Rethinking (Replacing) Regular Expressions after 50 Years

Jamie A. Jennings, Ph.D.
Department of Computer Science
NC State University
23 September 2019

@jamietheriveter
https://rosie-lang.org
https://gitlab.com/roseie-pattern-language
What’s wrong with regex?
Syntax
Syntax

- **Compact** (dense)
  - Great for slow terminals!
Syntax

- **Compact** (dense)
  - Great for slow terminals!

https://commons.wikimedia.org/wiki/User:AlisonW
Syntax

- **Compact** (dense)
  - Great for slow terminals!

- **Efficient** (confusing)
  - A symbol can have many meanings!
  - E.g. ^ * − ( ) ?
Syntax

- **Compact** (dense)
  - Great for slow terminals!

- **Efficient** (confusing)
  - A symbol can have many meanings!
  - E.g. `^ * - ( ) ?`

- **Write and forget** (unmaintainable)
  - `grep  -v "^#\|^\\|^\\/\\/"`
  - `egrep '((\d{1,3})([.]\d{1,3}){2}|\w+([.]\w+)+)'`

https://commons.wikimedia.org/wiki/User:AlisonW
Syntaxes

- **Compact** (dense)
  - Great for slow terminals!

- **Efficient** (confusing)
  - A symbol can have many meanings!
  - E.g. `^ * - ( ) ?`

- **Write and forget** (unmaintainable)
  - `grep -v "^#|'|\|\|\|\|"`
  - `egrep '((\d{1,3})([.]\d{1,3}){2}|\w+(\[.\]\w+)+)'`

https://commons.wikimedia.org/wiki/User:AlisonW
Semantics
Semantics

- Not that of Regular Languages
Semantics

- Not that of Regular Languages
- Posix or Perl (or PCRE or js or …)
Semantics

- Not that of Regular Languages
- Posix or Perl (or PCRE or js or …)
- Variations by implementation
  - What does . (dot) match?
  - What does \10 mean?
Semantics

- Not that of Regular Languages
- Posix or Perl (or PCRE or js or …)
- Variations by implementation

What does (dot) match?
What does \10 mean?

Regular expression - Wikipedia
https://en.wikipedia.org › wiki › Regular_expression

A regular expression, regex or regexp is a sequence of characters that define a search pattern. Usually such patterns are used by string searching algorithms for …
Semantics

- Not that of Regular Languages
- Posix or Perl (or PCRE or js or …)
- Variations by implementation
  - What does . (dot) match?
  - What does \10 mean?
- Depends on flags *not in the expr!*
Semantics

\texttt{re.compile(pattern, flags=0)}

 Compile a regular expression pattern into a \textit{regular expression object}, which can be used for matching using its \texttt{match()}, \texttt{search()} and other methods, described below.

The expression’s behaviour can be modified by specifying a \texttt{flags} value. Values can be any of the following variables, combined using bitwise OR (the | operator).

- What does . (dot) match?
- What does \texttt{\10} mean?

\textbullet\quad \textbf{Depends on flags not in the expr!}
**Semantics**

```
re.compile(pattern, flags=0)
```

Compile a regular expression pattern into a regular expression object. This can be used for matching using its `match()` method, as described below.

The expression’s behaviour can be modified by using flags. A flag can be any of the following variables, combined using an OR operator.

- What does . (dot) match?
- What does \10 mean?
- Depends on flags **not in the expr.**

### PCRE:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Default</th>
<th>Change with</th>
</tr>
</thead>
<tbody>
<tr>
<td>. matches newline</td>
<td>no</td>
<td>PCRE_DOTALL</td>
</tr>
<tr>
<td>newline matches [^a]</td>
<td>yes</td>
<td>not changeable</td>
</tr>
<tr>
<td>$ matches \n at end</td>
<td>yes</td>
<td>PCRE_DOLLARENDONLY</td>
</tr>
<tr>
<td>$ matches \n in middle</td>
<td>no</td>
<td>PCRE_MULTILINE</td>
</tr>
<tr>
<td>^ matches \n in middle</td>
<td>no</td>
<td>PCRE_MULTILINE</td>
</tr>
</tbody>
</table>

This is the equivalent table for POSIX:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Default</th>
<th>Change with</th>
</tr>
</thead>
<tbody>
<tr>
<td>. matches newline</td>
<td>yes</td>
<td>REG_NEWLINE</td>
</tr>
<tr>
<td>newline matches [^a]</td>
<td>yes</td>
<td>REG_NEWLINE</td>
</tr>
<tr>
<td>$ matches \n at end</td>
<td>no</td>
<td>REG_NEWLINE</td>
</tr>
<tr>
<td>$ matches \n in middle</td>
<td>no</td>
<td>REG_NEWLINE</td>
</tr>
<tr>
<td>^ matches \n in middle</td>
<td>no</td>
<td>REG_NEWLINE</td>
</tr>
</tbody>
</table>
Semantics

