

Stop overusing regular expressions!

Franklin Chen

<http://franklinchen.com/>

Pittsburgh Tech Fest 2013

June 1, 2013

Famous quote

In 1997, **Jamie Zawinski** famously wrote:

*Some people, when confronted with a problem, think,
“I know, I’ll use regular expressions.”
Now they have two problems.*

Purpose of this talk

Assumption: you already have experience using regexes

Goals:

- ▶ Change how you think about and use regexes
- ▶ Introduce you to advantages of using parser combinators
- ▶ Show a smooth way to transition from regexes to parsers
- ▶ Discuss *practical* tradeoffs
- ▶ Show only tested, running code:
[https://github.com/franklinchen/
talk-on-overusing-regular-expressions](https://github.com/franklinchen/talk-on-overusing-regular-expressions)

Non-goals:

- ▶ Will not discuss computer science theory
- ▶ Will not discuss parser generators such as yacc and **ANTLR**

Example code in which languages?

Considerations:

- ▶ This is a *polyglot* conference
- ▶ Time limitations

Decision: focus primarily on two representative languages.

- ▶ **Ruby**: dynamic typing
 - ▶ Perl, Python, JavaScript, Clojure, etc.
- ▶ **Scala**: static typing:
 - ▶ Java, C++, C#, F#, ML, Haskell, etc.

An infamous regex for email

The reason for my talk!

A big Perl regex for email address based on RFC 822 grammar:

```
/(?: (?: \r\n)? [ \t] ) * (?: (?: (?: [^()<>@,;: \\". \[\] \000-\031] -
)+ | \Z | (?: [= \["()<>@,;: \\". \[\]] ) ) | " (?: [^\" \r\\] | \\ . | (?: (?: \r\n)?
\r\n)? [ \t] ) * ) (?: \. (?: (?: \r\n)? [ \t] ) * (?: [^()<>@,;: \\". \[\]
?: \r\n)? [ \t] ) + | \Z | (?: [= \["()<>@,;: \\". \[\]] ) ) | " (?: [^\" \r\\]
\t] ) ) * " (?: (?: \r\n)? [ \t] ) * ) * @ (?: (?: \r\n)? [ \t] ) * (?: [^()<>
31] + (?: (?: (?: \r\n)? [ \t] ) + | \Z | (?: [= \["()<>@,;: \\". \[\]] ) ) | \
] (?: (?: \r\n)? [ \t] ) * ) (?: \. (?: (?: \r\n)? [ \t] ) * (?: [^()<>@,;: \
(?: (?: (?: \r\n)? [ \t] ) + | \Z | (?: [= \["()<>@,;: \\". \[\]] ) ) | \[ ( [^ \
(?: \r\n)? [ \t] ) * ) * | (?: [^()<>@,;: \\". \[\] \000-\031] + (?: (?: \
| (?: [= \["()<>@,;: \\". \[\]] ) ) | " (?: [^\" \r\\] | \\ . | (?: (?: \r\n)?
[ \t] ) * ) * \< (?: (?: \r\n)? [ \t] ) * (?: @ (?: [^()<>@,;: \\". \[\] \0
\r\n)? [ \t] ) + | \Z | (?: [= \["()<>@,;: \\". \[\]] ) ) | \[ ( [^ \[\] \r\\] | \
\t] ) * ) (?: \. (?: (?: \r\n)? [ \t] ) * (?: [^()<>@,;: \\". \[\] \000-\
[ \t] ) + | \Z | (?: [= \["()<>@,;: \\". \[\]] ) ) | \[ ( [^ \[\] \r\\] | \\ . ) >
) * ) * (?: , @ (?: (?: \r\n)? [ \t] ) * (?: [^()<>@,;: \\". \[\] \000-\03
\t] ) + | \Z | (?: [= \["()<>@,;: \\". \[\]] ) ) | \[ ( [^ \[\] \r\\] | \\ . ) * \]
```

Personal story: my email address fiasco

To track and prevent spam: FranklinChen+spam@cmu.edu

- ▶ Some sites *wrongly* claimed invalid (because of +)
- ▶ Other sites did allow registration
 - ▶ I caught spam
 - ▶ Unsubscribing failed!
 - ▶ Some wrong claimed invalid (!?!)
 - ▶ Some silently failed to unsubscribe
 - ▶ Had to set up spam filter

Problem: different regexes for email?

