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/rosie-pattern-language

IBM

NC STATE
UNIVERSITY

Department of
Computer Science

IBM

NC STATE
UNIVERSITY

Department of
Computer Science

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. `^ * - ( ) ?`

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

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

Syntax**es**

- 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+)+)’`

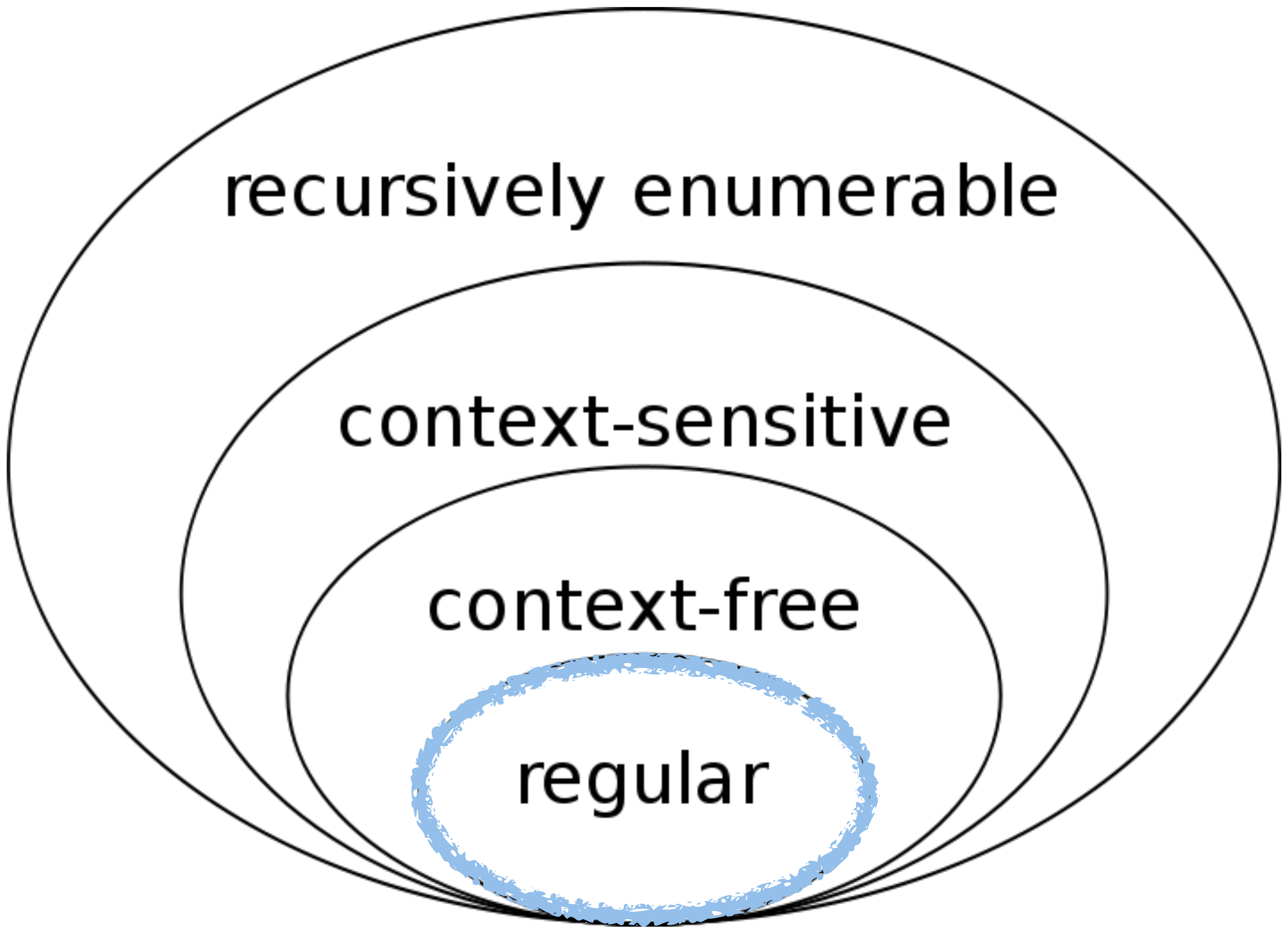
https://commons.wikimedia.org/wiki/User:AlisonW

Semantics

Semantics

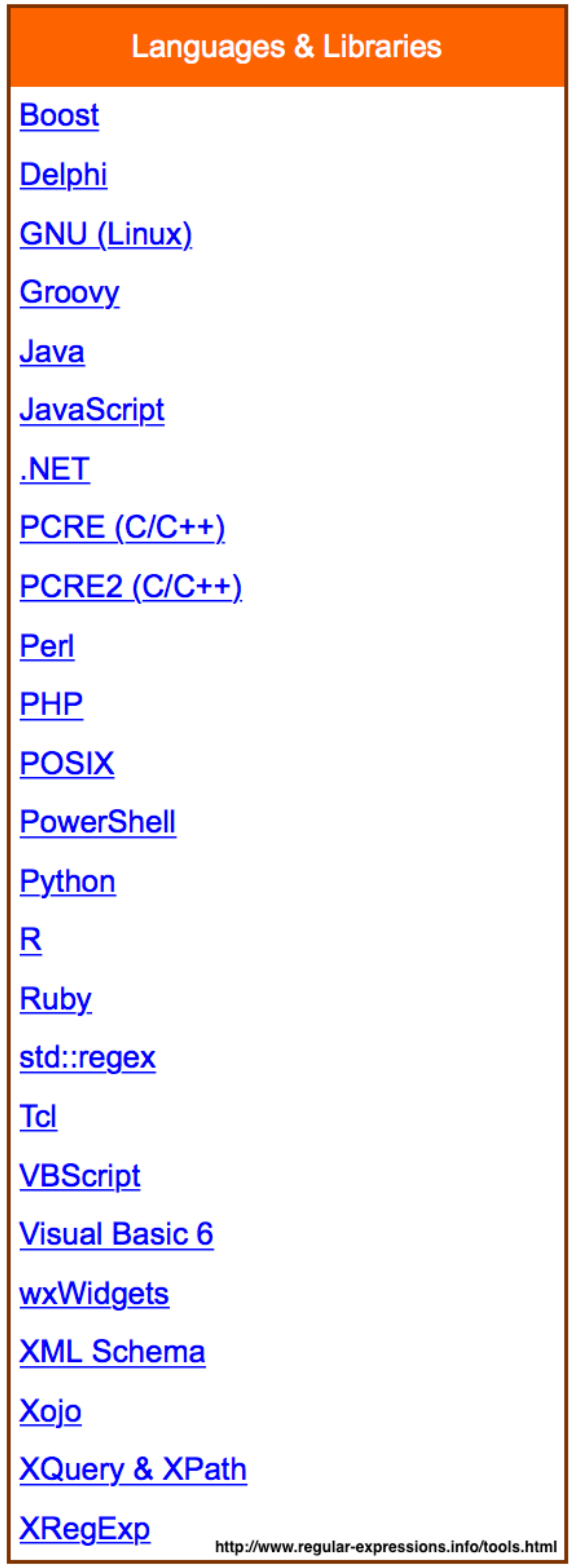
- Not that of Regular Languages

Chomsky hierarchy



Semantics

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



Languages & Libraries

Boost
Delphi
GNU (Linux)
Groovy
Java
JavaScript
.NET
PCRE (C/C++)
PCRE2 (C/C++)
Perl
PHP
POSIX
PowerShell
Python
R
Ruby
std::regex
Tcl
VBScript
Visual Basic 6
wxWidgets
XML Schema
Xojo
XQuery & XPath
XRegExp

http://www.regular-expressions.info/tools.html

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

Google

regular expression

All
Videos
Books
Images
News
More
Settings
Tools

About 360,000,000 results (0.60 seconds)

Regular expression - Wikipedia

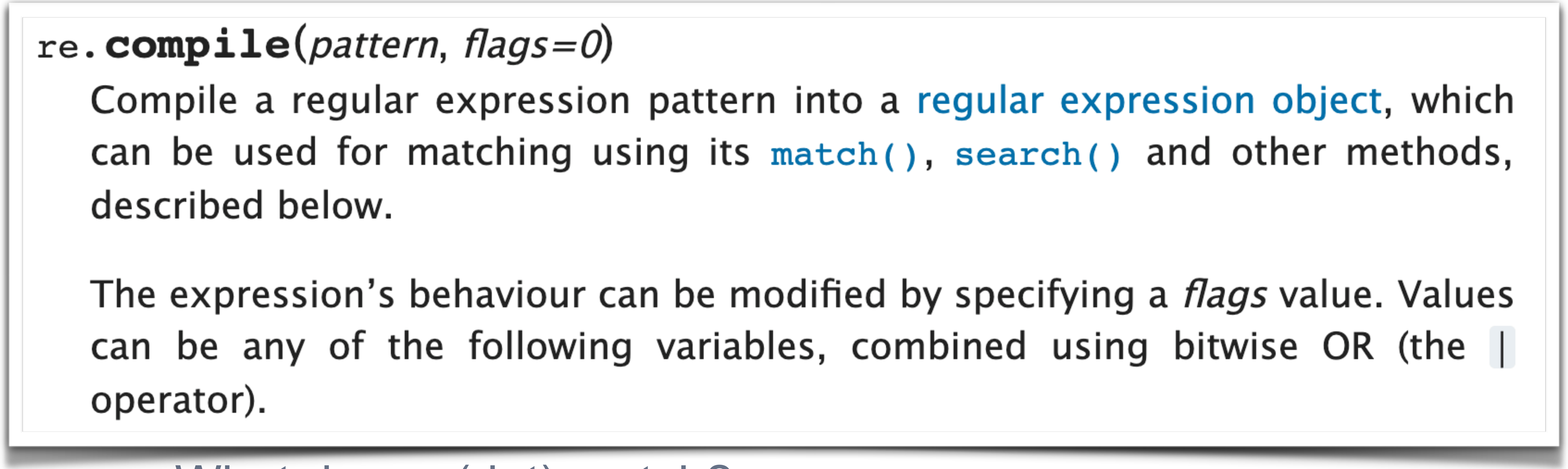
https://en.wikipedia.org › wiki › Regular_expression

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



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

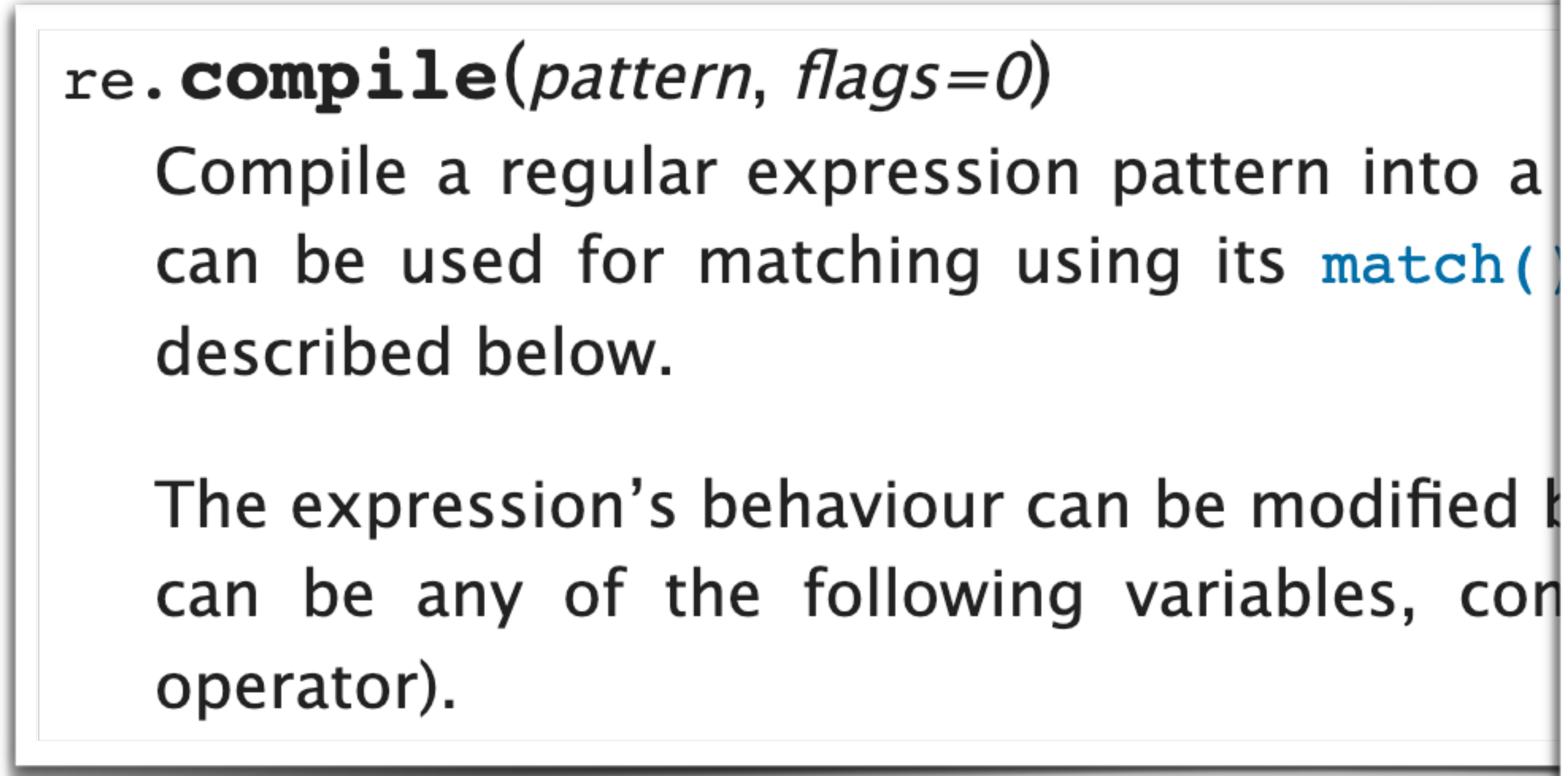
Compile a regular expression pattern into a regular expression object, which can be used for matching using its `match()`, `search()` and other methods, described below.

The expression's behaviour can be modified by specifying a *flags* value. Values can be any of the following variables, combined using bitwise OR (the `|` operator).

- What does `.` (dot) match?
- What does `\10` mean?

- Depends on flags *not in the expr!*

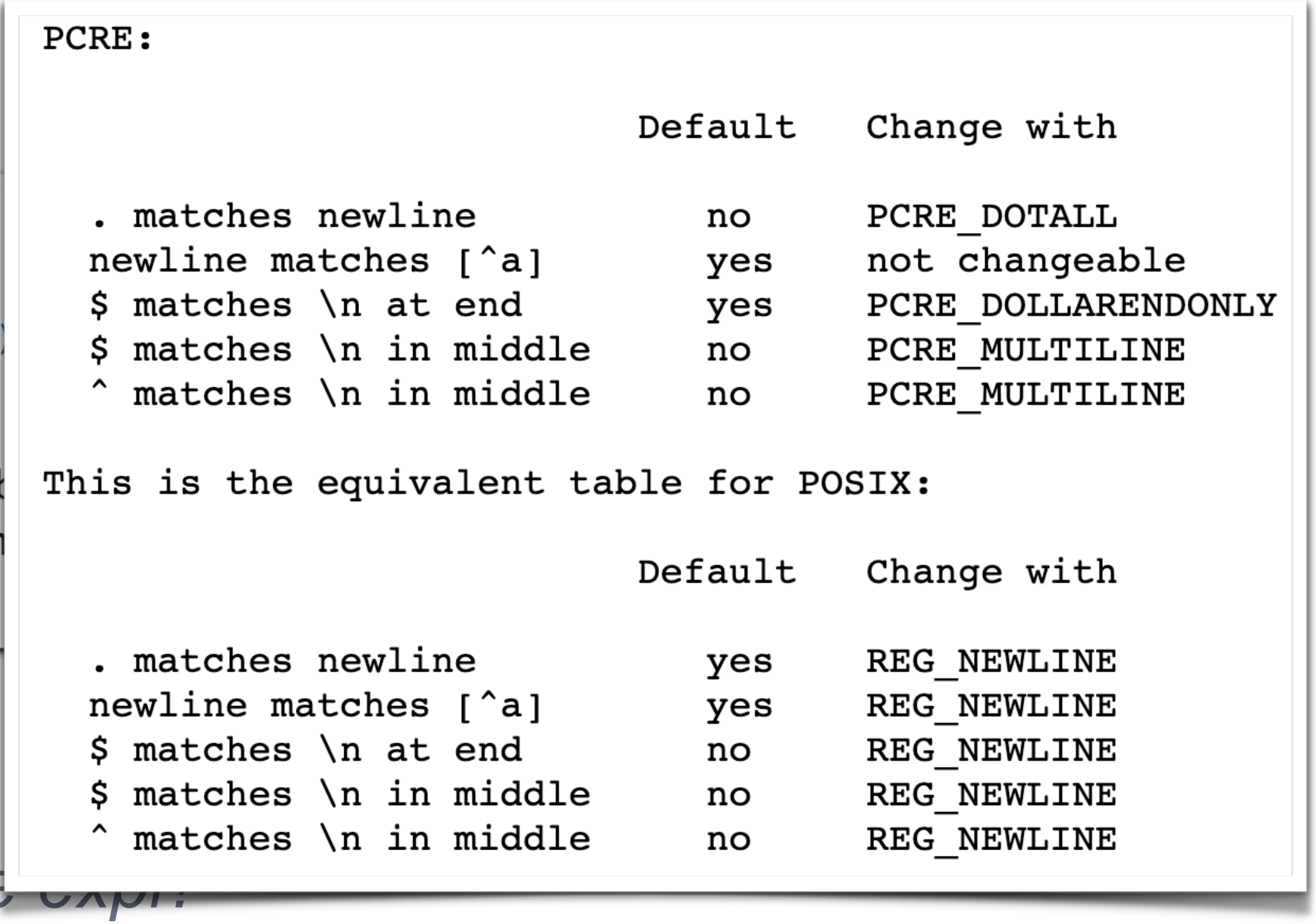
Semantics



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

Compile a regular expression pattern into a
can be used for matching using its `match(`
described below.