- Not that of Regular Languages
- Posix or Perl (or PCRE or js or …)
- Variations by implementation
  - What does . (dot) match?
  - What does \10 mean?
- Depends on flags *not in the expr!*
- Combining is fraught
Semantics

- Not that of Regular Languages
- Posix or Perl (or PCRE or js or …)
- Variations by implementation
  - What does . (dot) match?
  - What does \10 mean?
- Depends on flags *not in the expr!*
- Combining is fraught
Semantics

- Not that of Regular Languages
- Posix or Perl (or PCRE or js or …)
- Variations by implementation
  - What does . (dot) match?
  - What does \10 mean?
- Depends on flags *not in the expr!*
- Combining is fraught
- No “persistence” (packaging) std
The [fully RFC 822 compliant regex](http://emailregex.com/) is inefficient and obscure because of its length. Fortunately, RFC 822 was superseded twice and the current specification for email addresses is RFC 5322. RFC 5322 leads to a regex that can be understood if studied for a few minutes and is efficient enough for actual use.

One RFC 5322 compliant regex can be found at the top of the page at [http://emailregex.com/](http://emailregex.com/) but uses the IP address pattern that is floating around the internet with a bug that allows `00` for any of the unsigned byte decimal values in a dot-delimited address, which is illegal. The rest of it appears to be consistent with the RFC 5322 grammar and passes several tests using `grep -P`, including cases: domain names, IP addresses, bad ones, and account names with and without quotes.

Correcting the `00` bug in the IP pattern, we obtain a working and fairly fast regex. (Scrape the rendered version, not the markdown, for actual code.)

```
(?::[a-z0-9!#\$%\^&\*\(+\-\_\{\}\~\=\?\^\_\{"\}~\+]\.|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|\|}\]
``
The **fully RFC 822 compliant regex** is inefficient and obscure because of its length. Fortunately, RFC 822 was superseded twice and the current specification for email addresses is [RFC 5322](http://www.ietf.org/rfc/rfc5322.txt). RFC 5322 leads to a regex that can be understood if studied for a few minutes and is efficient enough for actual use.

One RFC 5322 compliant regex can be found at the top of the page at [http://emailregex.com/](http://emailregex.com/) but uses the IP address pattern that is floating around the internet with a bug that allows `00` for any of the unsigned byte decimal values in a dot-delimited address, which is illegal. The rest of it appears to be consistent with the RFC 5322 grammar and passes several tests using `grep -Po`, including cases: domain names, IP addresses, bad ones, and account names with and without quotes.

Correcting the `00` bug in the IP pattern, we obtain a working and fairly fast regex. (Scrape the rendered version, not the markdown, for actual code.)

```regex
(?::.\[a-z0-9!#\$%\&\'*+/=?^_`{}~-]+@[a-z0-9][a-z0-9-]*\.[a-z0-9][a-z0-9-]*\.[a-z]{2,})
```
Python regular expressions for IPv4 and IPv6 addresses and URI-references, based on RFC 3986's ABNF. The URI-reference regular expression includes IPv6 address zone ID support (RFC 6874).

```python
gistfile1.py

# Python regular expressions for IPv4 and IPv6 addresses and URI-references,
# based on RFC 3986's ABNF.
#
# ipv4_address and ipv6_address are self-explanatory.
# ipv6_addrz requires a zone ID (RFC 6874) follow the IPv6 address.
# ipv6_address_or_addrz allows an IPv6 address with optional zone ID.
# uri_reference is what you think of as a URI. (It uses ipv6_address_or_addrz.)

import re

ipv4_address = re.compile('^(?:\d{1,3}\.){3}\d{1,3}$')
ipv6_address = re.compile('^(?:[0-9A-Fa-f]{1,4}:){7}[0-9A-Fa-f]{1,4}$')
ipv6_addrz = re.compile('^(?:[0-9A-Fa-f]{1,4}:){7}[0-9A-Fa-f]{1,4}\:[0-9A-Fa-f]{1,4}$')
ipv6_address_or_addrz = re.compile('^(?:[0-9A-Fa-f]{1,4}:){6}[0-9A-Fa-f]{1,4}\:[0-9A-Fa-f]{1,4}$')
uri_reference = re.compile('^(?:[A-Za-z][A-Za-z-9]+\d+\])**$')

# len(ipv4_address) == 111
# len(ipv6_address) == 1501
# len(ipv6_addrz) == 1541
# len(ipv6_address_or_addrz) == 1546
# len(uri_reference) == 4445
```
Python regular expressions for IPv4 and IPv6 addresses and URI-references, based on RFC 3986's ABNF. The URI-reference regular expression includes IPv6 address zone ID support (RFC 6874).

