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`

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

```perl
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:

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

Modularized regex: Scala

```
val user = """[a-zA-Z]+"""
val at = """@"""
val domainSegment = """[^@\.]+"""
val dot = """\."""
val email = raw"""(?x)
  \A
  $user
  $at
  (
    $domainSegment $dot
  )+
  $domainSegment
  \z"""

val emailRegex = email.r
```

Scala notes:

- Triple quotes are for raw strings (no backslash interpretation)
- raw allows variable interpolation in raw strings

Email parser: Ruby

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

```
require ’parslet’

# A simplified email address parser
class EmailValidator::Parser < Parslet::Parser
  rule(:local_part) { match[’a-zA-Z’].repeat(1) }
  rule(:at)         { str(’@’) }
  rule(:domain_char) { match[’^@\\.’] }
  rule(:sub_domain) { domain_char.repeat(1) }
  rule(:dot)    { str(’.’) }
  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.

```scala
object EmailParsers extends RegexParsers {
  override def skipWhitespace = false

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

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

Error reporting: Ruby

We have achieved the goal of decent error reporting:

```ruby
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
```

```
$ bundle exec bin/validate_email 'FranklinChen+spam@cmu.edu
Parse failed, and here's why:
Failed to match sequence (LOCAL_PART AT DOMAIN) at line 1 c
`- Expected "@", but got "+" at line 1 char 13.
```

Error reporting: Scala

We have achieved the goal of decent error reporting:

```
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

- 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?

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 you may want to use someone else's regex
- If you write a regex, modularize it
- For error reporting, use a parser: convert from modularized regex
- Do you even need to solve the problem?

Example: toy JSON parsing

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

- Would you use a regex to parse?
- Could you use a regex to parse?

Toy JSON parser: Scala

(To save time: no more Ruby code in this presentation.)

```
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
}
```

Inheriting from `JavaTokenParsers` allows the reuse of `stringLiteral` and `floatingPointNumber` parsers.

Fancier toy JSON parser: use the data

Want to actually shape and use the data parsed.

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

Example: traverse the JSON to `address` and then to the second neighbor, `Y`.

Domain modeling: Scala

```scala
sealed trait 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

```scala
def value: Parser[ToyJSON] = obj | arr |
  stringLiteralStripped ^^ { JString(_) } |
  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 }
```

Using the parsed JSON

A test using Specs2 Scala testing framework:

```
parsers.value must succeedOn("""{
  "distance" : 5.6,
  "address" : {
    "street" : "0 Nowhere Road",
    "neighbors" : ["X", "Y"],
    "garage" : null
  }
}""").withResult({ result: ToyJSON =>
  result.asInstanceOf[JObject].
    map("address").asInstanceOf[JObject].
    map("neighbors").asInstanceOf[JArray].
    list(1).asInstanceOf[JString].
    string
} ^^ be_==("Y"))
```

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

- Just parsing is simple
- Domain modeling is trickier
- *Use an existing JSON parsing library already!*

Final example of real regex code

Recently in Octopress Ruby code:

```
EXPRESSION = /(.+?)\s+(unless|if)\s+(.+)/i
TERNARY = /(.*?)\(\s*(.+?)\s+\?\s+(.+?)\s+:\s+(.+?)\s*\)(.

def strip_expression(markup, context = false)
  if markup =~ TERNARY
    result = evaluate_ternary($2, $3, $4, context)
    markup = "#{$1} #{result} #{$5}"
  end
  markup =~ EXPRESSION ? $1 : markup
end
```

Depending on the nature of corner cases and nesting, real parser might be preferable.

Conclusion

- Regular expressions
  - No error reporting
  - Flat data
- More general grammars
  - Composable structure (using combinator parsers)
  - Hierarchical, nested data
- Avoid reinventing

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