The expression's behaviour can be modified b
can be any of the following variables, co
operator).



PCRE:

	Default	Change with
. matches newline	no	PCRE_DOTALL
newline matches [^a]	yes	not changeable
$ matches \n at end	yes	PCRE_DOLLARENDONLY
$ matches \n in middle	no	PCRE_MULTILINE
^ matches \n in middle	no	PCRE_MULTILINE

This is the equivalent table for POSIX:

	Default	Change with
. matches newline	yes	REG_NEWLINE
newline matches [^a]	yes	REG_NEWLINE
$ matches \n at end	no	REG_NEWLINE
$ matches \n in middle	no	REG_NEWLINE
^ matches \n in middle	no	REG_NEWLINE

  - What does `.` (dot) match?
  - What does `\10` mean?

- Depends on flags *not in the expr!*

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

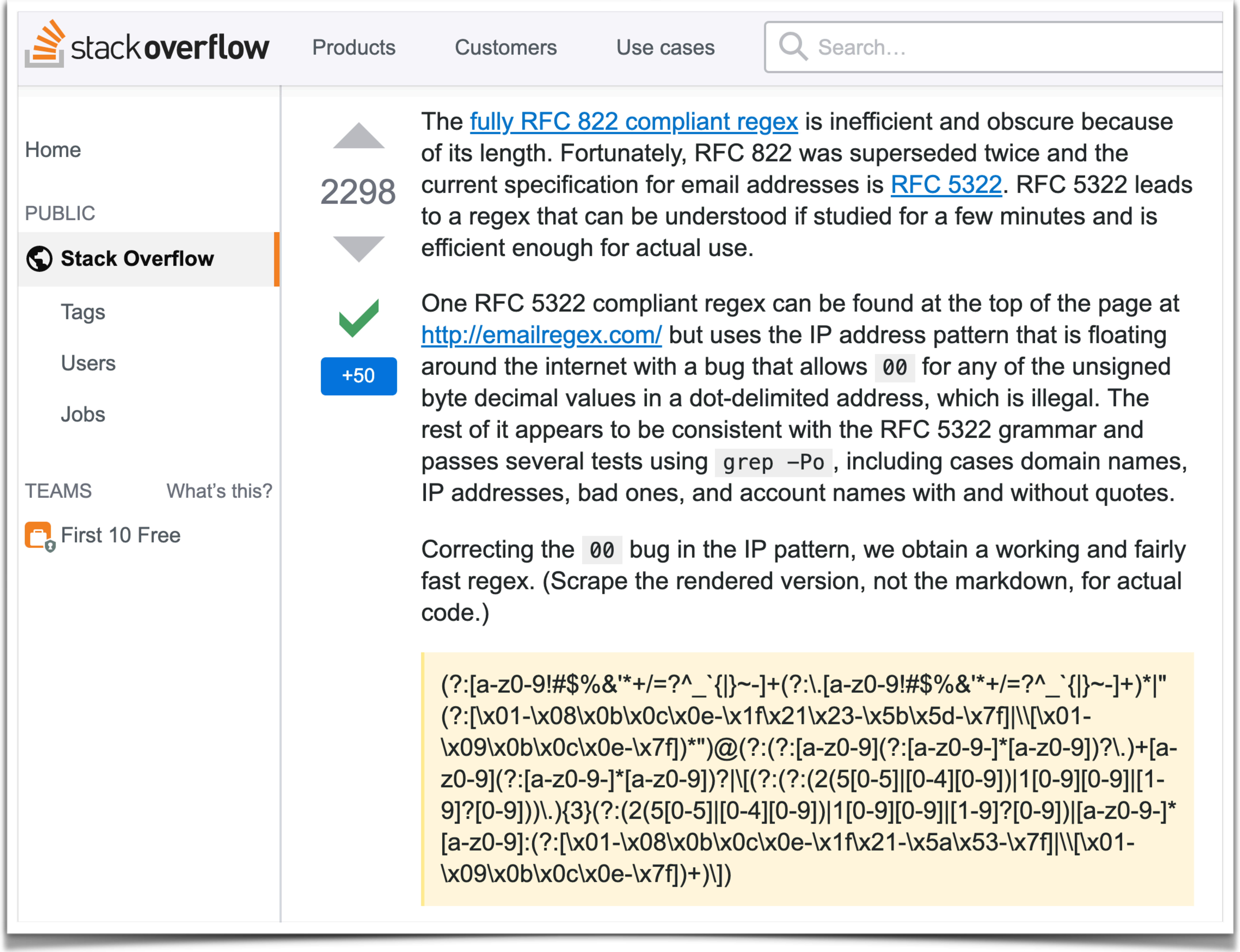
Semantics

- 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

Robert Jacobson: Design's store

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

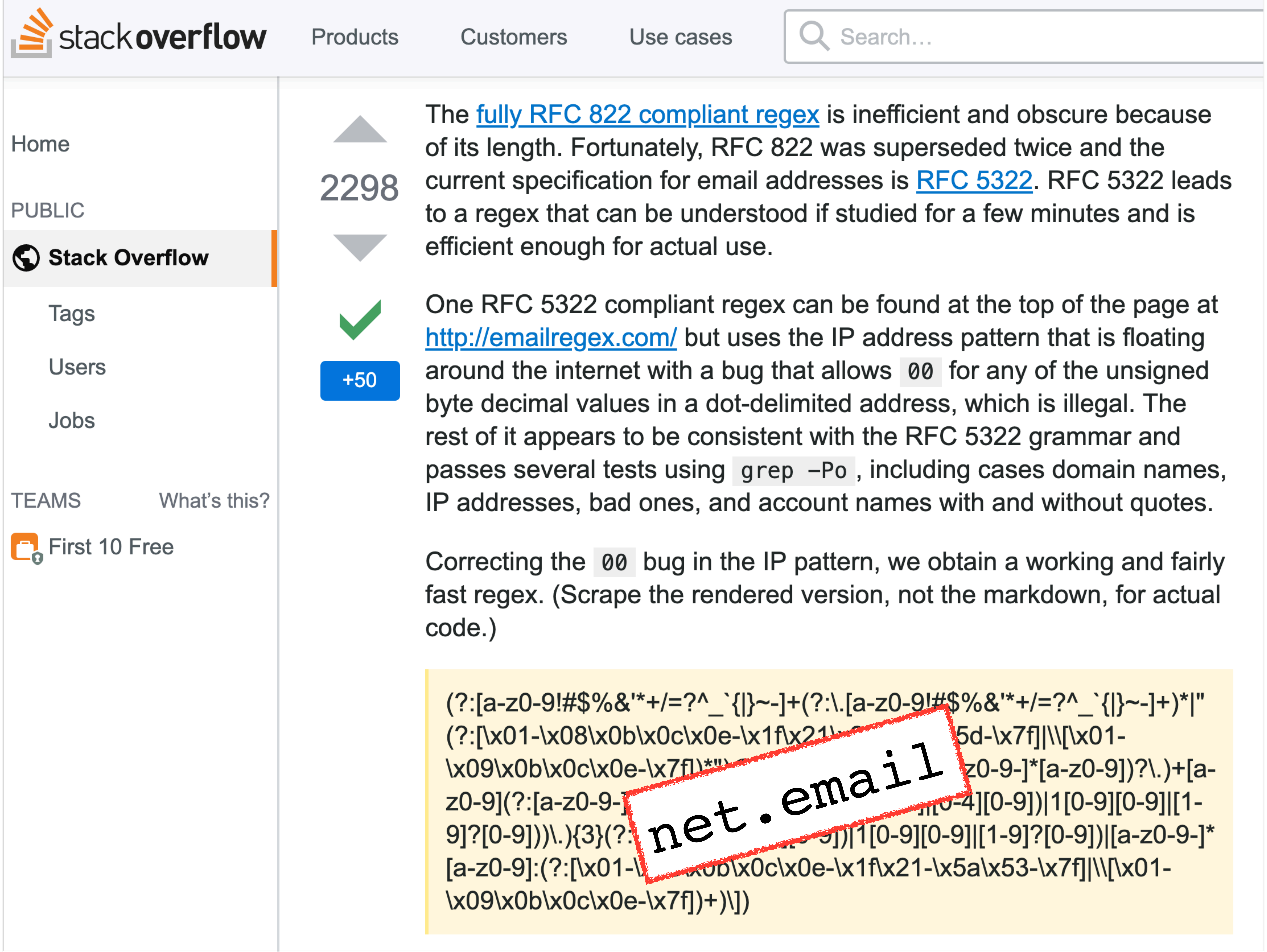


The fully RFC 822 compliant regex is inefficient and obscure because of its length. Fortunately, RFC 822 was superseded twice and the current specification for email addresses is RFC 5322. RFC 5322 leads to a regex that can be understood if studied for a few minutes and is efficient enough for actual use.

One RFC 5322 compliant regex can be found at the top of the page at http://emailregex.com/ 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.)

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"
(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-
\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-
z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-
9]?[0-9]))\.){3}(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9-]*
[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-
\x09\x0b\x0c\x0e-\x7f])+)\])
```



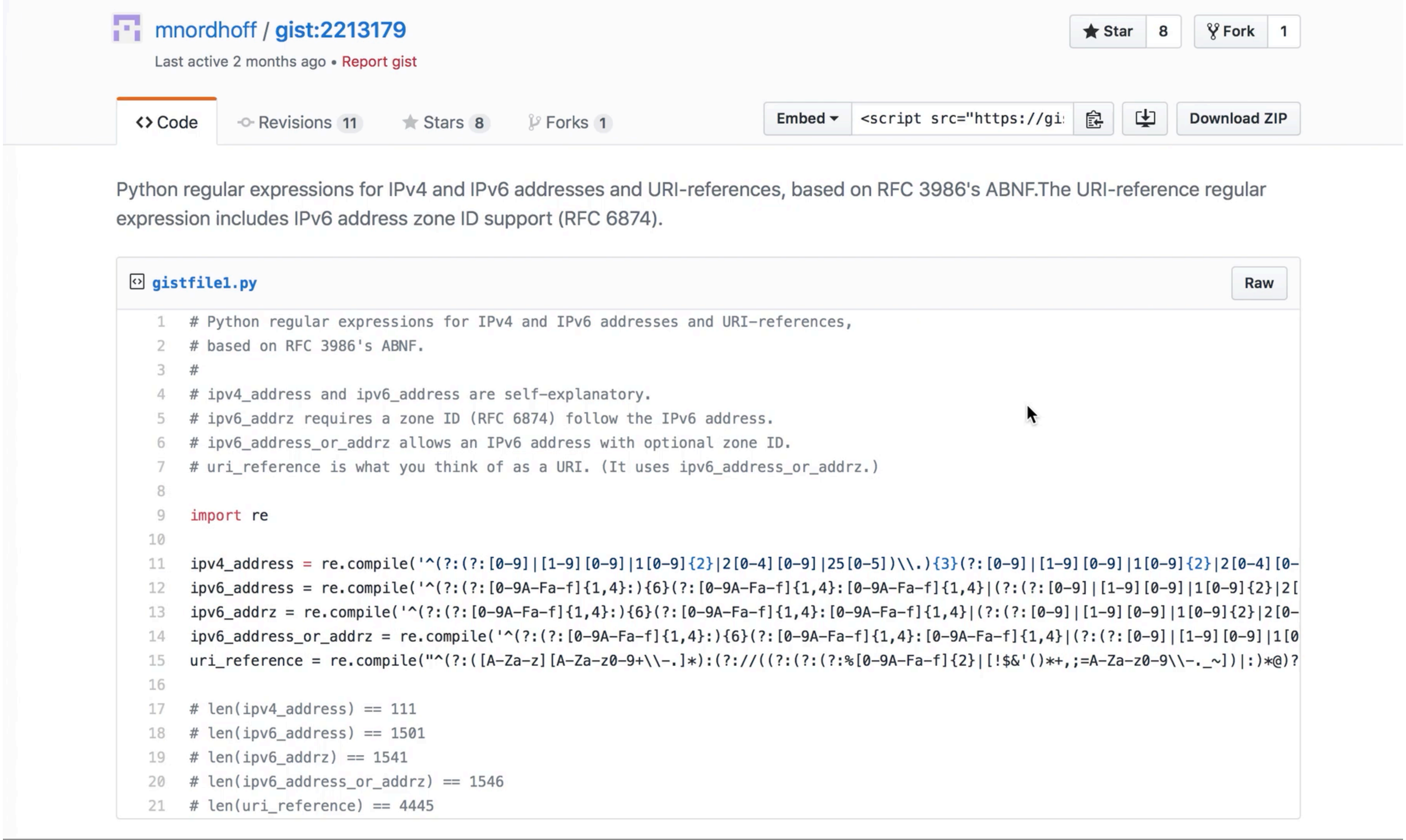
The [fully RFC 822 compliant regex](#) is inefficient and obscure because of its length. Fortunately, RFC 822 was superseded twice and the current specification for email addresses is [RFC 5322](#). RFC 5322 leads to a regex that can be understood if studied for a few minutes and is efficient enough for actual use.

One RFC 5322 compliant regex can be found at the top of the page at [http://emailregex.com/](#) 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.)

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"
(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21                      5d-\x7f]|\\[\x01-
\x09\x0b\x0c\x0e-\x7f])*"                               z0-9-]*[a-z0-9])?\.)+[a-
z0-9](?:[a-z0-9-                                       [0-4][0-9])|1[0-9][0-9]|[1-
9]?[0-9]))\.){3}(?:                                 9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9-]*
[a-z0-9]:(?:[\x01-\                    x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-
\x09\x0b\x0c\x0e-\x7f])+)\])
```

net.email



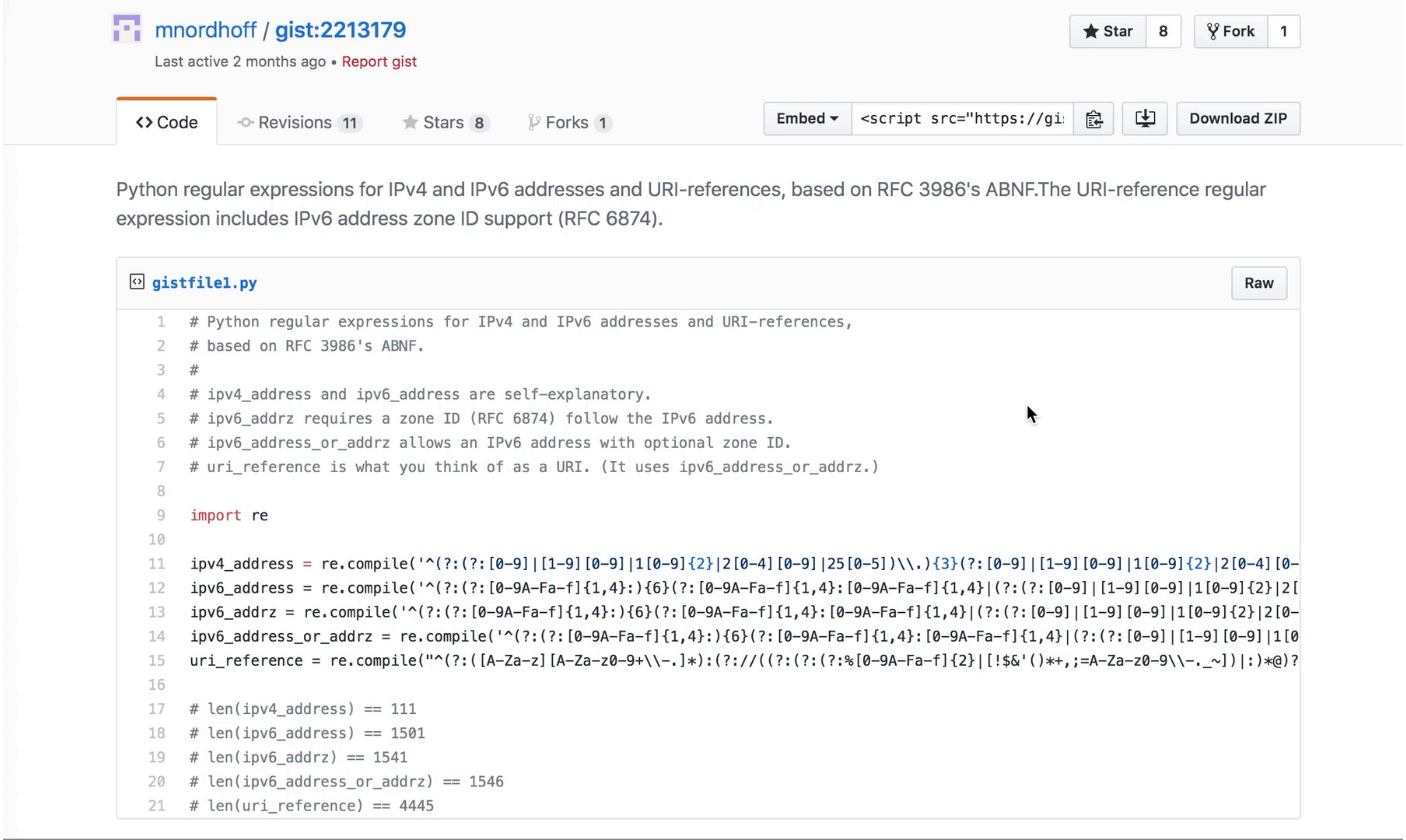
Python regular expressions for IPv4 and IPv6 addresses and URI-references, based on RFC 3986's ABNF.The URI-reference regular expression includes IPv6 address zone ID support (RFC 6874).

```python
# 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('^(?:(?:[0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\\.){3}(?:[0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-
ipv6_address = re.compile('^(?:(?:[0-9A-Fa-f]{1,4}:){6}(?:[0-9A-Fa-f]{1,4}:[0-9A-Fa-f]{1,4}|(?:(?:[0-9]|[1-9][0-9]|1[0-9]{2}|2[
ipv6_addrz = re.compile('^(?:(?:[0-9A-Fa-f]{1,4}:){6}(?:[0-9A-Fa-f]{1,4}:[0-9A-Fa-f]{1,4}|(?:(?:[0-9]|[1-9][0-9]|1[0-9]{2}|2[0-
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}|(?:(?:[0-9]|[1-9][0-9]|1[0
uri_reference = re.compile("^(?:([A-Za-z][A-Za-z0-9+\\-.]*):(?://((?:(?:(?:%[0-9A-Fa-f]{2}|[!$&'()*+,;=A-Za-z0-9\\-._~])|:)*@)?

# len(ipv4_address) == 111
# len(ipv6_address) == 1501
# len(ipv6_addrz) == 1541
# len(ipv6_address_or_addrz) == 1546
# len(uri_reference) == 4445
```