```python
gistfile1.py
1  # Python regular expressions for IPv4 and IPv6 addresses and URI-references,
2  # based on RFC 3986's ABNF.
3  #
4  # ipv4_address and ipv6_address are self-explanatory.
5  # ipv6_addrz requires a zone ID (RFC 6874) follow the IPv6 address.
6  # ipv6_address_or_addrz allows an IPv6 address with optional zone ID.
7  # uri_reference is what you think of as a URI. (It uses ipv6_address_or_addr.)
8  #
9  import re

10  ipv4_address = re.compile(r'^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$')
11  ipv6_address = re.compile(r'^[0-9A-Fa-f]{1,4}:\d{0,4}[0-9A-Fa-f]{1,4}([0-9A-Fa-f]{1,4}:\d{0,4}[0-9A-Fa-f]{1,4})*\d{1,4}:\d{0,4}[0-9A-Fa-f]{1,4}$')
12  ipv6_addrz = re.compile(r'^[0-9A-Fa-f]{1,4}:\d{0,4}[0-9A-Fa-f]{1,4}([0-9A-Fa-f]{1,4}:\d{0,4}[0-9A-Fa-f]{1,4})*\d{1,4}:\d{0,4}[0-9A-Fa-f]{1,4}:0$')
13  ipv6_address_or_addrz = re.compile(r'^([0-9A-Fa-f]{1,4}:\d{0,4}[0-9A-Fa-f]{1,4}([0-9A-Fa-f]{1,4}:\d{0,4}[0-9A-Fa-f]{1,4})*\d{1,4}:\d{0,4}[0-9A-Fa-f]{1,4}:0)$')
14  uri_reference = re.compile(r'^\w+:\/\/(?:[A-Za-z][A-Za-z-9]+\d+//\d+)*\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$')