Examples: which one is better?

```
/\A[^\@]+\@[^\@]+\z/
```

vs.

```
/\A[a-zA-Z]+\@([\^\@\.]+\\.)+[\^\@\.]+\z/
```

Readability: first example

Use x for readability!

```
/\A          # match string begin
  [^@]+      # local part: 1+ chars of not @
  @          # @
  [^@]+      # domain: 1+ chars of not @
  \z/x      # match string end
```

matches

FranklinChen+spam

@

cmu.edu

Advice: please write regexes in this formatted style!

Readability: second example

```
/\A          # match string begin
[a-zA-Z]+   # local part: 1+ of Roman alphabetic
@           # @
(           # 1+ of this group
  [^\@\.]+   # 1+ of not (@ or dot)
  \.         # dot
)+
[^\@\.]+    # 1+ of not (@ or dot)
/z/x       # match string end
```

does *not* match

FranklinChen+spam

@

cmu.edu

Don't Do It Yourself: find libraries

Infamous regex revisited: was automatically generated into the Perl module `Mail::RFC822::Address` based on the RFC 822 spec.

If you use Perl and need to validate email addresses:

```
# Install the library
```

```
$ cpanm Mail::RFC822::Address
```

Other Perl regexes: `Regexp::Common`

Regex match success vs. failure

Parentheses for captured grouping:

```
if ($address =~ /\A          # match string begin
    ([a-zA-Z]+) # $1: local part: 1+ Roman alphabetic
    @          # @
    (          # $2: entire domain
        (?:    # 1+ of this non-capturing group
            [^@\.] + # 1+ of not (@ or dot)
            \.      # dot
        )+
        ([^@\.] +) # $3: 1+ of not (@ or dot)
    )
    \z/x) {      # match string end
    print "local part: $1; domain: $2; top level: $3\n";
}
else {
    print "Regex match failed!\n";
}
```

Regexes do not report errors usefully

Success:

```
$ perl/extract_parts.pl 'prez@whitehouse.gov'  
local part: prez; domain: whitehouse.gov; top level: gov
```

But failure:

```
$ perl/extract_parts.pl 'FranklinChen+spam@cmu.edu'  
Regex match failed!
```

We have a dilemma:

- ▶ Bug in the regex?
- ▶ The data really doesn't match?

Better error reporting: how?

Would prefer something at least minimally like:

```
[1.13] failure: '@' expected but '+' found
```

```
FranklinChen+spam@cmu.edu  
                ^
```

Features of *parser* libraries:

- ▶ Extraction of line, column information of error
- ▶ Automatically generated error explanation
- ▶ Hooks for customizing error explanation
- ▶ Hooks for error recovery

Moving from regexes to grammar parsers

- ▶ Only discuss parser combinators, not generators
- ▶ Use the second email regex as a starting point
- ▶ Use modularization

Modularized regex: Ruby

Ruby allows interpolation of regexes into other regexes:

```
class EmailValidator::PrettyRegexMatcher
  LOCAL_PART = /[a-zA-Z]+/x
  DOMAIN_CHAR = /^[^\.]/x
  SUB_DOMAIN = /#{DOMAIN_CHAR}+/x
  AT = /@/x
  DOT = /\./x
  DOMAIN = /( #{SUB_DOMAIN} #{DOT} )+ #{SUB_DOMAIN}/x
  EMAIL = /\A #{LOCAL_PART} #{AT} #{DOMAIN} \z/x

  def match(s)
    s =~ EMAIL
  end
end
```