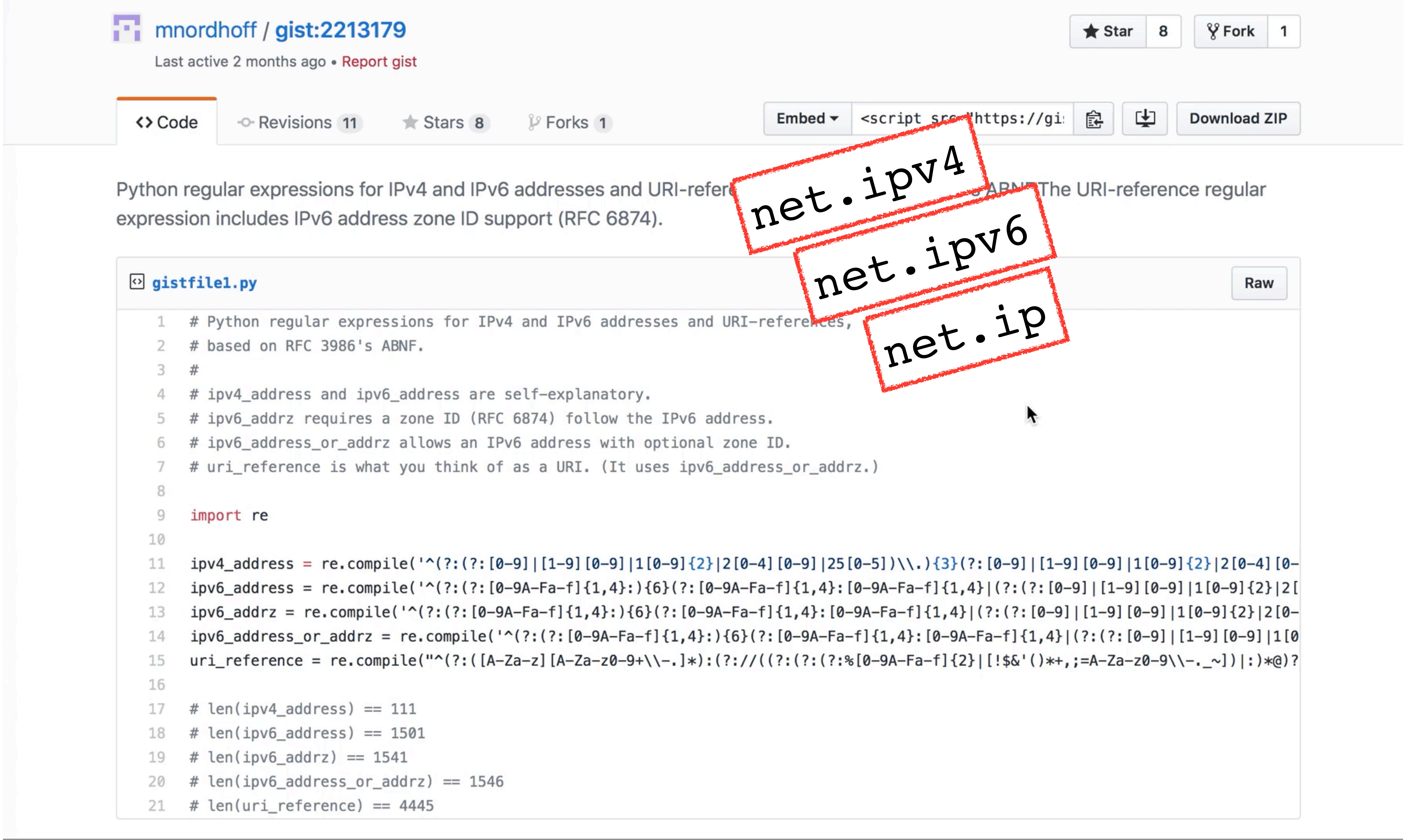
Python regular expressions for IPv4 and IPv6 addresses and URI-references, based on RFC 3986's ABNF.The URI-reference regular expression includes IPv6 address zone ID support (RFC 6874).

```python
# 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('^(?:(?:[0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\\.){3}(?:[0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-
ipv6_address = re.compile('^(?:(?:[0-9A-Fa-f]{1,4}:){6}(?:[0-9A-Fa-f]{1,4}:[0-9A-Fa-f]{1,4}|(?:(?:[0-9]|[1-9][0-9]|1[0-9]{2}|2[
ipv6_addrz = re.compile('^(?:(?:[0-9A-Fa-f]{1,4}:){6}(?:[0-9A-Fa-f]{1,4}:[0-9A-Fa-f]{1,4}|(?:(?:[0-9]|[1-9][0-9]|1[0-9]{2}|2[0-
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}|(?:(?:[0-9]|[1-9][0-9]|1[0
uri_reference = re.compile("^(?:([A-Za-z][A-Za-z0-9+\\-.]*):(?://((?:(?:(?:%[0-9A-Fa-f]{2}|[!$&'()*+,;=A-Za-z0-9\\-._~])|:)*@)?

# len(ipv4_address) == 111
# len(ipv6_address) == 1501
# len(ipv6_addrz) == 1541
# len(ipv6_address_or_addrz) == 1546
# len(uri_reference) == 4445
```



net.ipv4

net.ipv6

net.ip

Python regular expressions for IPv4 and IPv6 addresses and URI-refere ABNF.The URI-reference regular expression includes IPv6 address zone ID support (RFC 6874).

gistfile1.py

```python
# 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('^(?:(?:[0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\\.){3}(?:[0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-
ipv6_address = re.compile('^(?:(?:[0-9A-Fa-f]{1,4}:){6}(?:[0-9A-Fa-f]{1,4}:[0-9A-Fa-f]{1,4}|(?:(?:[0-9]|[1-9][0-9]|1[0-9]{2}|2[
ipv6_addrz = re.compile('^(?:(?:[0-9A-Fa-f]{1,4}:){6}(?:[0-9A-Fa-f]{1,4}:[0-9A-Fa-f]{1,4}|(?:(?:[0-9]|[1-9][0-9]|1[0-9]{2}|2[0-
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}|(?:(?:[0-9]|[1-9][0-9]|1[0
uri_reference = re.compile("^(?:([A-Za-z][A-Za-z0-9+\\-.]*):(?://((?:(?:(?:%[0-9A-Fa-f]{2}|[!$&'()*+,;=A-Za-z0-9\\-._~])|:)*@)?

# len(ipv4_address) == 111
# len(ipv6_address) == 1501
# len(ipv6_addrz) == 1541
# len(ipv6_address_or_addrz) == 1546
# len(uri_reference) == 4445
```

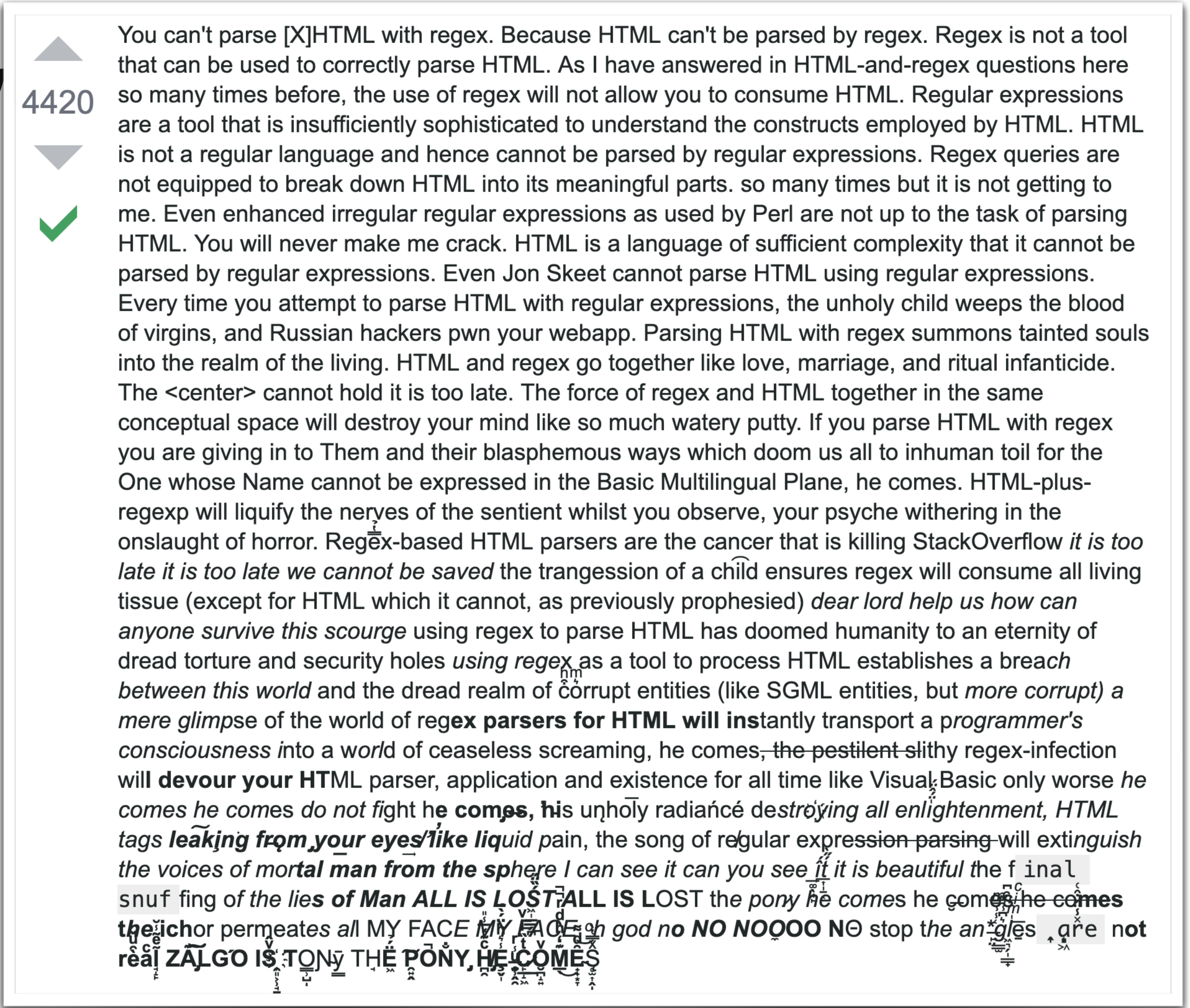
Expressive Power

Expressive Power

- Regular languages are limited
  - But DFAs are fast!

Expressive Pow

- Regular languages are
  - But DFAs are fast!

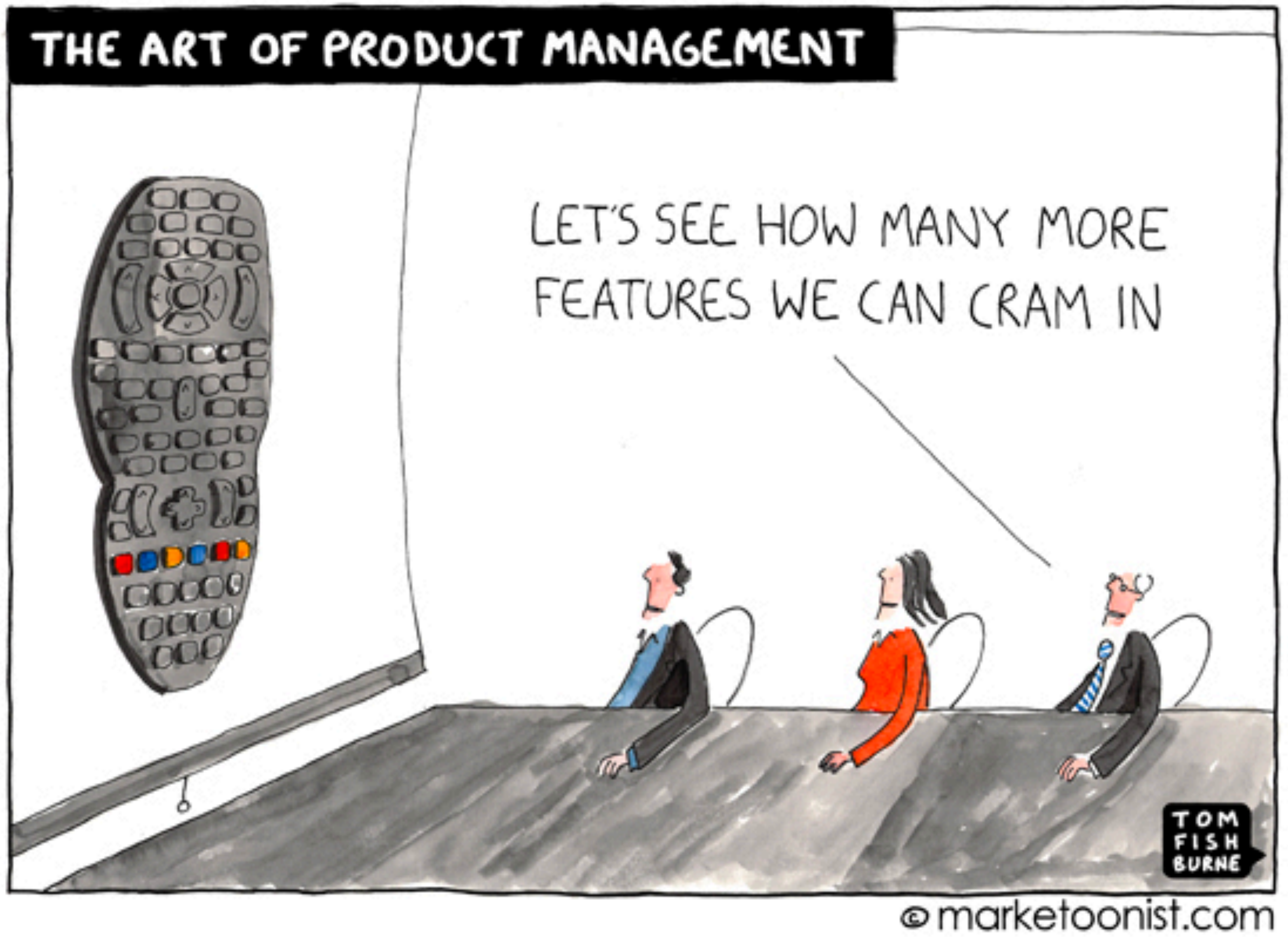


You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The <center> cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the trangession of a chіld ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a brea*ch between this world* and the dread realm of c͒ͪo͛ͫrrupt entities (like SGML entities, but *more corrupt) a mere glimp*se of the world of reg**ex parsers for HTML will ins**tantly transport a p*rogrammer's consciousness i*nto a wor*l*d of ceaseless screaming, he comes~~, the pestilent sl~~ithy regex-infection wil**l devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fi*ght h**e comes, hi**s unholy radiancé de*stroying all enlightenment, HTML tags* ***leaking from your eyes/like liq****uid p*ain, the song of re̸gular exp~~ression parsing~~ will exti*nguish the voices of mor****tal man from the sp****here I can see it can you see it it is beautiful t*he f `inal snuf` fing o*f the lie****s of Man ALL IS LOST ALL IS L***OST th*e pon*y he *com*es he c~~omes he co~~**mes the ich**or permeat*es al*l MY FACE *MY FACE ᵒh god n***o NO NOOOO N*Θ stop t*he an*gles `are` n**ot real ZALGO IS TONY THE PONY HE COMES**