17  # len(ipv4_address) == 111
18  # len(ipv6_address) == 1501
19  # len(ipv6_addrz) == 1541
20  # len(ipv6_address_or_addrz) == 1546
21  # len(uri_reference) == 4445
```
Python regular expressions for IPv4 and IPv6 addresses and URI-references from net.ipv4 and net.ipv6.
Expressive Power
Expressive Power

- Regular languages are limited
  - But DFAs are fast!
Expressive Power

- Regular languages are limited
  - But DFAs are fast!
Expressive Power

- Regular languages are limited
  - But DFAs are fast!
- Hence, feature creep
  - Backreferences
  - Conditionals
  - “Subroutines”, Perl6 grammars
  - Recursion
Expressive Power

- Regular languages are limited
  - But DFAs are fast!
- Hence, feature creep
  - Backreferences
  - Conditionals
  - “Subroutines”, Perl6 grammars
  - Recursion
- Yet, static analysis needed!
  - Challenge: # dialects × # impls
Expressive Power

- Regular languages are limited
  - But DFAs are fast!
- Hence, feature creep
  - Backreferences
  - Conditionals
  - “Subroutines”, Perl6 grammars
  - Recursion
- Yet, static analysis needed!
  - Challenge: # dialects \times # impls
Implementation Issues
Implementation Issues

- Exponential time algorithm is by far the most common
Implementation Issues

- Exponential time algorithm is by far the most common

- Most regex are embedded DSLs
  - Syntax issues (escaping)
  - Type issues
  - Requires scaffolding to write/debug regex
    - Less than 17% are tested, most lacking both positive & negative tests [Wang, Stolee ESEC/FSE ’18]
Why work on this?

1. “Every day, we create 2.5 quintillion bytes of data”

Estimates are that less than 0.5% of data is ever analyzed.

Antonio Regalado
MIT Technology Review
Why work on this?

1. “Every day, we create 2.5 quintillion bytes of data”
   
2. Regex use does not scale (# exps, # people, project lifetime)

Estimates are that less than 0.5% of data is ever analyzed.
Rosie Pattern Language

“All progress depends on the unreasonable [woman]”
George Bernard Shaw, paraphrased
Chomsky hierarchy

Formal basis

- recursively enumerable
- context-sensitive
- context-free
- regular

Parsen Expression Grammars: A Recognition-Based Syntactic Foundation

Bryan Ford
Massachusetts Institute of Technology
Cambridge, MA
bford@mit.edu

Abstract

For decades we have been using Chomsky’s generative syntax of grammars, particularly context-free grammars (CFGs) and regular languages, in the form of programming languages and protocols. The power of generative grammar is in its ability to describe a large class of possible languages. By construction, however, CFGs are not able to capture all of the nuances of human languages, but this very power makes it nearly impossible to express the syntax of natural languages, let alone to recognize them. Parsing Expression Grammars (PEGs) provide an alternative, recognition-based formalism for describing multilevel grammars, which solves the ambiguity problem by not introducing ambiguity in the first place. When CFGs express undecidability, PEGs address this by expressing grammars in a formal manner. PEGs address the problem of expression limitation of CFGs and REX by specifying syntax definitions and making it necessary to separate their lexical and syntactic components. A context-free parser can be built for any PEG, avoiding both the complexity and futility of LR parser and the inefficiency of generalized CFG parsing. While PEGs provide a rich set of operators for defining abstract syntax trees, they do not support declarations, and include an abstract syntax tree for the abstract syntax tree of the abstract syntax tree of the abstract syntax tree. This is a syntax tree that is a tree that is a tree that is a tree.

1 Introduction

Most language-on syntax theory and practice is based on generative systems, such as regular expressions and context-free grammar, which is a language defined by a set of rules applied recursively to generate strings of the language. A recognition-based syntax theory, such as PEGs, is based on the idea that languages can be defined by a set of rules that do not generate strings in the language. In contrast, PEGs are expressed purely as concrete parsers. For example, \( r \rightarrow r + r \) is a syntactic definition of a trivial language over the symbols \(+\) and \( \cdot \). In contrast, \( r \rightarrow [r \cdot] [\cdot \cdot [\cdot \cdot [\cdot \cdot [\cdot \cdot ]]]] \) is a recognition-based definition of the same language, which is a string that is a string that is a string that is a string.

While most language theory adopts the generative paradigm, most practical language applications in computer science involve the recognition and structural decomposition, or parsing, of strings. PEGs avoid the gaps from generative definitions to practical recogni- tion in the purpose of use in supporting the theory of parsing algorithms with clear restrictions and made straightforward.

Chomsky’s generative syntax of grammars, from which the origi-
Regular Expressions (strict)
Chomsky hierarchy

Rosie Pattern Language (and all PEG grammars)

Formal basis

- recursively enumerable
- context-sensitive
- context-free

Regular Expressions (strict)

Concrete example of regular expressions:
- `(a|b)*` matches any string consisting of any number of `a`'s and/or `b`'s.

Regular Expressions (strict)

- Chomsky hierarchy
  - Formal basis
  - Regular Expressions (strict)
  - Context-free
  - Context-sensitive
  - Recursively enumerable

Chomsky’s generative system of grammars, from which the origi-
Formal basis

Chomsky hierarchy

- recursively enumerable
- context-free
- context-sensitive

Regular Expressions (strict)

- regular

Open Question: PEG > CFG

Chomsky hierarchy:

- Formal basis
- Regular Expressions (strict)
- Language (and all PEG grammars)

Rosie Pattern Language

- Chomsky hierarchy

RPL syntax: like a programming language

--- ←→ Mode: rpl; ←→
---
--- json.rpl  rpl patterns for processing json input
---
--- © Copyright IBM Corporation 2016, 2017.
--- LICENSE: MIT License (https://opensource.org/licenses/mit-license.html)
--- AUTHOR: Jamie A. Jennings

package json

import word, num

local key = word.dq
local string = word.dq
local number = num.signed_number

local true = "true"
local false = "false"
local null = "null"

grammar
  value = ~ string / number / object / array / true / false / null
  member = key ":" value
  object = "{" (member{""," member)*? "}"
  array = "[" (value{""," value)*? "]"
end

--- test value accepts "true", "false", "null"
--- test value rejects "true", "f", "NULL"
--- test value accepts "0", "123", "-1", "1.1001", "1.2e10", "1.2e-10", "+3.3"
--- test value accepts "\"hello\\\"", "\"this string has \\
  embedded\\
  double quotes\"
--- test value rejects "hello", "\"this string has no \\
  final quote\\"
--- test value rejects "-2", "9.1.", "9.1.2", "42", "2E02."
--- test value accepts "[", "[1, 2, 3.14, \\
  6.02e23, true]", "[1, 2, [7], [8]]"
--- test value rejects "[", "[", "[]", "[1, 2]"
--- test value accepts "\"one\":1", "{ \"one\": 1 }", "\"one\": 1\"
--- test value accepts "\"one\":1, \"two\": 2", "\"one\":1, \"two\": 2, \"array\":[1,2]"
--- test value accepts "\"v\":1", "\"v\":2", "\"v\":3"
RPL syntax: like a programming language

package json

import word, num

local key = word.dq
local string = word.dq
local number = num.signed_number

local true = "true"
local false = "false"
local null = "null"

grammar
value = ~ string / number / object / array / true / false / null
member = key ":" value
object = "{" (member ("," member)* )? "}"
array = "[" (value ("," value)* )? "]"

end

-- test value accepts "true", "false", "null"
-- test value rejects "true", "f", "NULL"
-- test value accepts 0, "123", "-1", "1.1001", "1.2e10", "1.2e-10", "+3.3"
-- test value accepts "\"hello\"", "\"this string has \"double quotes\""
-- test value rejects "hello", "\"this string has no \"double quote\"
-- test value rejects --2", "9.1", "9.1.2", "+42", "2E02."
-- test value accepts [], [1, 2, 3.14, \"\", 6.02e23, true], [1, 2, [7], [[8]]]
-- test value rejects [], ["", [""], [1, 2]]
-- test value accepts "\"one\":1", "{\"one\": :1}", "{\"one\": :1 }
-- test value accepts "\"one\":1, \"two\": 2", "{\"one\":1, \"two\": 2, \"array\":[1,2]}
-- test value accepts "{\"\":1}, {\"\":2}, {\"\":3}"
RPL syntax: like a programming language