Modularized regex: Scala

```
object PrettyRegexMatcher {  
  val user = "[a-zA-Z]+"  
  val domainChar = "[^@\\.]+"  
  val domainSegment = raw"$(domainChar)+"  
  val at = "@"  
  val dot = "\\."  
  val domain = raw"$(domainSegment $dot)+ $domainSegment"  
  val email = raw"$(domain) \\A $user $at $domain \\z"  
  val emailRegex = email.r  
  
  def matches(s: String): Boolean =  
    (emailRegex findFirstMatchIn s).nonEmpty  
end
```

- ▶ Triple quotes are for raw strings (no backslash interpretation)
- ▶ raw allows raw variable interpolation
- ▶ .r is a method converting String to **Regex**

Email parser: Ruby

Ruby does not have a standard parser combinator library. One that is popular is **Parslet**.

```
require 'parslet'

class EmailValidator::Parser < Parslet::Parser
  rule(:local_part) { match['a-zA-Z'].repeat(1) }
  rule(:domain_char) { match['~@\\\\.'] }
  rule(:at)          { str('@') }
  rule(:dot)         { str('.') }
  rule(:sub_domain) { domain_char.repeat(1) }
  rule(:domain) { (sub_domain >> dot).repeat(1) >>
                  sub_domain }
  rule(:email) { local_part >> at >> domain }
end
```

Email parser: Scala

Scala comes with **standard parser combinator library**.

```
import scala.util.parsing.combinator.RegexParsers

object EmailParsers extends RegexParsers {
  override def skipWhitespace = false

  def localPart: Parser[String] = "[a-zA-Z]+"
  def domainChar = "[^@\\.]"
  def at = "@"
  def dot = "."
  def subDomain = rep1(domainChar)
  def domain = rep1(subDomain ~ dot) ~ subDomain
  def email = localPart ~ at ~ domain
}
```

Inheriting from `RegexParsers` allows the implicit conversions from regexes into parsers.

Running and reporting errors: Ruby

```
parser = EmailValidator::Parser.new
begin
  parser.email.parse(address)
  puts "Successfully parsed #{address}"
rescue Parslet::ParseFailed => error
  puts "Parse failed, and here's why:"
  puts error.cause.ascii_tree
end
```

```
$ validate_email 'FranklinChen+spam@cmu.edu'
```

```
Parse failed, and here's why:
```

```
Failed to match sequence (LOCAL_PART AT DOMAIN) at
line 1 char 13.
```

```
'- Expected "@", but got "+" at line 1 char 13.
```

Running and reporting errors: Scala

```
def runParser(address: String) {  
  import EmailParsers._  
  
  // parseAll is method inherited from RegexParsers  
  parseAll(email, address) match {  
    case Success(_, _) =>  
      println(s"Successfully parsed $address")  
    case failure: NoSuccess =>  
      println("Parse failed, and here's why:")  
      println(failure)  
  }  
}
```

Parse failed, and here's why:

[1.13] failure: '@' expected but '+' found

FranklinChen+spam@cmu.edu

Infamous email regex revisited: conversion!

- ▶ Mail::RFC822::Address Perl regex: actually manually back-converted from a parser!
- ▶ Original module used Perl parser library `Parse::RecDescent`
- ▶ Regex author turned the grammar rules into a modularized regex
- ▶ Reason: speed

Perl modularized regex `source code excerpt`:

```
# ...  
my $localpart = "$word(?:\\.$lwsp*$word)*";  
my $sub_domain = "(?:$atom|$domain_literal)";  
my $domain = "$sub_domain(?:\\.$lwsp*$sub_domain)*";  
my $addr_spec = "$localpart\\@$lwsp*$domain";  
# ...
```

Why validate email address anyway?

- ▶ Software development: always look at the bigger picture
- ▶ This guy advocates simply sending a user an activation email
- ▶ Engineering tradeoff: the email sending and receiving programs need to handle the email address anyway

Email example wrapup

- ▶ It is possible to write a regex
- ▶ But first try to use someone else's tested regex
- ▶ If you do write your own regex, modularize it
- ▶ For error reporting, use a parser: convert from modularized regex
- ▶ Do you even need to solve the problem?