Author: bobince https://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags

Expressive Power

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


THE ART OF PRODUCT MANAGEMENT
LET'S SEE HOW MANY MORE FEATURES WE CAN CRAM IN
TOM FISH BURNE
© marketoonist.com

Expressive Power

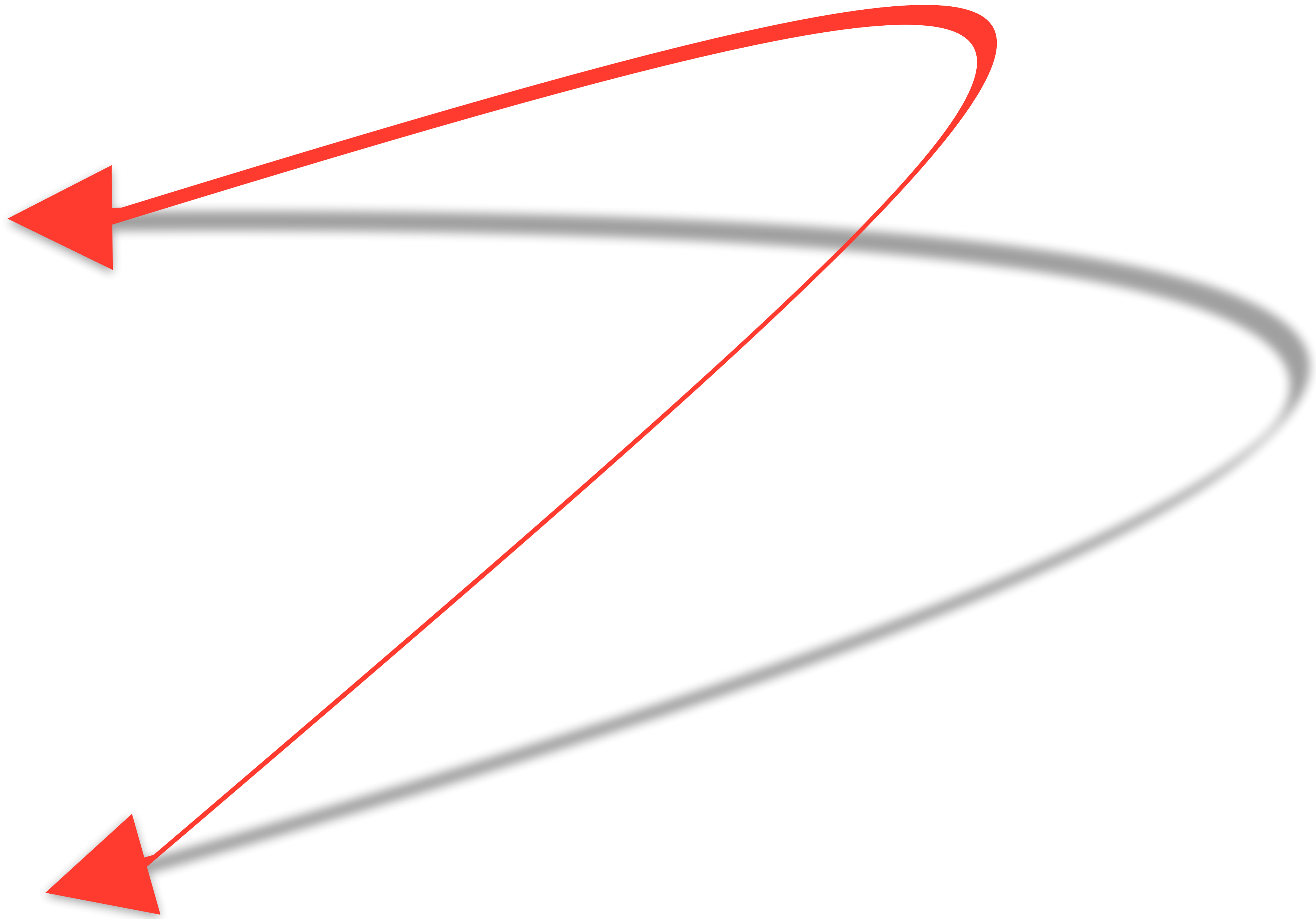
- Regular languages are limited
  - But DFAs are fast!
- Hence, feature creep
  - Backreferences
  - Conditionals
  - “Subroutines”, Perl6 grammars
  - Recursion
- Yet, static analysis needed!
  - Challenge: # dialects x # impls

THE ART OF PRODUCT MANAGEMENT
LET'S SEE HOW MANY MORE FEATURES WE CAN CRAM IN
TOM FISH BURNE

© marketoonist.com

Expressive Power

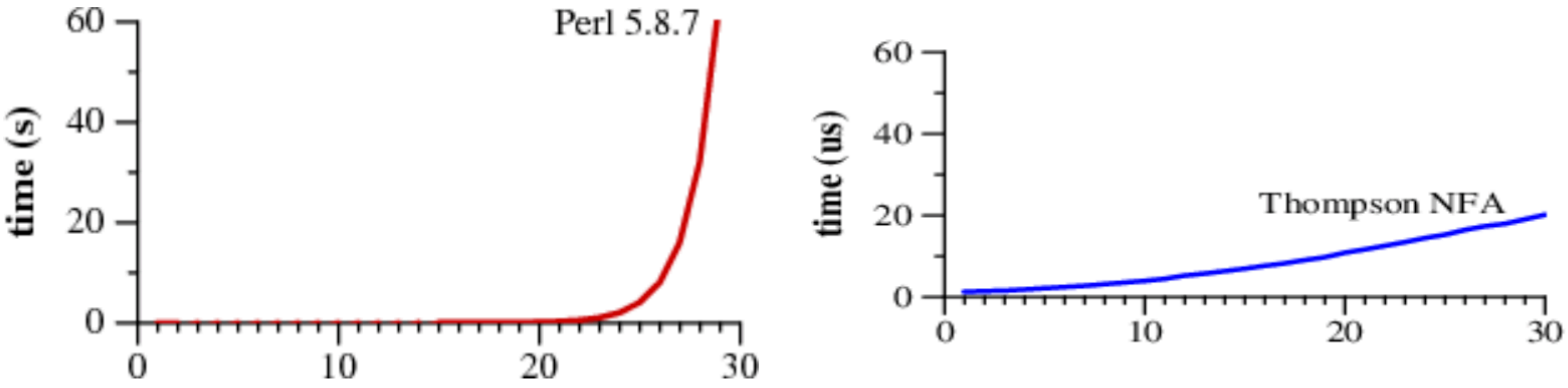
- Regular languages are limited
  - But DFAs are fast!
- Hence, feature creep
  - Backreferences
  - Conditionals
  - “Subroutines”, Perl6 grammars
  - Recursion
- Yet, static analysis needed!
  - Challenge: # dialects x # impls



Implementation Issues

Implementation Issues

- Exponential time algorithm is by far the most common



Time to match $\texttt{a?}^n\texttt{a}^n$ against $\texttt{a}^n$

Russ Cox https://swtch.com/~rsc/regexp/regexp1.html

Implementation Issues

- Exponential time algorithm is by far the most common

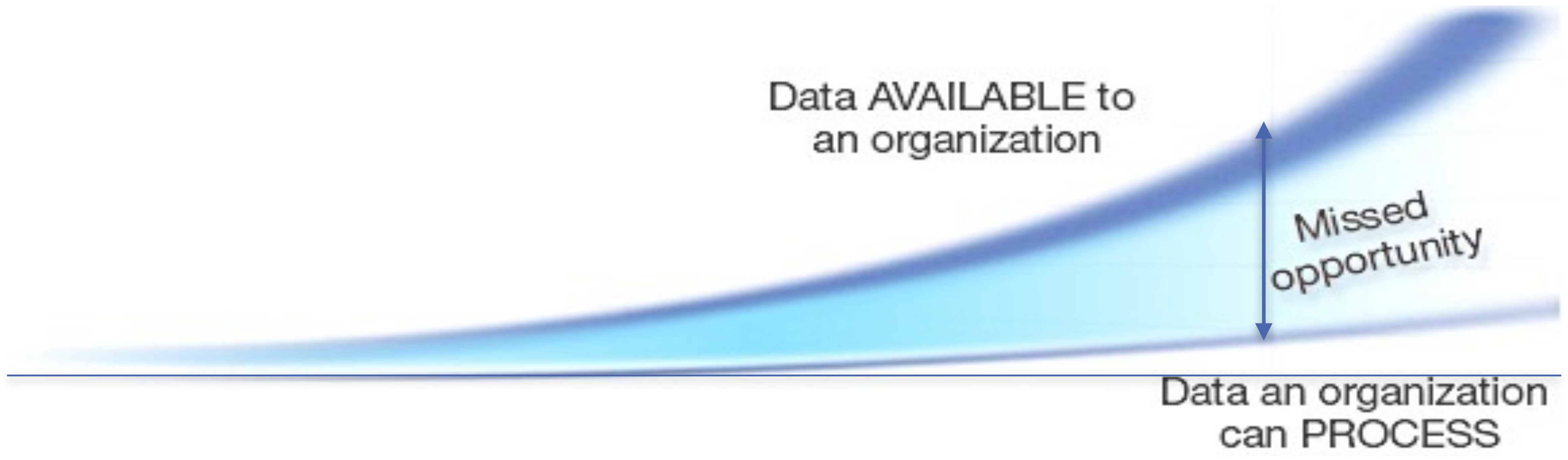
- Most regex are embedded DSLs
  - Syntax issues (escaping)
  - Type issues
  - Requires scaffolding to write/debug regex
    - Less than 17% are tested, most lacking both positive & negative tests [Wang, Stolee ESEC/FSE ’18]

Why work on this?

1

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

IBM



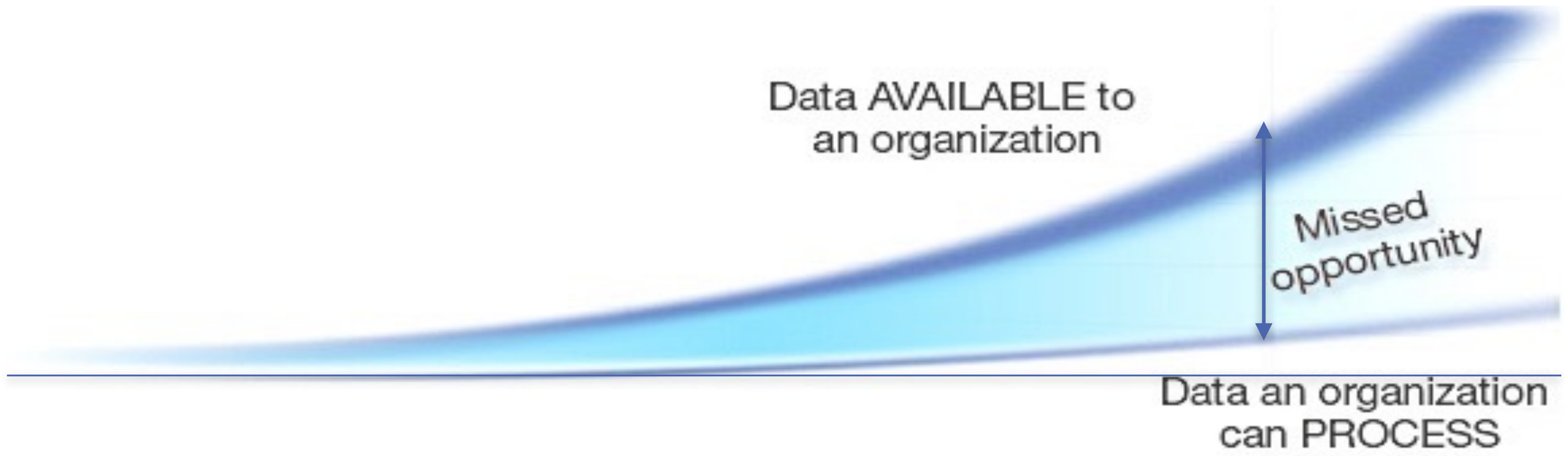
Estimates are that less than 0.5% of data is ever analyzed.

Antonio Regalado
MIT Technology Review

Why work on this?

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

IBM



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

Antonio Regalado
MIT Technology Review

2 Regex use does not scale (# exps, # people, project lifetime)

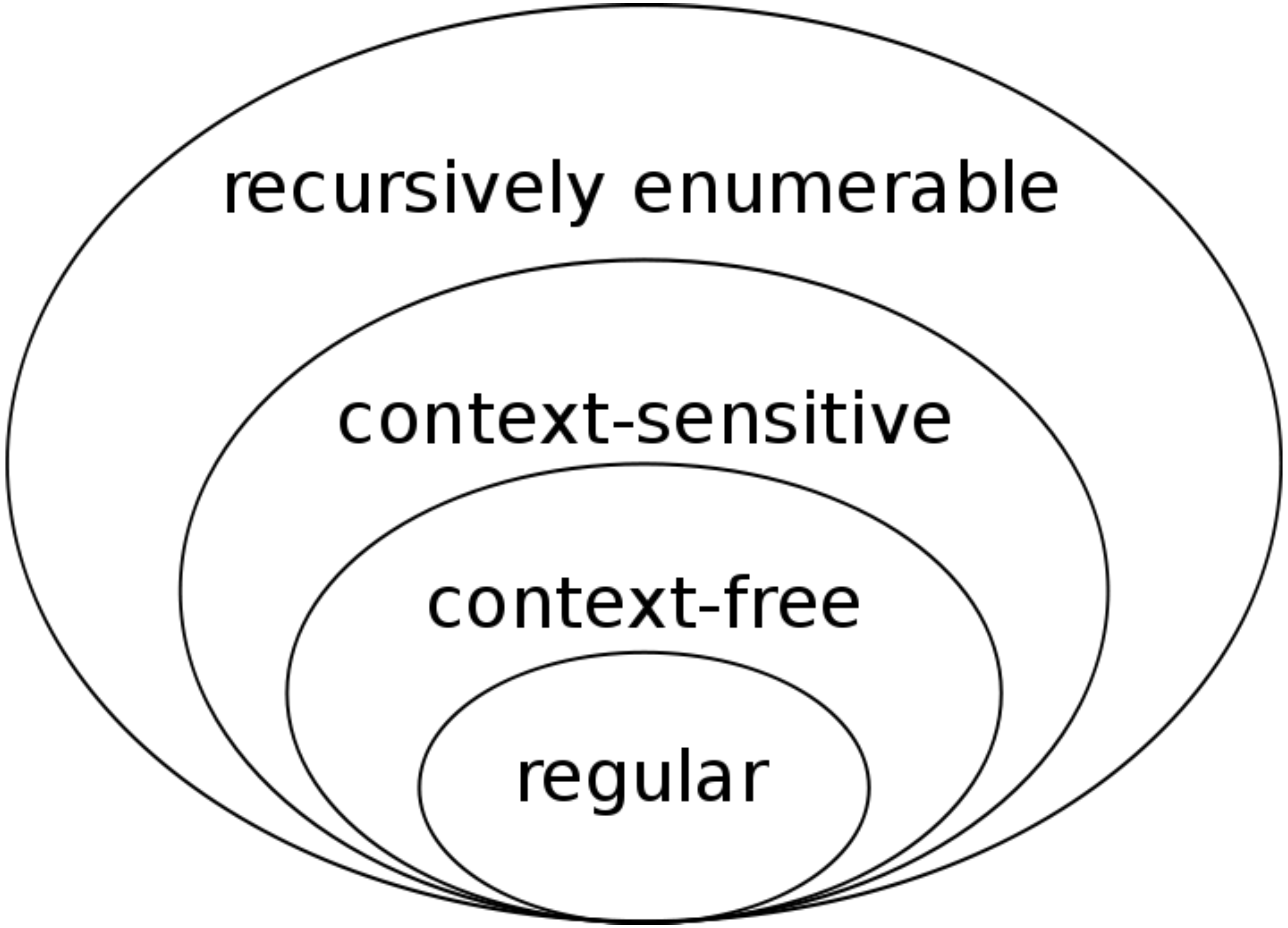
Rosie Pattern Language

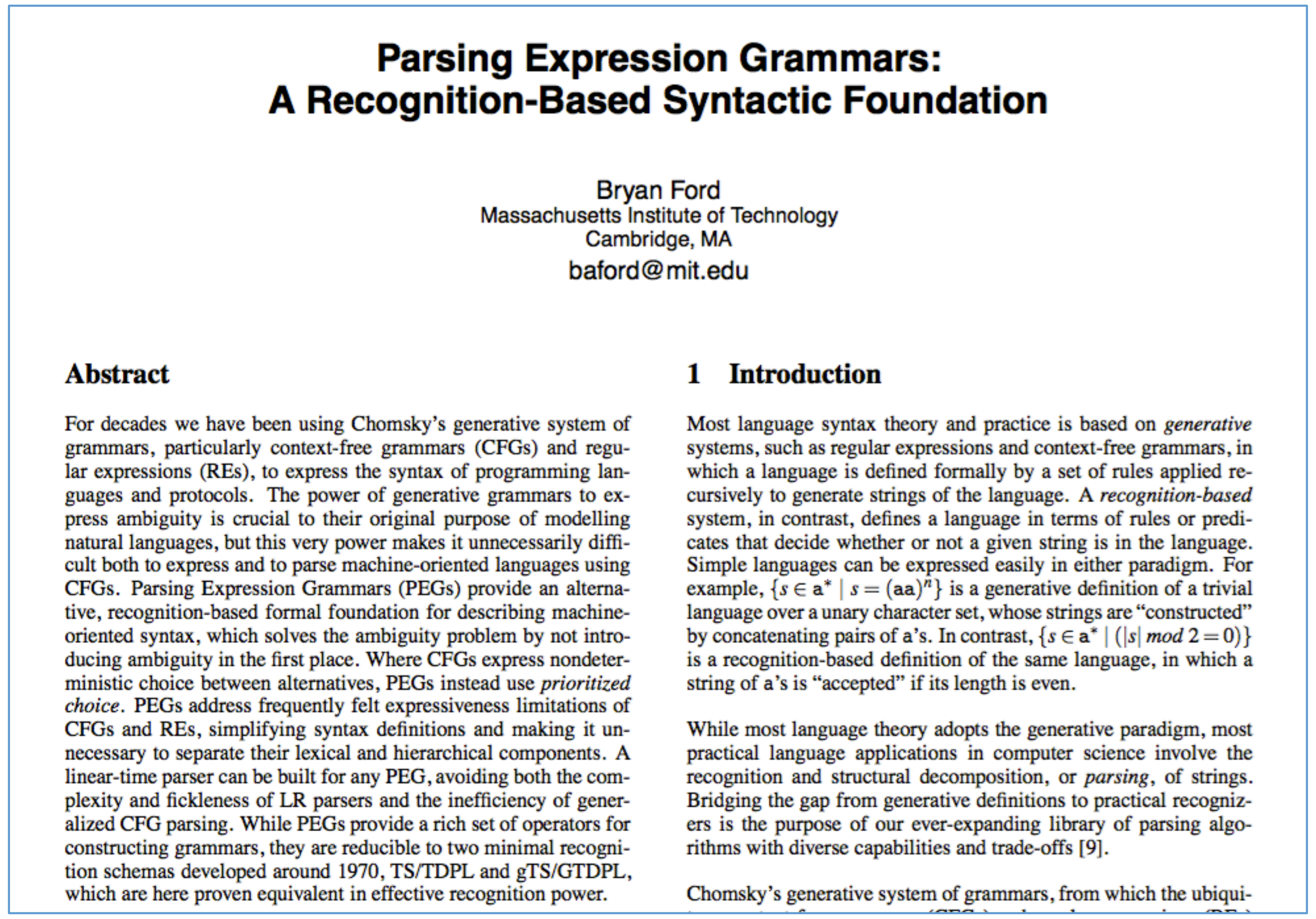
“All progress depends on the unreasonable [woman]”