```
**** RPL syntax: like a programming language ****

-- mode: rpl

package json

import word, num

local key = word.do
local string = word.do
local number = num.signed

local true = "true"
local false = "false"
local null = "null"

grammar

value = ~ string / number / object / array / true / false / null
member = key ":" value
object = "{" (member ("," member)*)? "}"
array = "[" (value ("," value)*)? "]"

end

-- test value accepts "true", "false", "null"
-- test value rejects "true", "f", "null"
-- test value accepts "0", "123", "-1", "1.1001", "1.2e0", "1.2e-10", "+3.3"
-- test value accepts ""hello"", ""this string has <<<embedded>> double quotes"
-- test value rejects "hello", ""this string has no <<<final quote""
-- test value rejects "-2", "9.1", "9.12", "+2", "2E02."
-- test value accepts "[]", "[1, 2, 3.14, "\"\", 6.02e23, true]", "[1, 2, [7], [[8]]]"
-- test value rejects "[]", "[", "[[]", ",[1, 2]"
-- test value accepts "\"one\":1", ",\"one\":1\", ",\"one\":1\"
-- test value accepts "\"one\":1\", "\"two\":2\", "\"one\":1, \"two\":2, \"array\":[1,2]"
-- test value accepts "\"one\":1\", "\"two\":2\", "\"one\":1, \"two\":2, \"array\":[1,2]"
```
Semantics

- Combinators
Semantics

- Combinators

Matt Might
http://matt.might.net/articles/compiling-up-to-lambda-calculus/
Semantics

- Combinators

\[ \lambda f. (\lambda x. (f (x x))) \lambda x. (f (x x)) \]
Semantics

- Combinators
- Lisp-like macros

*Kleene star is possessive, so  \(\cdot \ast \ "x"\) always fails*

\[ \{ !"x" \ . \} \ast \ "x" \]
Semantics

- Combinators
- Lisp-like macros

Kleene star is possessive, so \( .* \text{"x"} \) always fails

\[
\text{find:} \text{"x"} \overset{\text{def}}{=} \{ !\text{"x"} . \}^* \text{"x"}
\]

Can write this instead
Semantics

- Combinators
- Lisp-like macros

Kleene star is possessive, so $\cdot^* "x"$ always fails

```
find:"x" def \{ !"x" . \}^* "x"
```

Can write this instead

Macros implemented in Lua ... for now.
Semantics

- Combinators
- Lisp-like macros
- Import mechanism like Go
- Prelude like Haskell
- Environments like any Lisp-1
- Binding rules like Scheme
Semantics

- Combinators
- Lisp-like macros
- Import mechanism like Go
- Prelude like Haskell
- Environments like any Lisp-1
- Binding rules like Scheme
Semantics

- Combinators
- Lisp-like macros
- Import mechanism like Go
- Prelude like Haskell
- Environments like any Lisp-1
- Binding rules like Scheme
Semantics

- Combinators
- Lisp-like macros
- Import mechanism like Go
- Prelude like Haskell
- Environments like any Lisp-1
- Binding rules like Scheme
Implementation

“I want to believe”  
Fox Mulder, FBI
Can your ‘grep’ do this?

```
$ curl -s www.google.com | rosie grep -o subs net.url
http://schema.org/WebPage
http://www.google.com/imghp?hl=en&tab=wi
http://maps.google.com/maps?hl=en&tab=wl
https://play.google.com/?hl=en&tab=w8
http://www.youtube.com/?gl=US&tab=w1
http://news.google.com/nwshp?hl=en&tab=wn
https://mail.google.com/mail/?tab=wm
https://drive.google.com/?tab=wo
http://www.google.com/history/optout?hl=en
https://plus.google.com/116899029375914044550
```

- `grep -o subs net.url` outputs sub-matches for the pattern `net.url`.
Can your ‘grep’ do this?

$ rosie match 'word.any (net.any)+' resolv.conf
domain abc.aus.example.com
search ibm.com mylocaldomain.myisp.net example.com
nameserver 192.9.201.1
nameserver 192.9.201.2
nameserver fde9:4789:96dd:03bd::1
$
Can your ‘grep’ do this?

$ rosie match 'word.any (net.any)+' resolv.conf
domain abc.aus.example.com
search ibm.com mylocaldomain.myisp.net example.com
nameserver 192.9.201.1
nameserver 192.9.201.2
nameserver fde9:4789:96dd:03bd::1
$

$ rosie --colors='net.ipv4=blue;bold' match 'word.any (net.any)+' resolv.conf
domain abc.aus.example.com
search ibm.com mylocaldomain.myisp.net example.com
nameserver 192.9.201.1
nameserver 192.9.201.2
nameserver fde9:4789:96dd:03bd::1
$

Customizable Output Highlighting
Can your ‘grep’ do this?

$ sed -n 46,49p /var/log/system.log
Jul 30 10:18:42 Jamies-Compabler systemstats[71]: assertion failed: 17665: systemstats + 914800 [D1E75C38-62CE-3D77-9ED3-5F6D38EF0676]: 0x40
Jul 30 10:18:43 Jamies-Compabler ContainerMetadataExtractor[92065]: objc[92065]: Class BRMangledID is implemented in both /System/Library/PrivateFrameworks/CloudDocs.framework>Versions/A/CloudDocs (0x7fff8b848c88) and /System/Library/PrivateFrameworks/CloudDocsDaemon.framework>XPCServices/ContainerMetadataExtractor.xpc/Contents/MacOS/ContainerMetadataExtractor (0x10a8e0528). One of the two will be used. Which one is undefined.
Jul 30 10:18:50 Jamies-Compabler systemstats[71]: assertion failed: 17665: systemstats + 914800 [D1E75C38-62CE-3D77-9ED3-5F6D38EF0676]: 0x40
Can your ‘grep’ do this?