More complex parsing: toy JSON

```
{  
  "distance" : 5.6,  
  "address" : {  
    "street" : "0 Nowhere Road",  
    "neighbors" : ["X", "Y"],  
    "garage" : null  
  }  
}
```

Trick question: could you use a regex to parse this?

Recursive regular expressions?

2007: **Perl 5.10** introduced “**recursive regular expressions**” that can parse *context-free grammars* that have rules embedding a reference to themselves.

```
$regex = qr/  
    \(                # match an open paren (  
        (            #   followed by  
            [^()]+    #       1+ non-paren characters  
        |            #   OR  
            (??{$regex}) #       the regex itself  
        )*          #   repeated 0+ times  
    \)                # followed by a close paren )  
/x;
```

- ▶ *Not recommended!*
- ▶ But, gateway to context-free *grammars*

Toy JSON parser: Ruby

To save time:

- ▶ No more Ruby code in this presentation
- ▶ Continuing with Scala since most concise
- ▶ Remember: concepts are language-independent!

Full code for a toy JSON parser is available in the Parslet web site at <https://github.com/kschiess/parslet/blob/master/example/json.rb>

Toy JSON parser: Scala

```
object ToyJSONParsers extends JavaTokenParsers {  
  def value: Parser[Any] = obj | arr |  
    stringLiteral | floatingPointNumber |  
    "null" | "true" | "false"  
  def obj = "{" ~ repsep(member, ",") ~ "}"  
  def arr = "[" ~ repsep(value, ",") ~ "]"  
  def member = stringLiteral ~ ":" ~ value  
}
```

- ▶ Inherit from JavaTokenParsers: reuse stringLiteral and floatingPointNumber parsers
- ▶ ~ is overloaded operator to mean *sequencing* of parsers
- ▶ value parser returns Any (equivalent to Java Object) because we have not yet refined the parser with our own domain model

Fancier toy JSON parser: domain modeling

Want to actually query the data upon parsing, to use.

```
{  
  "distance" : 5.6,  
  "address" : {  
    "street" : "0 Nowhere Road",  
    "neighbors" : ["X", "Y"],  
    "garage" : null  
  }  
}
```

- ▶ We may want to traverse the JSON to address and then to the second neighbor, to check whether it is "Y"
- ▶ Pseudocode after storing in domain model:
data.address.neighbors[1] == "Y"

Domain modeling as objects

```
{  
  "distance" : 5.6,  
  "address" : {  
    "street" : "0 Nowhere Road",  
    "neighbors" : ["X", "Y"],  
    "garage" : null  
  }  
}
```

A natural domain model:

```
JObject(  
  Map(distance -> JFloat(5.6),  
    address -> JObject(  
      Map(street -> JString(0 Nowhere Road),  
        neighbors ->  
          JArray(List(JString(X), JString(Y))),  
        garage -> JNull))))
```

Domain modeling: Scala

```
// sealed means: can't extend outside the source file  
sealed abstract class ToyJSON
```

```
case class JObject(map: Map[String, ToyJSON]) extends  
  ToyJSON
```

```
case class JArray(list: List[ToyJSON]) extends ToyJSON
```

```
case class JString(string: String) extends ToyJSON
```

```
case class JFloat(float: Float) extends ToyJSON
```

```
case object JNull extends ToyJSON
```

```
case class JBoolean(boolean: Boolean) extends ToyJSON
```