George Bernard Shaw, paraphrased

Formal basis

Chomsky hierarchy





Parsing Expression Grammars: A Recognition-Based Syntactic Foundation

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

Abstract

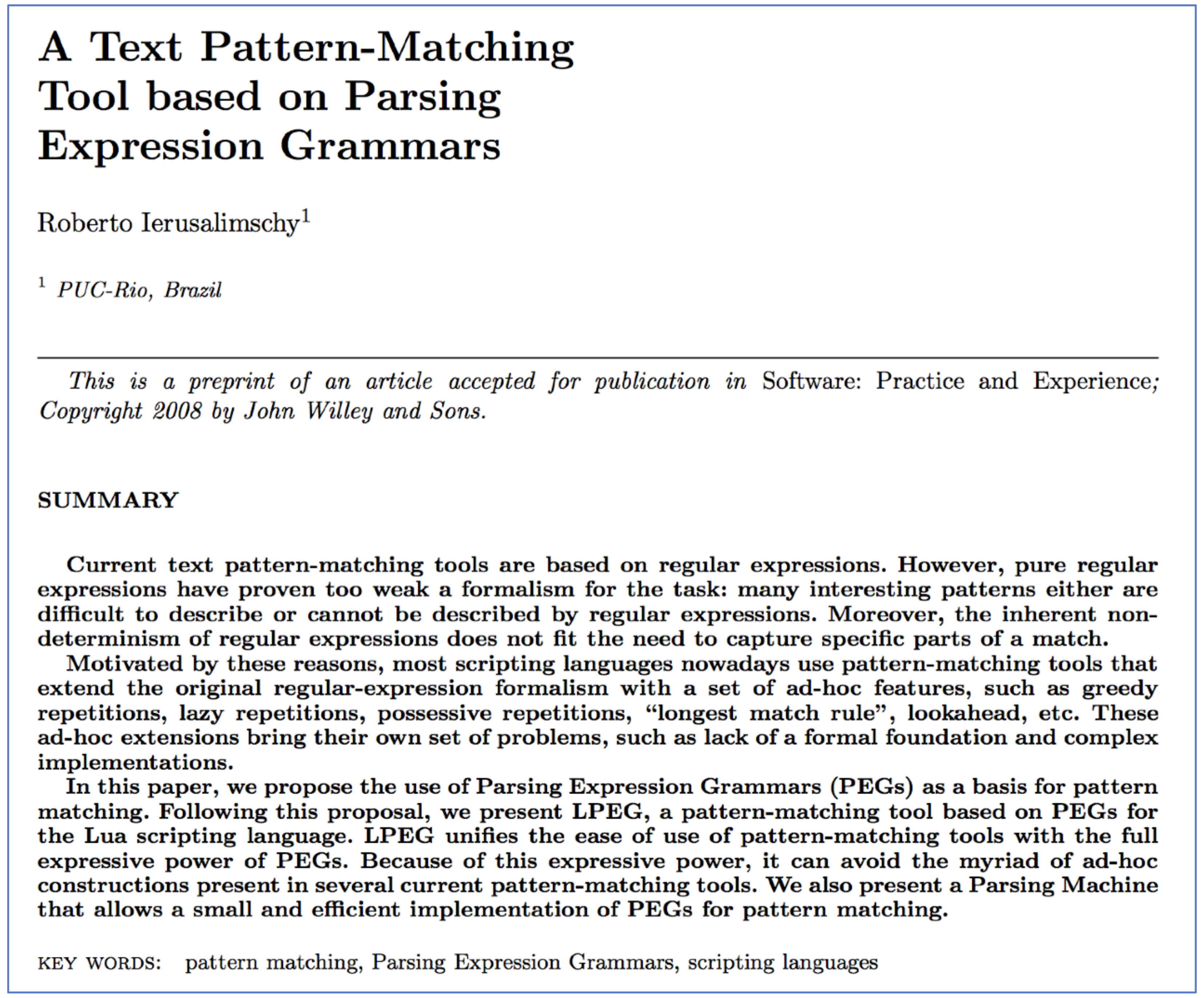
For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), to express the syntax of programming languages and protocols. The power of generative grammars to express ambiguity is crucial to their original purpose of modelling natural languages, but this very power makes it unnecessarily difficult both to express and to parse machine-oriented languages using CFGs. Parsing Expression Grammars (PEGs) provide an alternative, recognition-based formal foundation for describing machine-oriented syntax, which solves the ambiguity problem by not introducing ambiguity in the first place. Where CFGs express nondeterministic choice between alternatives, PEGs instead use *prioritized choice*. PEGs address frequently felt expressiveness limitations of CFGs and REs, simplifying syntax definitions and making it unnecessary to separate their lexical and hierarchical components. A linear-time parser can be built for any PEG, avoiding both the complexity and fickleness of LR parsers and the inefficiency of generalized CFG parsing. While PEGs provide a rich set of operators for constructing grammars, they are reducible to two minimal recognition schemas developed around 1970, TS/TDPL and gTS/GTDPL, which are here proven equivalent in effective recognition power.

1 Introduction

Most language syntax theory and practice is based on *generative* systems, such as regular expressions and context-free grammars, in which a language is defined formally by a set of rules applied recursively to generate strings of the language. A *recognition-based* system, in contrast, defines a language in terms of rules or predicates that decide whether or not a given string is in the language. Simple languages can be expressed easily in either paradigm. For example, $\{s \in \mathtt{a}^* \mid s = (\mathtt{aa})^n\}$ is a generative definition of a trivial language over a unary character set, whose strings are "constructed" by concatenating pairs of a's. In contrast, $\{s \in \mathtt{a}^* \mid (|s| \; mod \; 2 = 0)\}$ is a recognition-based definition of the same language, in which a string of a's is "accepted" if its length is even.

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. Bridging the gap from generative definitions to practical recognizers is the purpose of our ever-expanding library of parsing algorithms with diverse capabilities and trade-offs [9].

Chomsky's generative system of grammars, from which the ubiqui-



A Text Pattern-Matching Tool based on Parsing Expression Grammars

Roberto Ierusalimschy[1]

[1] *PUC-Rio, Brazil*

This is a preprint of an article accepted for publication in Software: Practice and Experience; *Copyright 2008 by John Willey and Sons.*

SUMMARY

Current text pattern-matching tools are based on regular expressions. However, pure regular expressions have proven too weak a formalism for the task: many interesting patterns either are difficult to describe or cannot be described by regular expressions. Moreover, the inherent non-determinism of regular expressions does not fit the need to capture specific parts of a match.

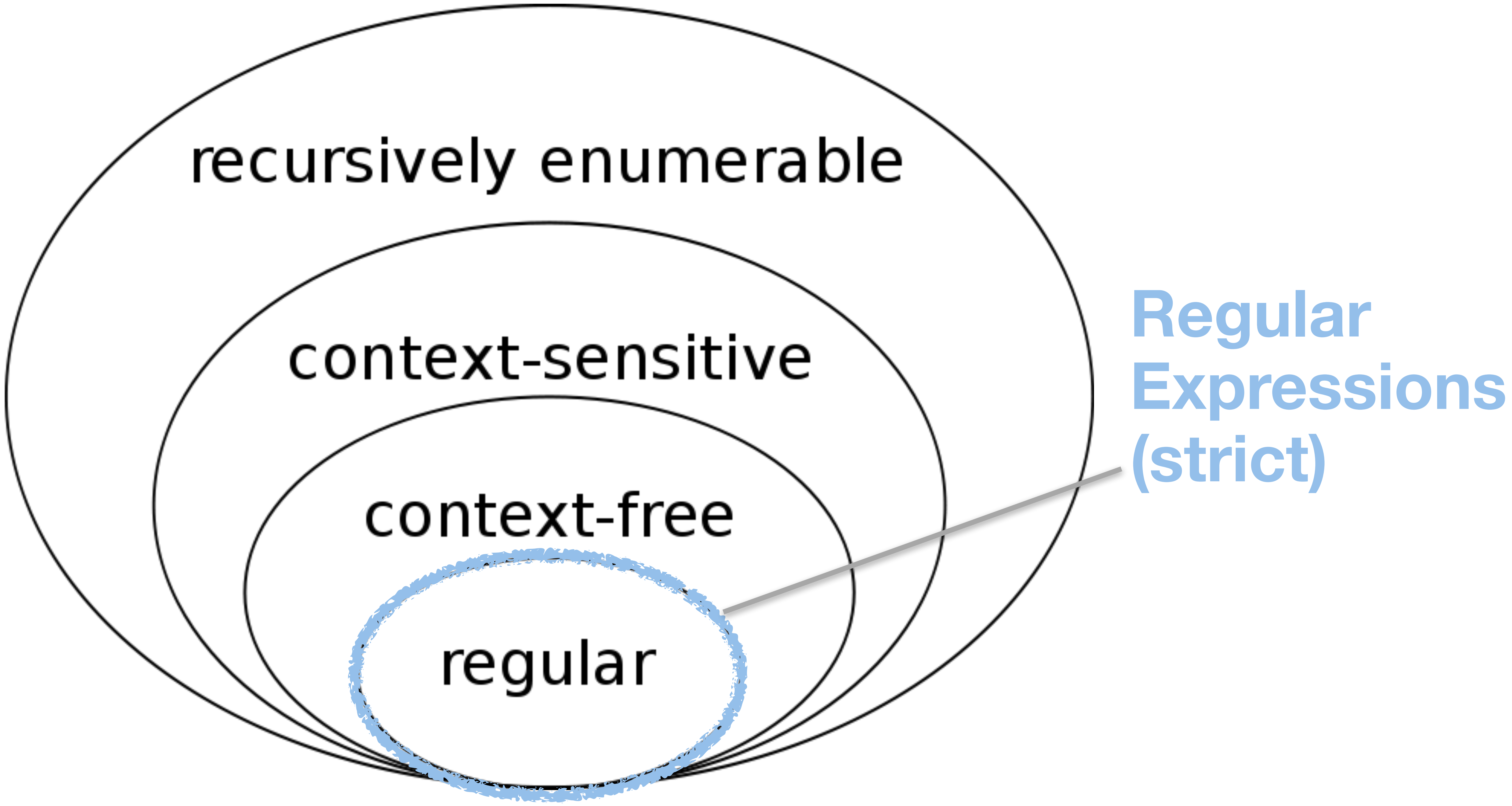
Motivated by these reasons, most scripting languages nowadays use pattern-matching tools that extend the original regular-expression formalism with a set of ad-hoc features, such as greedy repetitions, lazy repetitions, possessive repetitions, "longest match rule", lookahead, etc. These ad-hoc extensions bring their own set of problems, such as lack of a formal foundation and complex implementations.

In this paper, we propose the use of Parsing Expression Grammars (PEGs) as a basis for pattern matching. Following this proposal, we present LPEG, a pattern-matching tool based on PEGs for the Lua scripting language. LPEG unifies the ease of use of pattern-matching tools with the full expressive power of PEGs. Because of this expressive power, it can avoid the myriad of ad-hoc constructions present in several current pattern-matching tools. We also present a Parsing Machine that allows a small and efficient implementation of PEGs for pattern matching.

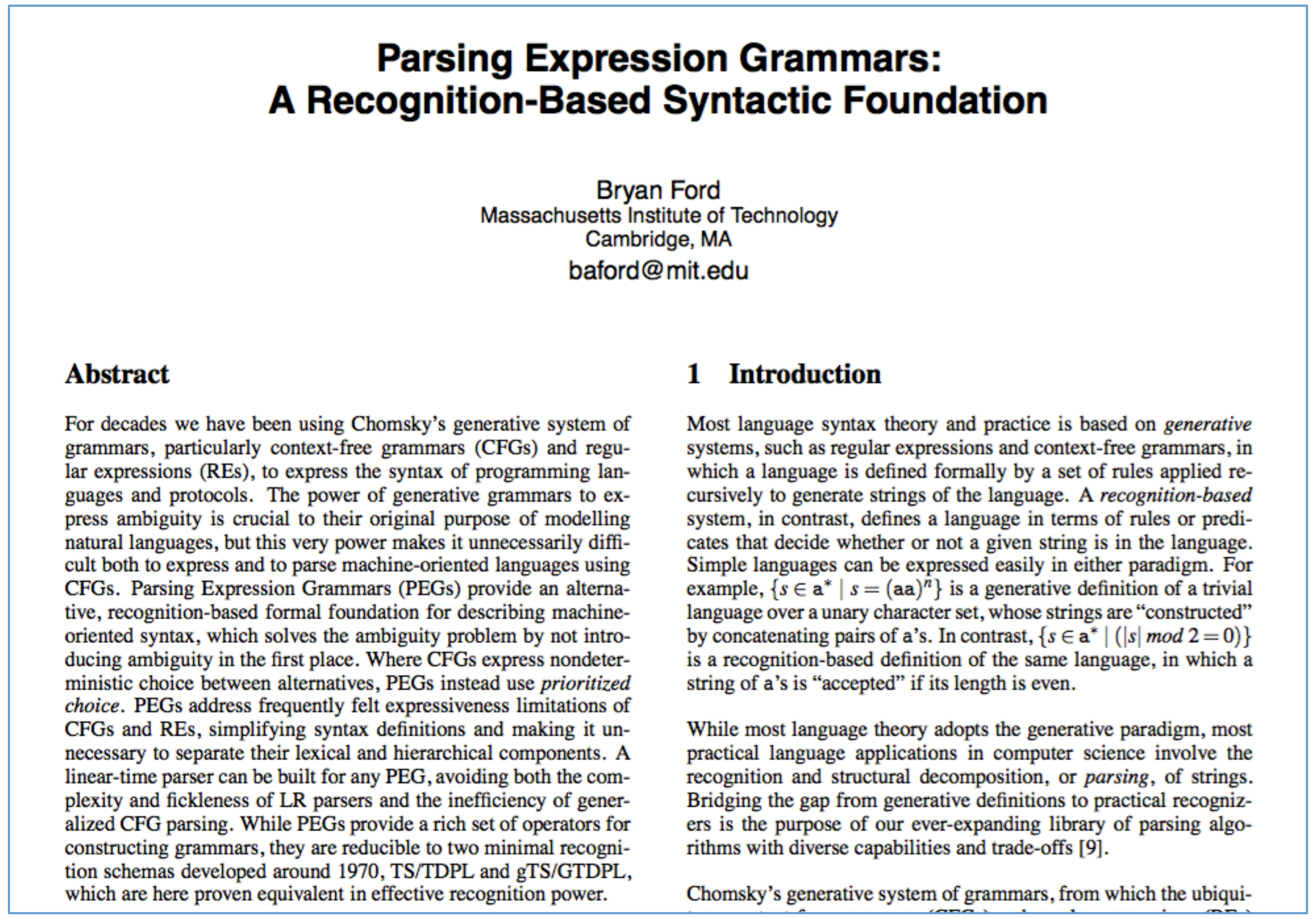
KEY WORDS: pattern matching, Parsing Expression Grammars, scripting languages

Formal basis

Chomsky hierarchy



By J. Finkelstein - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=9405226



Parsing Expression Grammars: A Recognition-Based Syntactic Foundation

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

Abstract

For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), to express the syntax of programming languages and protocols. The power of generative grammars to express ambiguity is crucial to their original purpose of modelling natural languages, but this very power makes it unnecessarily difficult both to express and to parse machine-oriented languages using CFGs. Parsing Expression Grammars (PEGs) provide an alternative, recognition-based formal foundation for describing machine-oriented syntax, which solves the ambiguity problem by not introducing ambiguity in the first place. Where CFGs express nondeterministic choice between alternatives, PEGs instead use *prioritized choice*. PEGs address frequently felt expressiveness limitations of CFGs and REs, simplifying syntax definitions and making it unnecessary to separate their lexical and hierarchical components. A linear-time parser can be built for any PEG, avoiding both the complexity and fickleness of LR parsers and the inefficiency of generalized CFG parsing. While PEGs provide a rich set of operators for constructing grammars, they are reducible to two minimal recognition schemas developed around 1970, TS/TDPL and gTS/GTDPL, which are here proven equivalent in effective recognition power.