```bash
$ sed -n 46,49p /var/log/system.log
Jul 30 10:18:42 Jamies-Compabler systemstats[71]: assertion failed: 17G65: systemstats + 914800 [D1E75C 38-62CE-3D77-9ED3-5F6D38EF0676]: 0x40
Jul 30 10:18:43 Jamies-Compabler ContainerMetadataExtractor[92065]: objc[92065]: Class BRMangledID is implemented in both /System/Library/PrivateFrameworks/CloudDocs.framework/Versions/A/CloudDocs (0x7fff8b 848c88) and /System/Library/PrivateFrameworks/CloudDocsDaemon.framework/XPCServices/ContainerMetadataExtractor.xpc/Contents/MacOS/ContainerMetadataExtractor (0x10a8e0528). One of the two will be used. Which one is undefined.
Jul 30 10:18:50 Jamies-Compabler systemstats[71]: assertion failed: 17G65: systemstats + 914800 [D1E75C 38-62CE-3D77-9ED3-5F6D38EF0676]: 0x40
```

```bash
$ sed -n 46,49p /var/log/system.log | rosie match all.things
Jul 30 10:18:42 Jamies-Compabler systemstats[71]: assertion failed: 17G65: systemstats + 914800 [D1E75C 38-62CE-3D77-9ED3-5F6D38EF0676]: 0x40
Jul 30 10:18:43 Jamies-Compabler ContainerMetadataExtractor[92065]: objc[92065]: Class BRMangledID is implemented in both /System/Library/PrivateFrameworks/CloudDocs.framework/Versions/A/CloudDocs (0x7fff8b 848c88) and /System/Library/PrivateFrameworks/CloudDocsDaemon.framework/XPCServices/ContainerMetadataExtractor.xpc/Contents/MacOS/ContainerMetadataExtractor (0x10a8e0528). One of the two will be used. Which one is undefined.
Jul 30 10:18:50 Jamies-Compabler systemstats[71]: assertion failed: 17G65: systemstats + 914800 [D1E75C 38-62CE-3D77-9ED3-5F6D38EF0676]: 0x40
```
Can your ‘grep’ do this?

```
$ head -n 1 /var/log/system.log | rosie grep -o jsonpp num.denoted_hex
{"s": 1, "e": 80, "data": "Jul 29 16:17:13 Jamies-Compabler timed[90268]: settimeofday({0x5b5e20c9,0x75bd3},
subs": [{"s": 62, "e": 72, "data": "0x5b5e20c9", subs": [{"s": 64, "e": 72, "data": "5b5e20c9", type": "num.hex"}],
 "type": "num.denoted_hex"},
{s": 73, "e": 80, "data": "0x75bd3", subs": [{"s": 75, "e": 80, "data": "75bd3", type": "num.hex"}],
 "type": "num.denoted_hex"}],
 "type": "*"}
```

Can your 'grep' do this?

```
$ head -n 1 /var/log/system.log | rosie grep -o jsonpp num.denoted_hex
{"s": 1,
 "e": 80,
 "data": "Jul 29 16:17:13 Jamies-Compabler timed[90268]: settimeofday({0x5b5e20c9,0x75bd3},
 "subs":
 [{"s": 62,
  "e": 72,
  "data": "0x5b5e20c9",
  "subs":
   [{"s": 64,
    "e": 72,
    "data": "5b5e20c9",
    "type": "num.hex"}],
  "type": "num.denoted_hex"},
  {"s": 73,
    "e": 80,
    "data": "0x75bd3",
    "subs":
     [{"s": 75,
      "e": 80,
      "data": "75bd3",
      "type": "num.hex"}],
     "type": "num.denoted_hex"}],
  "type": "*"}
```

$
Can your ‘grep’ do this?