Fancier toy JSON parser: Scala

```
def value: Parser[ToyJSON] = obj | arr |  
  stringLiteralStripped ^^ { s => JString(s) } |  
  floatingPointNumber ^^ { s => JFloat(s.toFloat) } |  
  "null" ^^^ JNull | "true" ^^^ JBoolean(true) |  
  "false" ^^^ JBoolean(false)  
  
def stringLiteralStripped: Parser[String] =  
  stringLiteral ^^ { s => s.substring(1, s.length-1) }  
  
def obj: Parser[JObject] = "{" ~> (repsep(member, ",") ^^  
  { aList => JObject(aList.toMap) }) <~ "  
def arr: Parser[JArray] = "[" ~> (repsep(value, ",") ^^  
  { aList => JArray(aList) }) <~ "]"  
  
def member: Parser[(String, ToyJSON)] =  
  ((stringLiteralStripped <~ ":" ) ~ value) ^^  
  { case s ~ v => s -> v }
```

Running the fancy toy JSON parser

```
val j = """{
  "distance" : 5.6,
  "address" : {
    "street" : "0 Nowhere Road",
    "neighbors" : ["X", "Y"],
    "garage" : null
  }
}"""

parseAll(value, j) match {
  case Success(result, _) =>
    println(result)
  case failure: NoSuccess =>
    println("Parse failed, and here's why:")
    println(failure)
}
```

Output: what we proposed when data modeling!

JSON: reminder not to reinvent

- ▶ Use a standard JSON parsing library!
 - ▶ **Scala has one.**
 - ▶ All languages have a standard JSON parsing library.
 - ▶ Shop around: alternate libraries have different tradeoffs.
- ▶ Other standard formats: HTML, XML, CSV, etc.

JSON wrapup

- ▶ Parsing can be simple
- ▶ Domain modeling is trickier but still fit on one page
- ▶ *Use an existing JSON parsing library already!*

Final regex example

- ▶ These slides use the Python library **Pygments** for syntax highlighting of all code (highly recommended)
- ▶ Experienced bugs in the highlighting of regexes in Perl code
- ▶ Found out why, in **source code** for class PerlLexer

#TODO: give this to a perl guy who knows how to parse perl

```
tokens = {
```

```
    'balanced-regex': [
```

```
        (r'/(\\\\\\\\|\\\\[^\n]|[^\\\/])*/[egimosx]*', String.Regex, '#p
```

```
        (r'!(\\\\\\\\\\\\|\\\\[^\n]|[^\\!])*! [egimosx]*', String.Regex, '#p
```

```
        (r'\\(\\\\\\\\\\\\|\\\\[^\n])*\\\\ [egimosx]*', String.Regex, '#pop'),
```

```
        (r'{(\\\\\\\\\\\\|\\\\[^\n]|[^\\}]*)}[egimosx]*', String.Regex, '#p
```

```
        (r'<(\\\\\\\\\\\\|\\\\[^\n]|[^\\>])*>[egimosx]*', String.Regex, '#p
```

```
        (r'\\(\\\\\\\\\\\\|\\\\[^\n]|[^\\\\\\\\])*\ [egimosx]*', String.Regex,
```

```
        (r'\\((\\\\\\\\\\\\|\\\\[^\n]|[^\\\\\\\\\\\\])*\ [egimosx]*', String.Regex,
```

```
        (r'@(\\\\\\\\\\\\|\\\\[^\n]|[^\\\\\\\\@])*@[egimosx]*', String.Regex, '#
```

```
        (r'%(\\\\\\\\\\\\|\\\\[^\n]|[^\\\\\\\\%])*%[egimosx]*', String.Regex, '#
```

```
        (r'\\$(\\\\\\\\\\\\|\\\\[^\n]|[^\\\\\\\\$])*\\$[egimosx]*', String.Regex,
```

Conclusion

- ▶ Regular expressions
 - ▶ No error reporting
 - ▶ Flat data
- ▶ More general grammars
 - ▶ Composable structure (using combinator parsers)
 - ▶ Hierarchical, nested data
- ▶ Avoid reinventing
- ▶ Concepts are language-independent: find and use a parser library for your language

All materials for this talk available at <https://github.com/franklinchen/talk-on-overusing-regular-expressions>.
The hyperlinks on the slide PDFs are clickable.