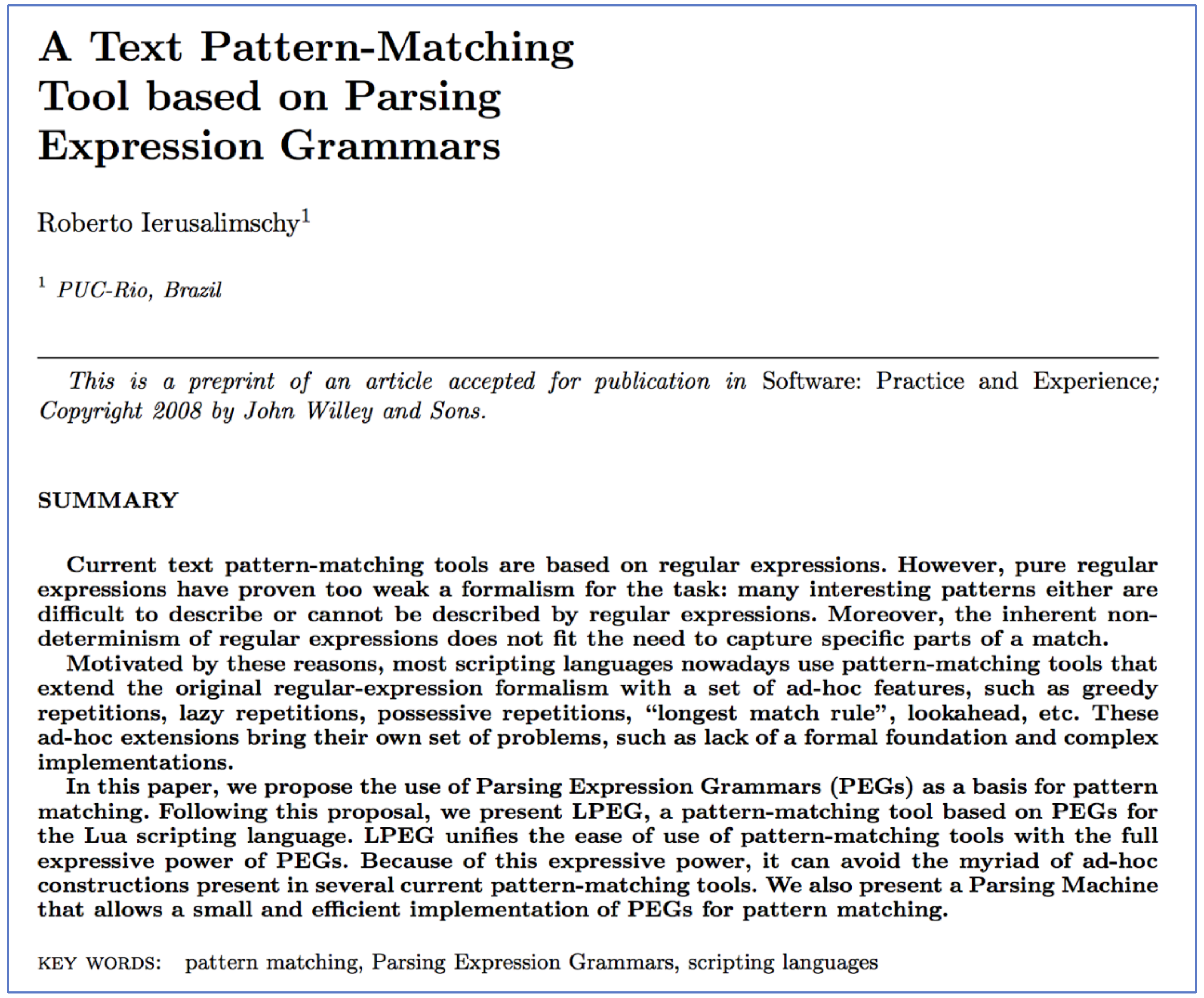
1 Introduction

Most language syntax theory and practice is based on *generative* systems, such as regular expressions and context-free grammars, in which a language is defined formally by a set of rules applied recursively to generate strings of the language. A *recognition-based* system, in contrast, defines a language in terms of rules or predicates that decide whether or not a given string is in the language. Simple languages can be expressed easily in either paradigm. For example, $\{s \in \mathtt{a}^* \mid s = (\mathtt{aa})^n\}$ is a generative definition of a trivial language over a unary character set, whose strings are "constructed" by concatenating pairs of a's. In contrast, $\{s \in \mathtt{a}^* \mid (|s| \; mod \; 2 = 0)\}$ is a recognition-based definition of the same language, in which a string of a's is "accepted" if its length is even.

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. Bridging the gap from generative definitions to practical recognizers is the purpose of our ever-expanding library of parsing algorithms with diverse capabilities and trade-offs [9].

Chomsky's generative system of grammars, from which the ubiqui-

⋮



A Text Pattern-Matching Tool based on Parsing Expression Grammars

Roberto Ierusalimschy[1]

[1] *PUC-Rio, Brazil*

This is a preprint of an article accepted for publication in Software: Practice and Experience; *Copyright 2008 by John Willey and Sons.*

SUMMARY

Current text pattern-matching tools are based on regular expressions. However, pure regular expressions have proven too weak a formalism for the task: many interesting patterns either are difficult to describe or cannot be described by regular expressions. Moreover, the inherent non-determinism of regular expressions does not fit the need to capture specific parts of a match.

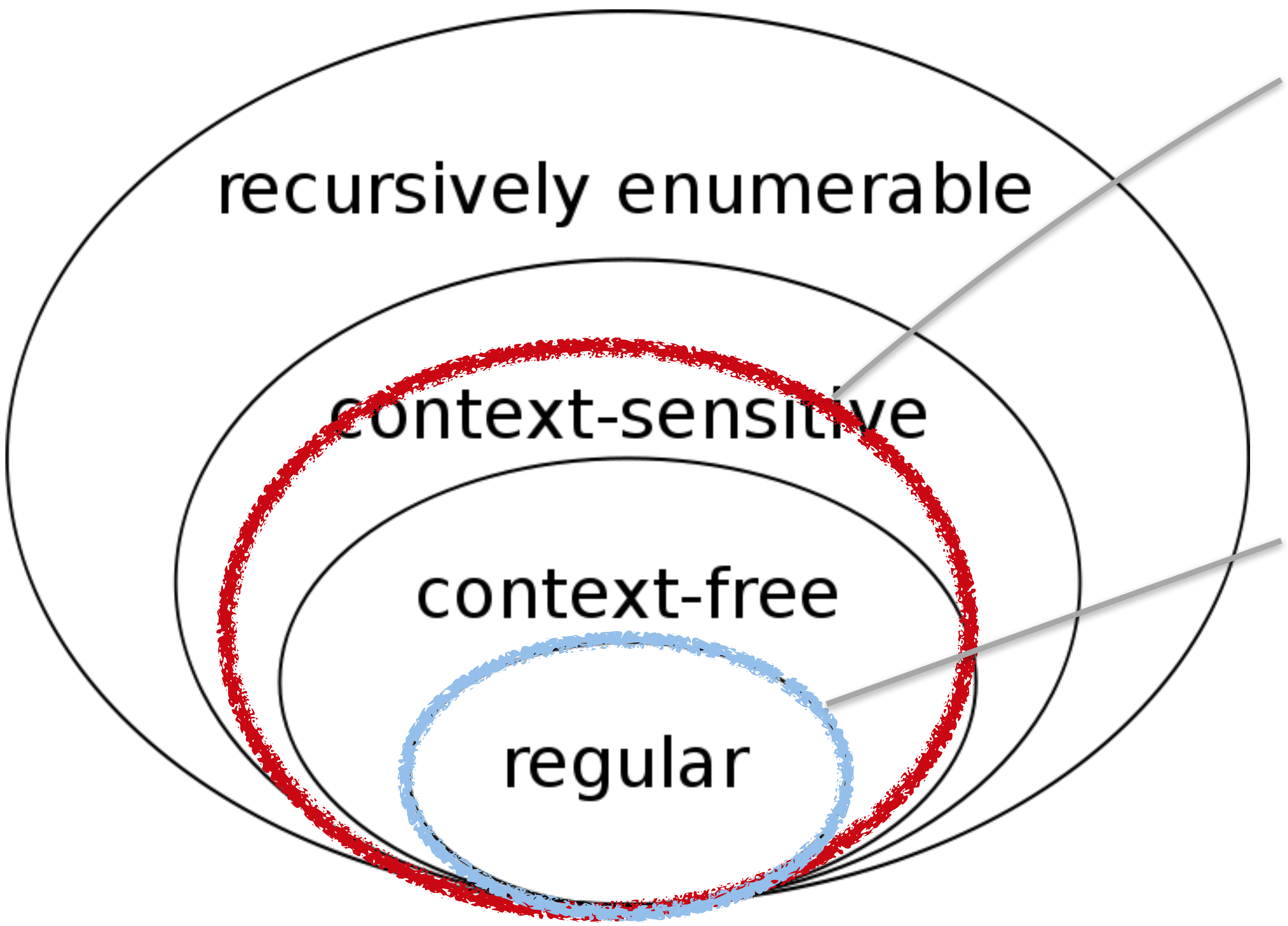
Motivated by these reasons, most scripting languages nowadays use pattern-matching tools that extend the original regular-expression formalism with a set of ad-hoc features, such as greedy repetitions, lazy repetitions, possessive repetitions, "longest match rule", lookahead, etc. These ad-hoc extensions bring their own set of problems, such as lack of a formal foundation and complex implementations.

In this paper, we propose the use of Parsing Expression Grammars (PEGs) as a basis for pattern matching. Following this proposal, we present LPEG, a pattern-matching tool based on PEGs for the Lua scripting language. LPEG unifies the ease of use of pattern-matching tools with the full expressive power of PEGs. Because of this expressive power, it can avoid the myriad of ad-hoc constructions present in several current pattern-matching tools. We also present a Parsing Machine that allows a small and efficient implementation of PEGs for pattern matching.

KEY WORDS: pattern matching, Parsing Expression Grammars, scripting languages

Formal basis

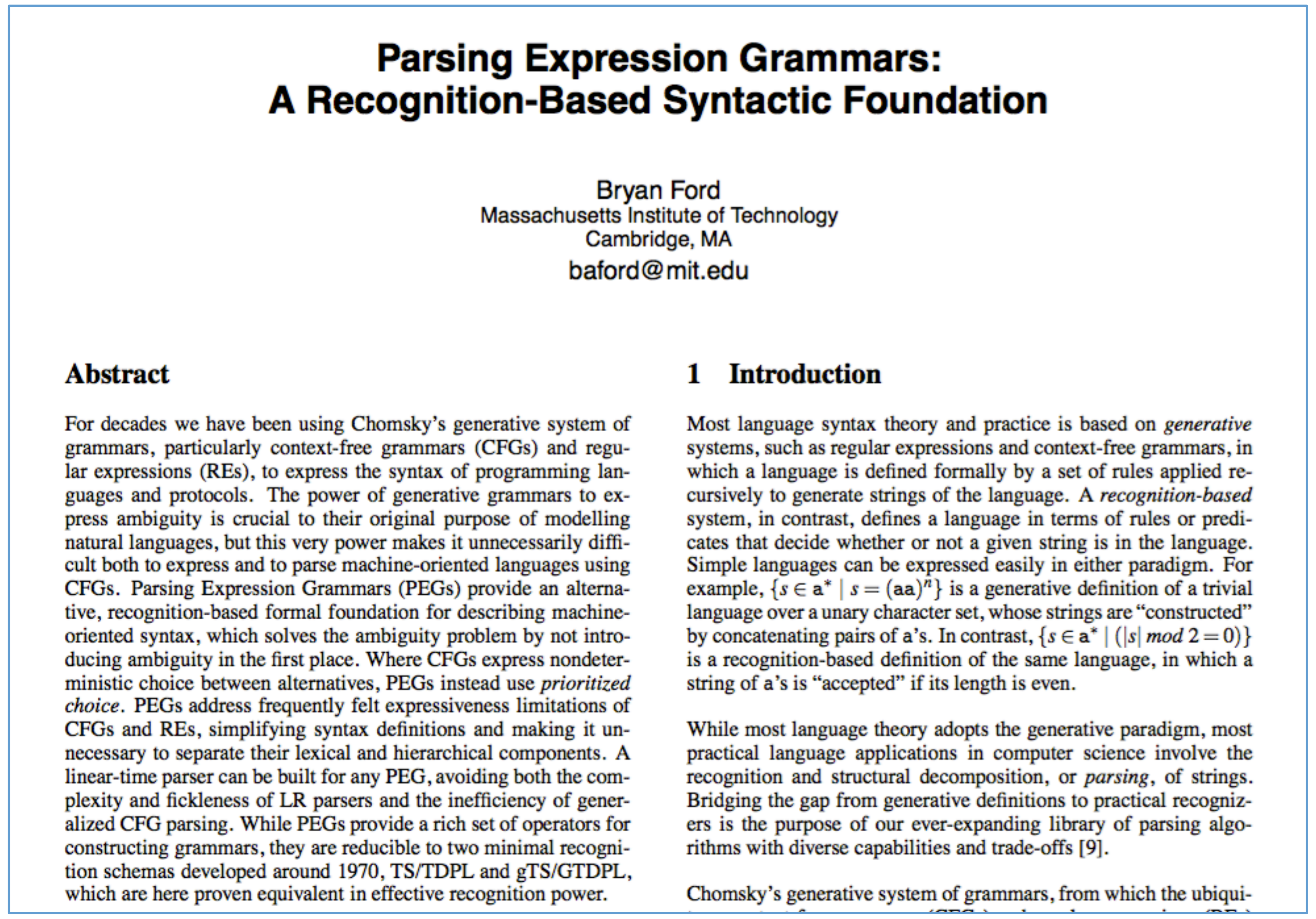
Chomsky hierarchy



Rosie Pattern Language
(and all PEG grammars)

Regular Expressions (strict)

By J. Finkelstein - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=9405226



Parsing Expression Grammars: A Recognition-Based Syntactic Foundation

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

Abstract

For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), to express the syntax of programming languages and protocols. The power of generative grammars to express ambiguity is crucial to their original purpose of modelling natural languages, but this very power makes it unnecessarily difficult both to express and to parse machine-oriented languages using CFGs. Parsing Expression Grammars (PEGs) provide an alternative, recognition-based formal foundation for describing machine-oriented syntax, which solves the ambiguity problem by not introducing ambiguity in the first place. Where CFGs express nondeterministic choice between alternatives, PEGs instead use *prioritized choice*. PEGs address frequently felt expressiveness limitations of CFGs and REs, simplifying syntax definitions and making it unnecessary to separate their lexical and hierarchical components. A linear-time parser can be built for any PEG, avoiding both the complexity and fickleness of LR parsers and the inefficiency of generalized CFG parsing. While PEGs provide a rich set of operators for constructing grammars, they are reducible to two minimal recognition schemas developed around 1970, TS/TDPL and gTS/GTDPL, which are here proven equivalent in effective recognition power.