```
$ head -n 1 /var/log/system.log | rosie grep -o jsonpp num.denoted_hex
{"s": 1,
 "e": 80,
 "data": "Jul 29 16:17:13 Jamies-Compabler timed[90268]: settimeofday({0x5b5e20c9,0x75bd3},
 "subs":
 [{"s": 62,
   "e": 72,
   "data": "0x5b5e20c9",
   "subs":
    [{"s": 64,
      "e": 72,
      "data": "5b5e20c9",
      "type": "num.hex"}],
    "type": "num.denoted_hex"},
  [{"s": 73,
    "e": 80,
    "data": "0x75bd3",
    "subs":
     [{"s": 75,
       "e": 80,
       "data": "75bd3",
       "type": "num.hex"}],
     "type": "num.denoted_hex"}],
  "type": "*"
}
```

$
Can your ‘grep’ do this?

```
$ head -n 1 /var/log/system.log | rosie grep -o jsonpp num.denoted_hex
{"s": 1,
 "e": 80,
 "data": "Jul 29 16:17:13 Jamies-Compabler timed[90268]: settimeofday({0x5b5e20c9,0x75bd3},
 "subs":
 [{"s": 62,
   "e": 72,
   "data": "0x5b5e20c9",
   "subs":
    [{"s": 64,
      "e": 72,
      "data": "5b5e20c9",
      "type": "num.hex"},
     "type": "num.denoted_hex"},
    [{"s": 75,
      "e": 80,
      "data": "75bd3",
      "type": "num.hex"},
     "type": "num.denoted_hex"},
    "type": "%"}

$`
```
Rosie Architecture

Patterns

RPL Compiler

Matching Engine

json
color
boolean

{"s": 1, 
"e": 12, 
"type": "net.any", 
"data": "192.168.0.1", 
"subs": 
[{"s": 1, 
"e": 12, 
"type": "net.ip", 
"data": "192.168.0.1", 
"subs": 
[{"s": 1, 
"e": 12, 
"type": "net.ipv4", 
"data": "192.168.0.1"}]
} 
}
1. RPL source
2. Parse tree (Rosie)
3. AST
4. Macro expansion
5. Simplification
6. IR
7. Code generation

rosie architecture

1. RPL source
2. ➞ Parse tree (Rosie)
3. ➞ AST
4. Macro expansion
5. Simplification
6. ➞ IR
7. Code generation

{"s": 1,
"e": 12,
"type": "net.any",
"data": "192.168.0.1",
"subs":
[{
"s": 1,
"e": 12,
"type": "net.ip",
"data": "192.168.0.1",
"subs":
[{
"s": 1,
"e": 12,
"type": "net.ipv4",
"data": "192.168.0.1"
}]
}]

192.168.0.1
Performance

Grok/ruby

Grok/jruby

Rosie 1.0.0

Rosie 1.1.0

Notes:
1. Log entry parsing is one narrow use case.
2. Hard to design fair comparisons.
3. Rosie output is nested JSON; Grok output is flat lists.
4. Rosie is single-threaded.
Debugging

“To err is human, but to really foul things up you need a computer.”

Paul R. Ehrlich
Trace a (mis-)match

$ date | rosie match date.us_dashed
$


Trace a (mis-)match

Pattern definition

Input text

Matching steps

---

Expression: {month "-" day "-" short_long_year}
Looking at: 《Mon Jul 30 12:43:09 EDT 2018》 (input pos = 1)
No match

Expression: month
Looking at: 《Mon Jul 30 12:43:09 EDT 2018》 (input pos = 1)
No match
  Expression: {{"1" [0-2]} / {{"0"}? [1-9]}}
  Looking at: 《Mon Jul 30 12:43:09 EDT 2018》 (input pos = 1)
  No match
    Expression: {{"1" [0-2]}
    Looking at: 《Mon Jul 30 12:43:09 EDT 2018》 (input pos = 1)
    No match
      Expression: "1"
      Looking at: 《Mon Jul 30 12:43:09 EDT 2018》 (input pos = 1)
      No match
        Expression: [0-2]
        Not attempted
          Expression: {{"0"}? [1-9]}
          Looking at: 《Mon Jul 30 12:43:09 EDT 2018》 (input pos = 1)
          No match

Expression: "-"
Not attempted

Expression: day
Not attempted

Expression: "-"
Not attempted

Expression: short_long_year
Not attempted
$ rosie repl
Rosie 1.0.0-sepcomp3
Rosie> import destructure as des
Rosie> .list des.*

<table>
<thead>
<tr>
<th>Name</th>
<th>Cap?</th>
<th>Type</th>
<th>Color</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>[snip]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>numalpha</td>
<td>Yes</td>
<td>pattern</td>
<td>default;bold</td>
<td>destructure</td>
</tr>
<tr>
<td>parentheses</td>
<td>Yes</td>
<td>pattern</td>
<td>default;bold</td>
<td>destructure</td>
</tr>
<tr>
<td>rest</td>
<td>Yes</td>
<td>pattern</td>
<td>default;bold</td>
<td>destructure</td>
</tr>
<tr>
<td>semicolons</td>
<td>Yes</td>
<td>pattern</td>
<td>default;bold</td>
<td>destructure</td>
</tr>
<tr>
<td>sep</td>
<td></td>
<td>pattern</td>
<td>default;bold</td>
<td>destructure</td>
</tr>
<tr>
<td>slashes</td>
<td>Yes</td>
<td>pattern</td>
<td>default;bold</td>
<td>destructure</td>
</tr>
<tr>
<td>term</td>
<td>Yes</td>
<td>pattern</td>
<td>default;bold</td>
<td>destructure</td>
</tr>
<tr>
<td>tryall</td>
<td></td>
<td>pattern</td>
<td>default;bold</td>
<td>destructure</td>
</tr>
<tr>
<td>~</td>
<td></td>
<td>pattern</td>
<td>default;bold</td>
<td>builtin/prelude</td>
</tr>
</tbody>
</table>

24/24 names shown
Rosie>
Read-eval-print loop

- Define patterns
- Try them
- Debug (trace) them

