don't notice.
How close can we get?

Stackless exceptions

Wow! Fast in all use cases!

Note: this is only because JVM can inline and convert to a jump. If it actually has to be thrown, it's still 10x faster than a real exception.
Now we have a glimmer of a solution.

- Internal to library: use error codes (fastest)
- External API: use stackless exceptions
- *How do we make them safe?*
- *How do we manage mutability?*
Manage safety with the \textit{key pattern}.

```scala
class Foo {
  def safeBar = ???
  def supervise[A](f: Key[this.type] => A): A = ???
  def unsafeBaz(implicit unlock: Key[this.type]) = ???
}
```

```scala
def useUnsafeFoo(foo: Foo)(implicit unlock: Key[foo.type]) = ???
```

- Just like checked exceptions: you must handle them.
- Inversion of control allows precise requirements.

```scala
val foo, other = new Foo
foo.supervise{ implicit key =>
  useUnsafeFoo(foo) // Works
  foo.unsafeBaz // Works
  otherFoo.unsafeBaz // Compile-time error, no key!
}
```
How does the CPU manage mutability?

PUSHA
CALL sub
POPA

So if we like our mutable state:

class WillMutate {
  var state: State
  def saveOnStack[A](f: => A): A = {
    val temp = state
    try { f } finally { state = temp }
  }
}

val m = new WillMutate
... // I like my state! But I need to change it for a bit.
  m.saveOnStack {
    m.mutate  // Just use the same m!
  }
  // Now my old state is back
Part Three
Grokking: simple, low-level, fast, optimistic
Grok is like an Iterator for parseable data. This is not new. Everything from PrintReader to IntBuffer does this. **But Grok lets you be optimistic and just grab what you want.**
def parseJany(g: Grok)(implicit fail: GrokHop[g.type]): Any = g.peek match {
  case '{' => parseJob(g)
  case '[' => parseJarr(g)
  case '\' => g.quoted
  case c if (c >= '0' && c <= '9') || c == '-' => g.D
  case 't' | 'f' => g.Z
  case _ => fail(g.customError)
}
def parseJob(g: Grok)(implicit fail: GrokHop[g.type]): Map[String, Any] = {
  val m = new AnyRefMap[String, Any]
  g exact '{'
  while (g.peek != '}' ) {
    if (!m.isEmpty) g exact ' ,'
    val key = g.quoted
    g exact ': '
    m += (key, parseJany(g))
  }
  g exact '}'
  m
}
**Grokking JSON**

### Specification

```
array
```

### Implementation

```scala
def parseJarr(g: Grok)(implicit fail: GrokHop[g.type]): Array[Any] = {
  val ab = Array.newBuilder[Any]
  var index = 0
  g exact '['
  while (g.peek != ']') {
    if (index > 0) g exact ','
    ab += parseJany(g)
    index += 1
  }
  g exact ']
  ab.result
}
```
Grokking JSON
Grokking JSON

Specification

Implementation (if you're happy with Double)

g.D
import collection.Map
import collection.mutable.AnyRefMap
import kse.flow._
import kse.eio._

object pJson {
  ...

  def parse(g: Grok): Ok[GrokError, Map[String, Any]] =
    g.delimit(0){ implicit fail => parseJob(g.trim) }

  def apply(ab: Array[Byte]) = parse(Grok text ab)
  def apply(s: String) = parse(Grok(s))
}
Yes, but anyone can parse JSON.

Of course! Parboiled2, for instance.

```python
def Json = rule { WhiteSpace ~ Value ~ EOI }

def JsonObject: Rule1[JsObject] = rule {
    ws('{}') ~ zeroOrMore(Pair).separatedBy(ws(',')) ~ ws('{}') ~> 
    ((fields: Seq[JsField]) => JsObject(fields : *))
}

def Pair = rule { JsonStringUnwrapped ~ ws(':') ~ Value ~> (_, _) }

def JsonArray = rule {
    ws('[') ~ zeroOrMore(Value).separatedBy(ws(',')) ~ ws(']') ~> 
    (JsArray(_ : *))
}

def Value: Rule1[JsValue] = rule {
    run {
        (cursorChar: @switch) match {
            case '"' => JsonString
            case '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'|'-' ~> JsonNumber
            case '{' => JsonObject
            ...
        }
    }
}

(From Parboiled2 examples directory.)
How fast is it?

Setting expectations.

Parser combinators

<table>
<thead>
<tr>
<th>Speed (MB/s)</th>
<th>&quot;a&quot;: &quot;b&quot;</th>
<th>[1, 2, 3]</th>
<th>0.50</th>
<th>0.75</th>
</tr>
</thead>
</table>

Jackson

| Speed (MB/s) | "a": "b" | [1, 2, 3] | 236  | 53   |
How fast is it?

Parser combinators

<table>
<thead>
<tr>
<th>Speed (MB/s)</th>
<th>&quot;a&quot;: &quot;b&quot;</th>
<th>[1, 2, 3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Par. Comb.</td>
<td>0.50</td>
<td>0.75</td>
</tr>
<tr>
<td>Jackson</td>
<td>236</td>
<td>53</td>
</tr>
<tr>
<td>Grok</td>
<td>220</td>
<td>89</td>
</tr>
</tbody>
</table>
Wait, what??

From parboiled2 mailing list thread, "Quick JSON parsing benchmark"

<table>
<thead>
<tr>
<th>benchmark</th>
<th>ms</th>
<th>linear runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParserCombinators</td>
<td>2931.64</td>
<td></td>
</tr>
<tr>
<td>Parboiled1JsonParser</td>
<td>117.15</td>
<td></td>
</tr>
<tr>
<td>Parboiled2JsonParser</td>
<td>12.79</td>
<td></td>
</tr>
<tr>
<td>Json4SNative</td>
<td>8.62</td>
<td></td>
</tr>
<tr>
<td>Argonaut</td>
<td>7.06</td>
<td></td>
</tr>
<tr>
<td>Json4SJackson</td>
<td>4.19</td>
<td></td>
</tr>
</tbody>
</table>
What's the trick?

• Direct parsing: don't talk about it *do it*
• Custom Double-parsing code
What about errors?
Adding context that matters

def parseJarr(g: Grok)(implicit fail: GrokHop[g.type]): Array[Any] = {
    val ab = Array.newBuilder[Array]
    var index = 0

    g.context("index " + index) {
        g exact '['
        while (g.peek != ']') {
            if (index > 0) g exact ','
            ab += parseJany(g)
            index += 1
        }
        g exact ']'
    }
    ab.result
}
Adding context that matters

Error in subset of data: improper format
  token 1, position 0 :
  parsing key web-app
  in subset of data: improper format
    token 4, position 12 :
      parsing key servlet
    in subset of data: improper format
      token 7, position 27 :
        parsing index 0
      in subset of data: improper format
        token 8, position 36 :
          parsing key init-param
        in subset of data: improper format
          token 19, position 143 :
            parsing key configGlossary:installationAt
          in expected string: improper format
            token 21, position 184 : installationAt"; "Philadelphia
Conclusions

• Non-local error handling is simple
• Non-local error handling can be fast
• With Grok, parsing is only as hard as it should be
• Grok is not ready to use. It's alpha, full of bugs. But it's getting close.
• For more: ichoran@gmail.com
• And I never tweet anything, but follow @_ichoran_ and I will announce when Grok is really ready.

Thanks for listening!