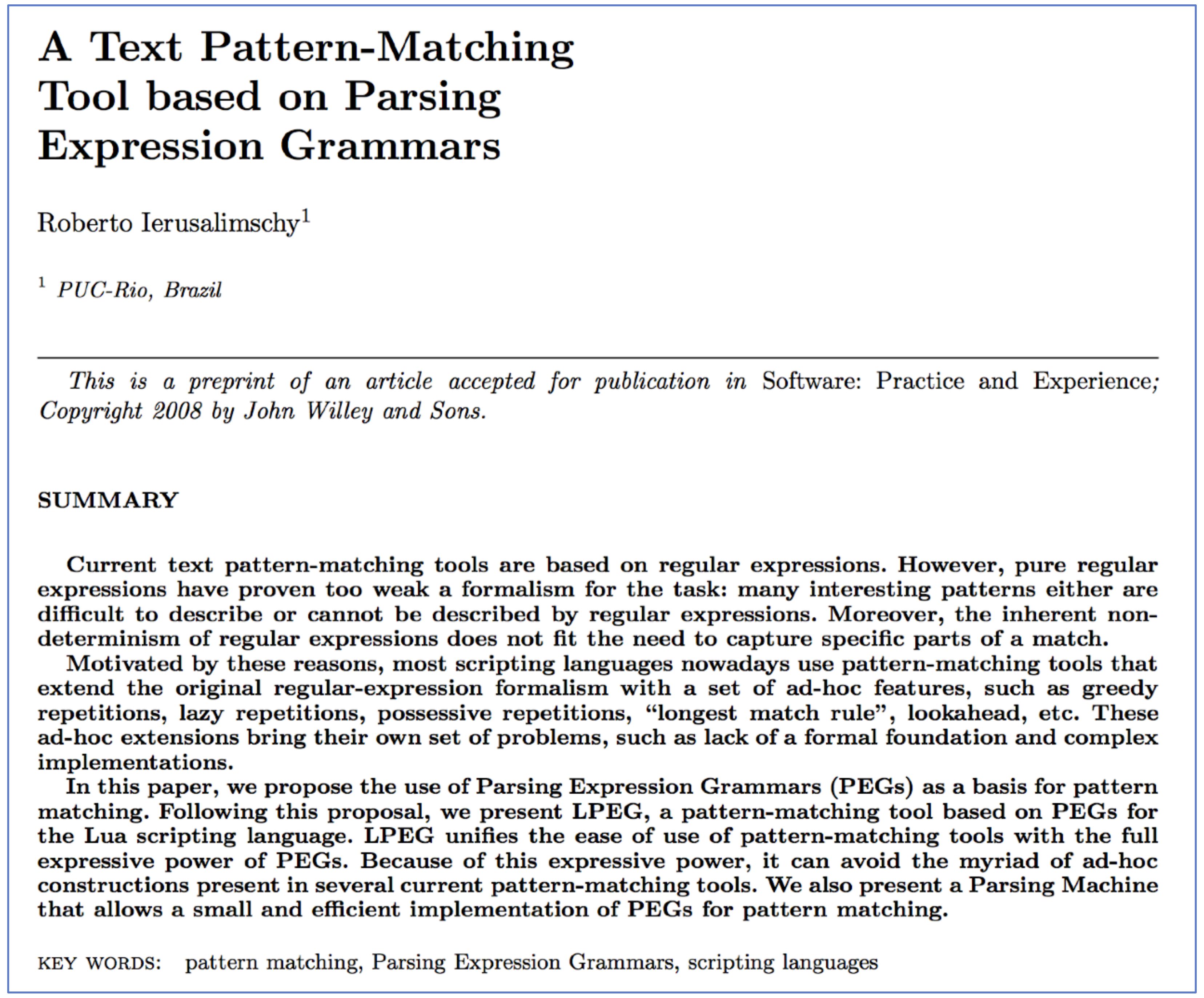
1 Introduction

Most language syntax theory and practice is based on *generative* systems, such as regular expressions and context-free grammars, in which a language is defined formally by a set of rules applied recursively to generate strings of the language. A *recognition-based* system, in contrast, defines a language in terms of rules or predicates that decide whether or not a given string is in the language. Simple languages can be expressed easily in either paradigm. For example, $\{s \in \mathtt{a}^* \mid s = (\mathtt{aa})^n\}$ is a generative definition of a trivial language over a unary character set, whose strings are "constructed" by concatenating pairs of a's. In contrast, $\{s \in \mathtt{a}^* \mid (|s| \; mod \; 2 = 0)\}$ is a recognition-based definition of the same language, in which a string of a's is "accepted" if its length is even.

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. Bridging the gap from generative definitions to practical recognizers is the purpose of our ever-expanding library of parsing algorithms with diverse capabilities and trade-offs [9].

Chomsky's generative system of grammars, from which the ubiqui-

⋮



A Text Pattern-Matching Tool based on Parsing Expression Grammars

Roberto Ierusalimschy[1]

[1] *PUC-Rio, Brazil*

This is a preprint of an article accepted for publication in Software: Practice and Experience; *Copyright 2008 by John Willey and Sons.*

SUMMARY

Current text pattern-matching tools are based on regular expressions. However, pure regular expressions have proven too weak a formalism for the task: many interesting patterns either are difficult to describe or cannot be described by regular expressions. Moreover, the inherent non-determinism of regular expressions does not fit the need to capture specific parts of a match.

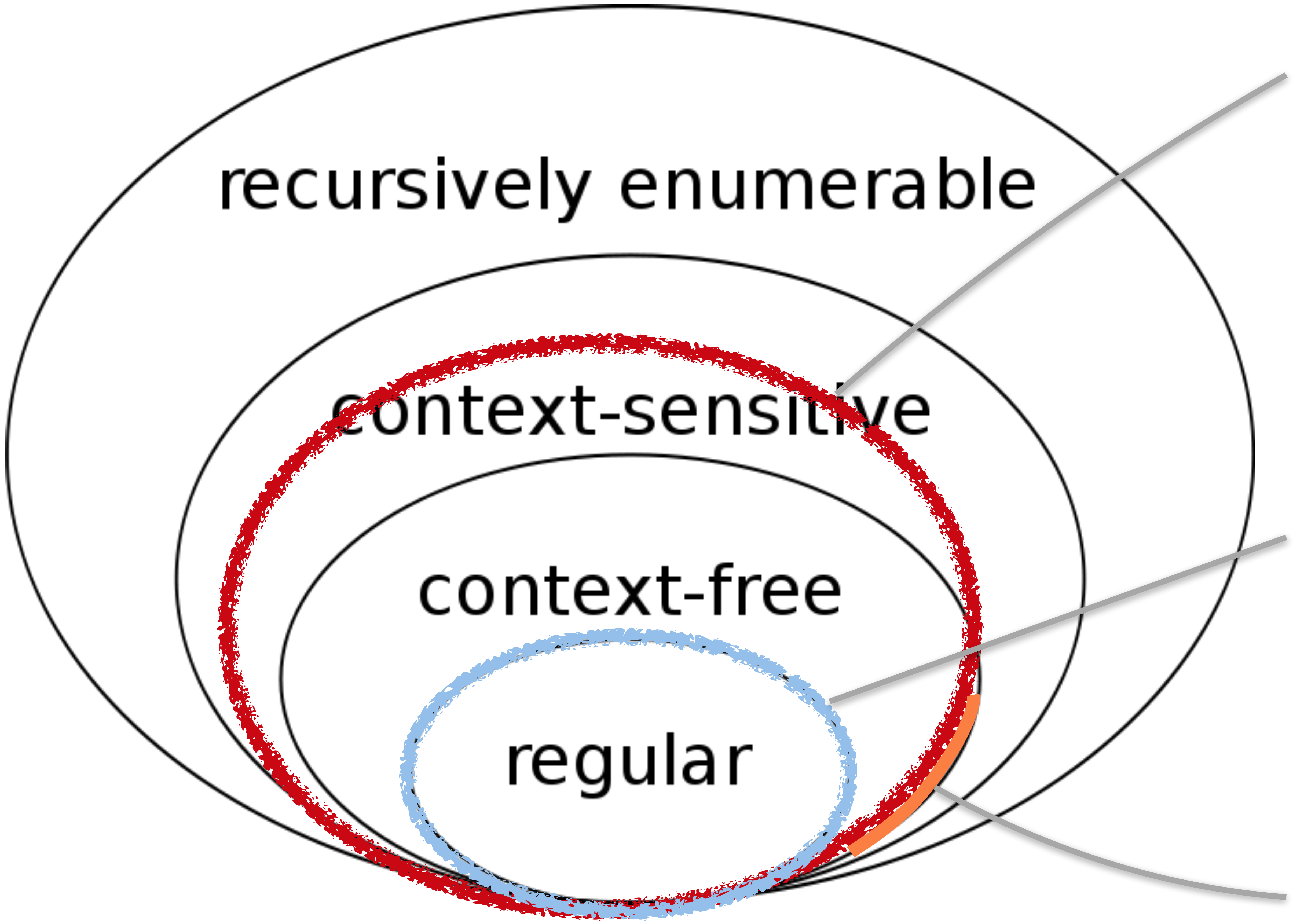
Motivated by these reasons, most scripting languages nowadays use pattern-matching tools that extend the original regular-expression formalism with a set of ad-hoc features, such as greedy repetitions, lazy repetitions, possessive repetitions, "longest match rule", lookahead, etc. These ad-hoc extensions bring their own set of problems, such as lack of a formal foundation and complex implementations.

In this paper, we propose the use of Parsing Expression Grammars (PEGs) as a basis for pattern matching. Following this proposal, we present LPEG, a pattern-matching tool based on PEGs for the Lua scripting language. LPEG unifies the ease of use of pattern-matching tools with the full expressive power of PEGs. Because of this expressive power, it can avoid the myriad of ad-hoc constructions present in several current pattern-matching tools. We also present a Parsing Machine that allows a small and efficient implementation of PEGs for pattern matching.

KEY WORDS: pattern matching, Parsing Expression Grammars, scripting languages

Formal basis

Chomsky hierarchy

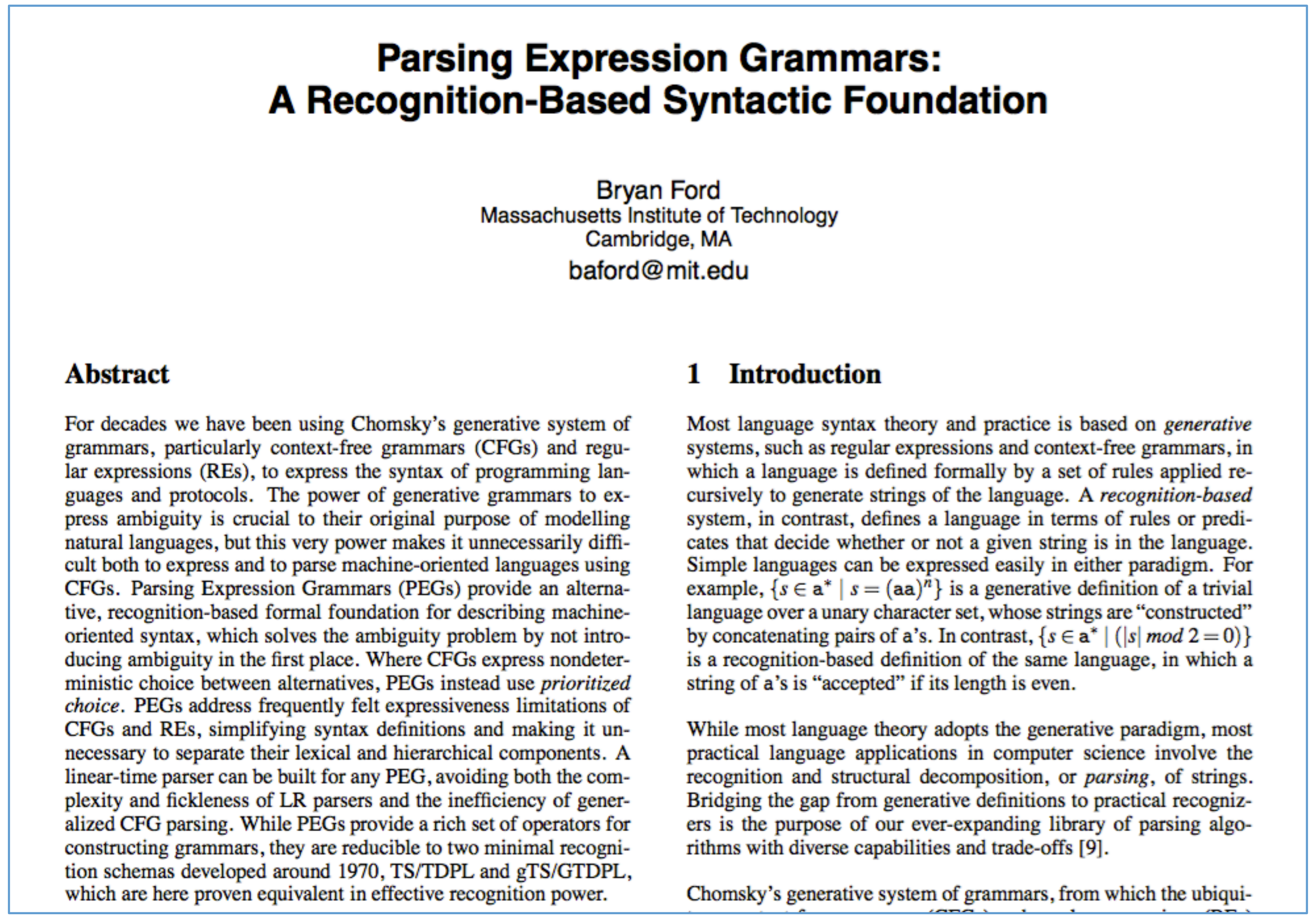


Rosie Pattern Language
(and all PEG grammars)

Regular Expressions (strict)

Open Question: PEG > CFG

By J. Finkelstein - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=9405226



Parsing Expression Grammars: A Recognition-Based Syntactic Foundation

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

Abstract

For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), to express the syntax of programming languages and protocols. The power of generative grammars to express ambiguity is crucial to their original purpose of modelling natural languages, but this very power makes it unnecessarily difficult both to express and to parse machine-oriented languages using CFGs. Parsing Expression Grammars (PEGs) provide an alternative, recognition-based formal foundation for describing machine-oriented syntax, which solves the ambiguity problem by not introducing ambiguity in the first place. Where CFGs express nondeterministic choice between alternatives, PEGs instead use *prioritized choice*. PEGs address frequently felt expressiveness limitations of CFGs and REs, simplifying syntax definitions and making it unnecessary to separate their lexical and hierarchical components. A linear-time parser can be built for any PEG, avoiding both the complexity and fickleness of LR parsers and the inefficiency of generalized CFG parsing. While PEGs provide a rich set of operators for constructing grammars, they are reducible to two minimal recognition schemas developed around 1970, TS/TDPL and gTS/GTDPL, which are here proven equivalent in effective recognition power.

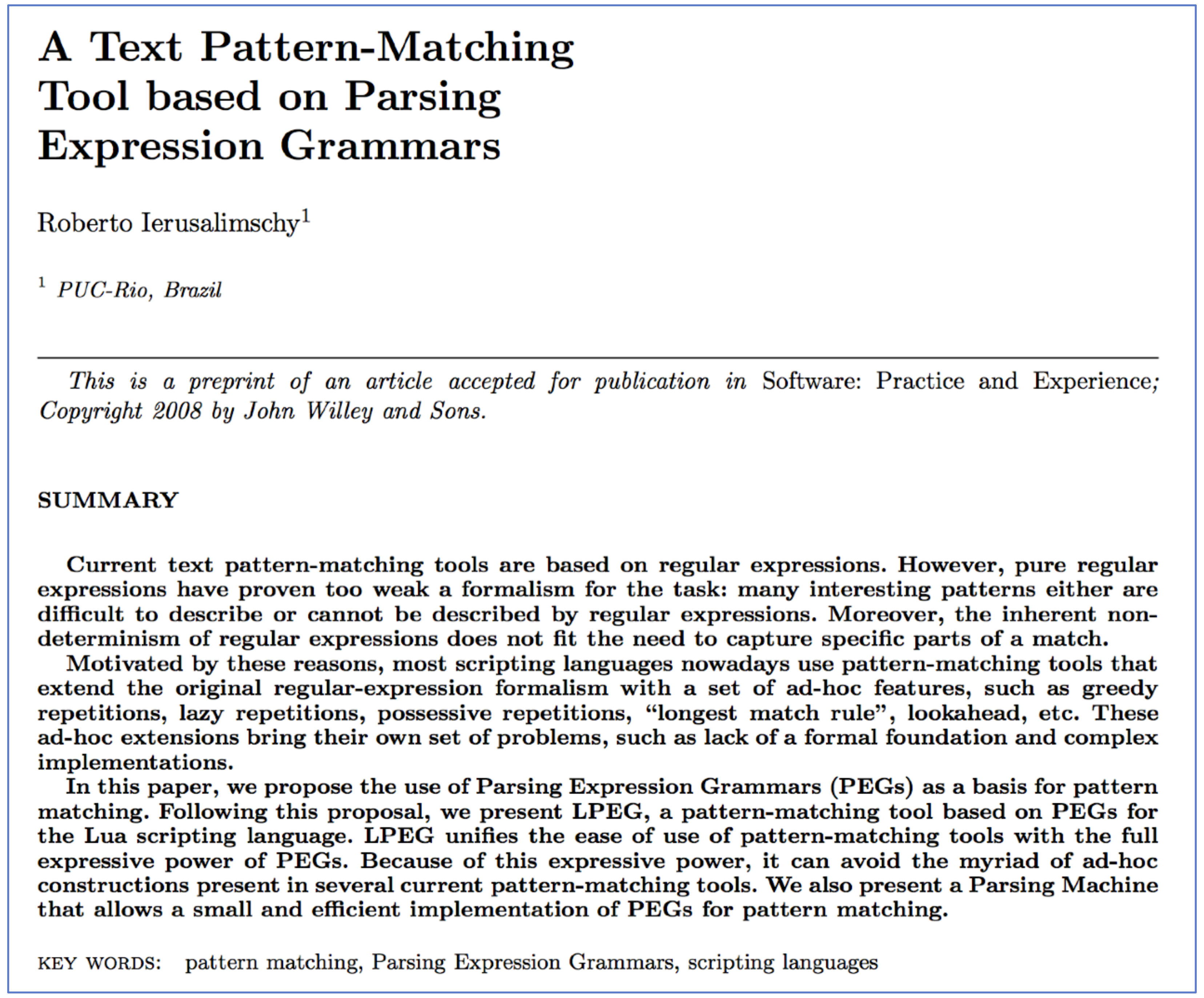
1 Introduction

Most language syntax theory and practice is based on *generative* systems, such as regular expressions and context-free grammars, in which a language is defined formally by a set of rules applied recursively to generate strings of the language. A *recognition-based* system, in contrast, defines a language in terms of rules or predicates that decide whether or not a given string is in the language. Simple languages can be expressed easily in either paradigm. For example, $\{s \in a^* \mid s = (aa)^n\}$ is a generative definition of a trivial language over a unary character set, whose strings are "constructed" by concatenating pairs of a's. In contrast, $\{s \in a^* \mid (|s| \ mod \ 2 = 0)\}$ is a recognition-based definition of the same language, in which a string of a's is "accepted" if its length is even.

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. Bridging the gap from generative definitions to practical recognizers is the purpose of our ever-expanding library of parsing algorithms with diverse capabilities and trade-offs [9].