```javascript
Rosie> .match des.tryall "(1.2; 3.77; 0)"
{"data": "(1.2; 3.77; 0)",
"e": 15,
"s": 1,
"subs":
[{"data": "(1.2; 3.77; 0)",
"e": 15,
"s": 1,
"subs":
[{"data": "1.2; 3.77; 0",
"e": 14,
"s": 2,
"subs":
[{"data": "1.2",
"e": 5,
"s": 2,
"type": "des.find.<search>"},
```

```javascript

```javascript

```javascript
```
```javascript

```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
```javascript
```
Read-eval-print loop

- Define patterns
- Try them
- Debug (trace) them

```cpp
Rosie> .match des.tryall "(1.2; 3.77; 0)"
{"data": "(1.2; 3.77; 0)",
 "e": 15,
 "s": 1,
 "subs":
  [{"data": "(1.2; 3.77; 0)",
    "e": 15,
    "s": 1,
    "subs":
      [{"data": "1.2; 3.77; 0",
        "e": 14,
        "s": 2,
        "subs":
          [{"data": "1.2",
            "e": 5,
            "s": 2,
            "type": "des.find.<search>"},
           {"data": "3.77",
            "e": 11,
            "s": 6,
            "type": "des.find.<search>"},
           {"data": "0",
            "e": 14,
            "s": 12
          ]}]
    ]
}]
```
Implementation Roadmap
Implementation Roadmap

✓ librosie as well as CLI, REPL
✓ Modules (shareable)
✓ Unit tests
✓ Output for humans and programs
✓ Standard library (~300 general, ~600 Unicode patterns)
Implementation Roadmap

✓ *librosie* as well as CLI, REPL
✓ Modules (shareable)
✓ Unit tests
✓ Output for humans and programs
✓ Standard library (~300 general, ~600 Unicode patterns)

➡ Automated generation from regex
Implementation Roadmap

✓ *librosie* as well as CLI, REPL
✓ Modules (shareable)
✓ Unit tests
✓ Output for humans and programs
✓ Standard library (~300 general, ~600 Unicode patterns)

➡ Automated generation from regex
➡ Ahead of time compilation
Implementation Roadmap

✓ *librosie* as well as CLI, REPL
✓ Modules (shareable)
✓ Unit tests
✓ Output for humans and programs
✓ Standard library (~300 general, ~600 Unicode patterns)

➡ Automated generation from regex
➡ Ahead of time compilation
➡ Formal semantics
Implementation Roadmap

✓ *librosie* as well as CLI, REPL
✓ Modules (shareable)
✓ Unit tests
✓ Output for humans and programs
✓ Standard library (~300 general, ~600 Unicode patterns)

→ Automated generation from regex
→ Ahead of time compilation
→ Formal semantics
→ Static analysis
  - Worst-case run-time bounds
  - Common errors (linting)
Using Rosie in programs

Today:

- C
- Python
- Haskell
- Go

Once and future:

- Node.js
- Ruby
- Java
- Lua
- Clojure

Programming Language
Thank you!
**Faster**
- Dev time:
  - ✓ library of patterns
  - ✓ composable patterns
- Run time:
  - ✓ good match perf.

**Better**
- Conformance to RFCs
- Readable syntax
- Clear semantics (and no flags)
- Plays well with
  - git/diff
  - package management
  - build automation (unit tests)

**Cheaper**
- ROI in reduced dev & maintenance
- Free open source software (MIT license)

On the interwebs:
@jamietheriveter
https://rosie-lang.org
https://gitlab.com/rosie-pattern-language