Chomsky's generative system of grammars, from which the ubiqui-

⋮



A Text Pattern-Matching Tool based on Parsing Expression Grammars

Roberto Ierusalimschy[1]

[1] *PUC-Rio, Brazil*

This is a preprint of an article accepted for publication in Software: Practice and Experience; *Copyright 2008 by John Willey and Sons.*

SUMMARY

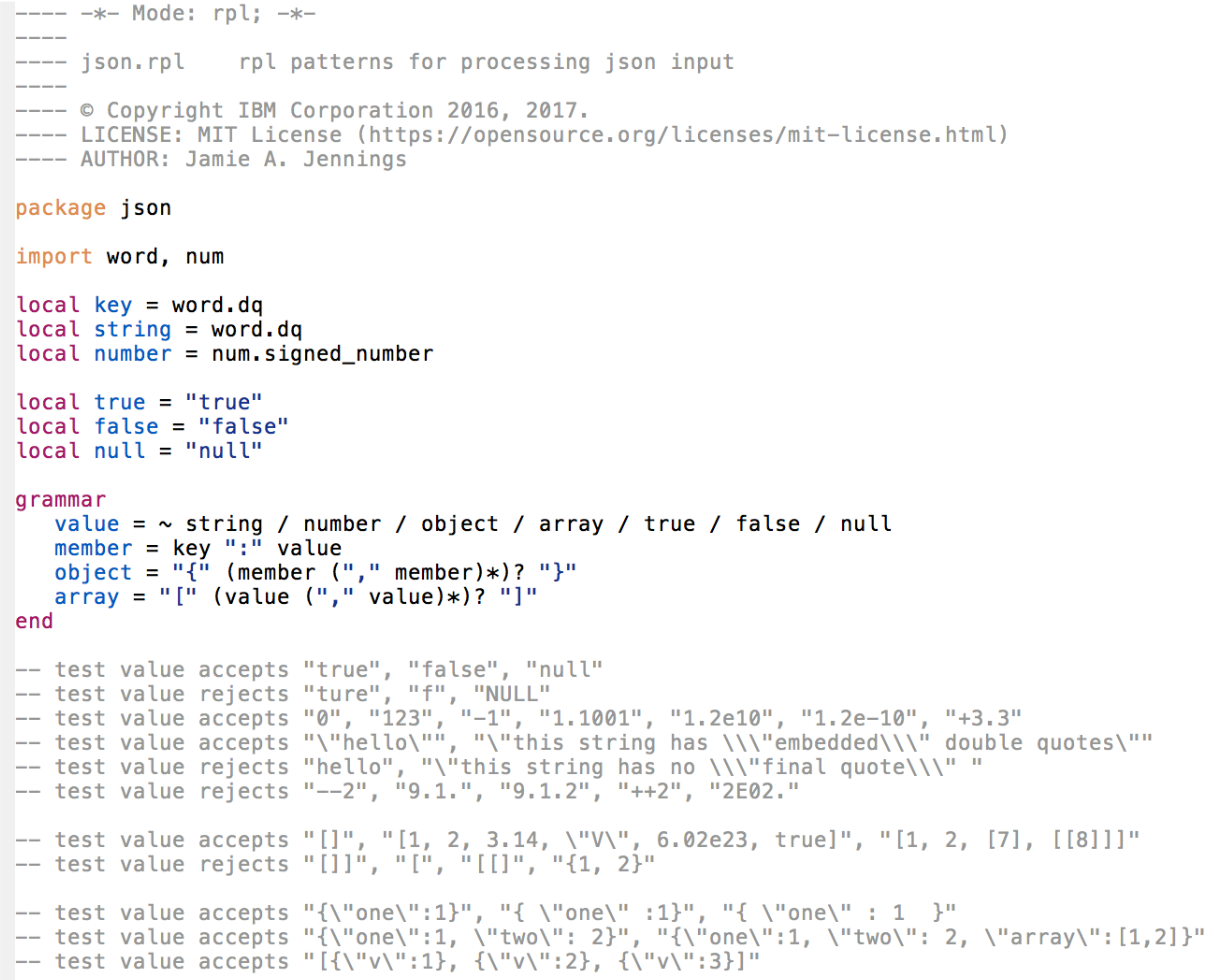
Current text pattern-matching tools are based on regular expressions. However, pure regular expressions have proven too weak a formalism for the task: many interesting patterns either are difficult to describe or cannot be described by regular expressions. Moreover, the inherent non-determinism of regular expressions does not fit the need to capture specific parts of a match.

Motivated by these reasons, most scripting languages nowadays use pattern-matching tools that extend the original regular-expression formalism with a set of ad-hoc features, such as greedy repetitions, lazy repetitions, possessive repetitions, "longest match rule", lookahead, etc. These ad-hoc extensions bring their own set of problems, such as lack of a formal foundation and complex implementations.

In this paper, we propose the use of Parsing Expression Grammars (PEGs) as a basis for pattern matching. Following this proposal, we present LPEG, a pattern-matching tool based on PEGs for the Lua scripting language. LPEG unifies the ease of use of pattern-matching tools with the full expressive power of PEGs. Because of this expressive power, it can avoid the myriad of ad-hoc constructions present in several current pattern-matching tools. We also present a Parsing Machine that allows a small and efficient implementation of PEGs for pattern matching.

KEY WORDS: pattern matching, Parsing Expression Grammars, scripting languages

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 "ture", "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", "++2", "2E02."

-- test value accepts "[]", "[1, 2, 3.14, \"V\", 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

Comments

Modules

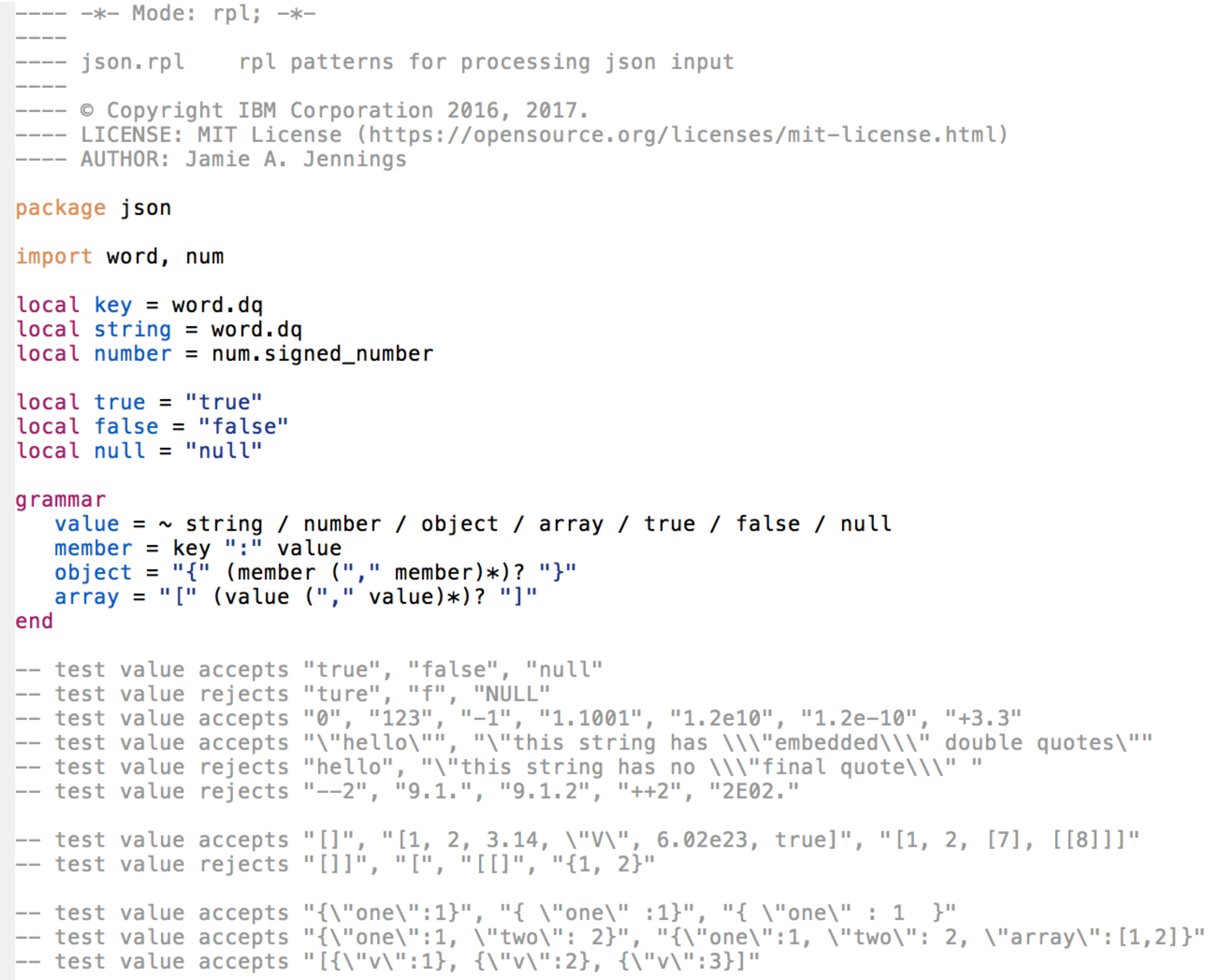
Identifiers

Whitespace

Quoted literals

Macros (not shown)

Unit tests



```
---- -*- 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 "ture", "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", "++2", "2E02."

-- test value accepts "[]", "[1, 2, 3.14, \"V\", 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

Comments

Modules

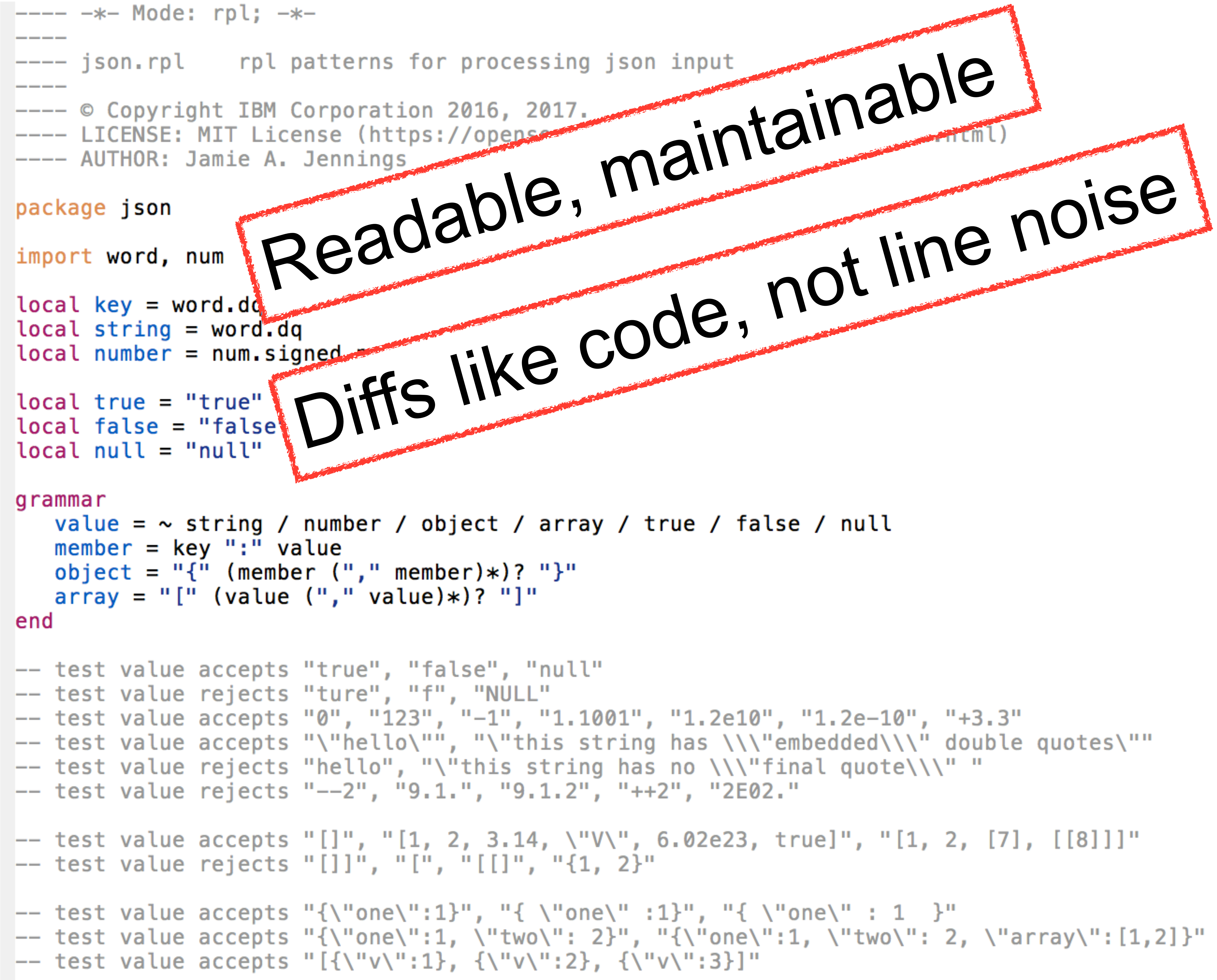
Identifiers

Whitespace

Quoted literals

Macros (not shown)

Unit tests



```
---- -*- Mode: rpl; -*-
----
---- json.rpl    rpl patterns for processing json input
----
---- © Copyright IBM Corporation 2016, 2017.
---- LICENSE: MIT License (https://opens                                    html)
---- AUTHOR: Jamie A. Jennings

package json

import word, num

local key = word.dq
local string = word.dq
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 "ture", "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", "++2", "2E02."

-- test value accepts "[]", "[1, 2, 3.14, \"V\", 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}]"
```

Readable, maintainable

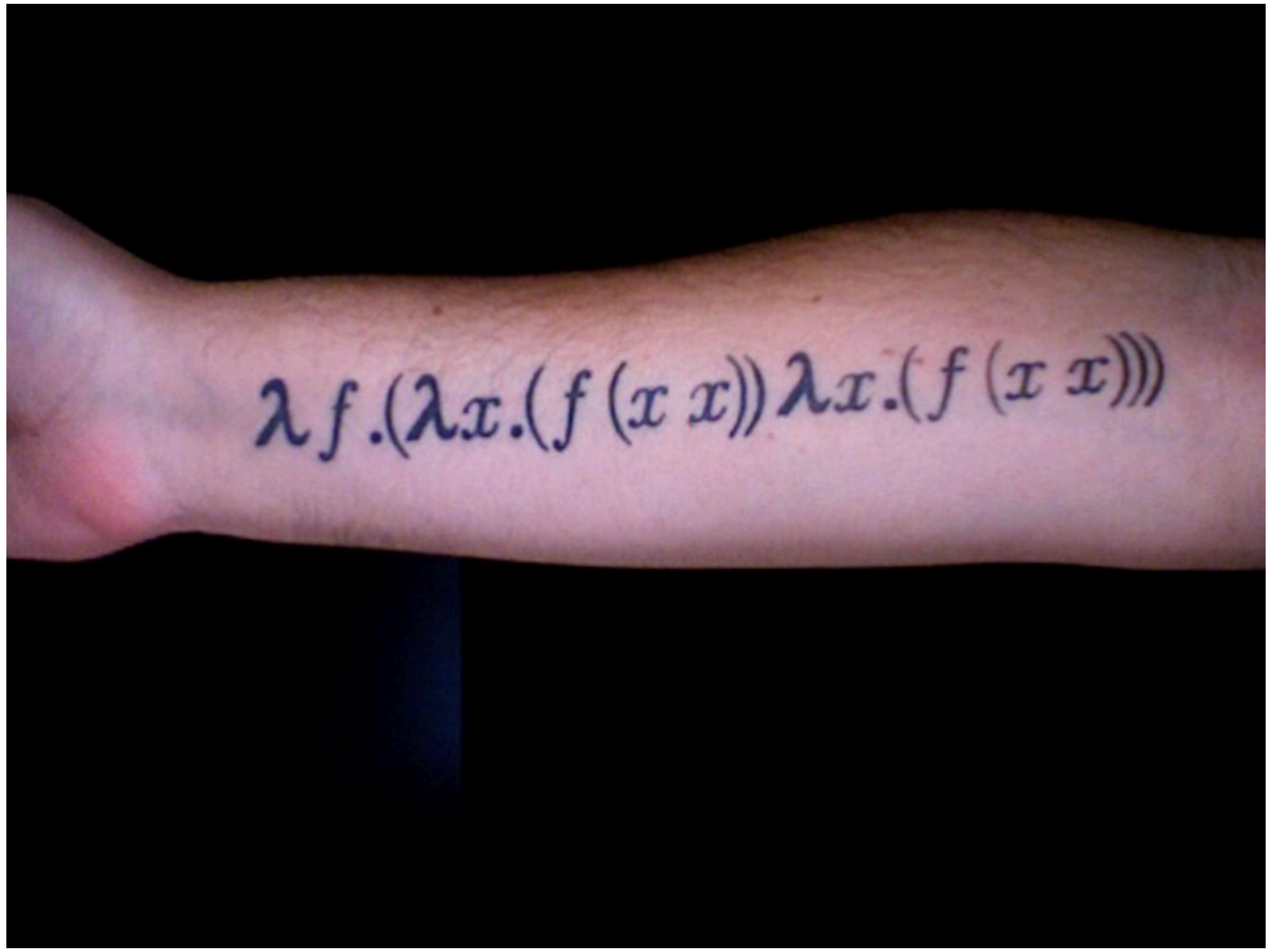
Diffs like code, not line noise

Semantics

- Combinators

Semantics

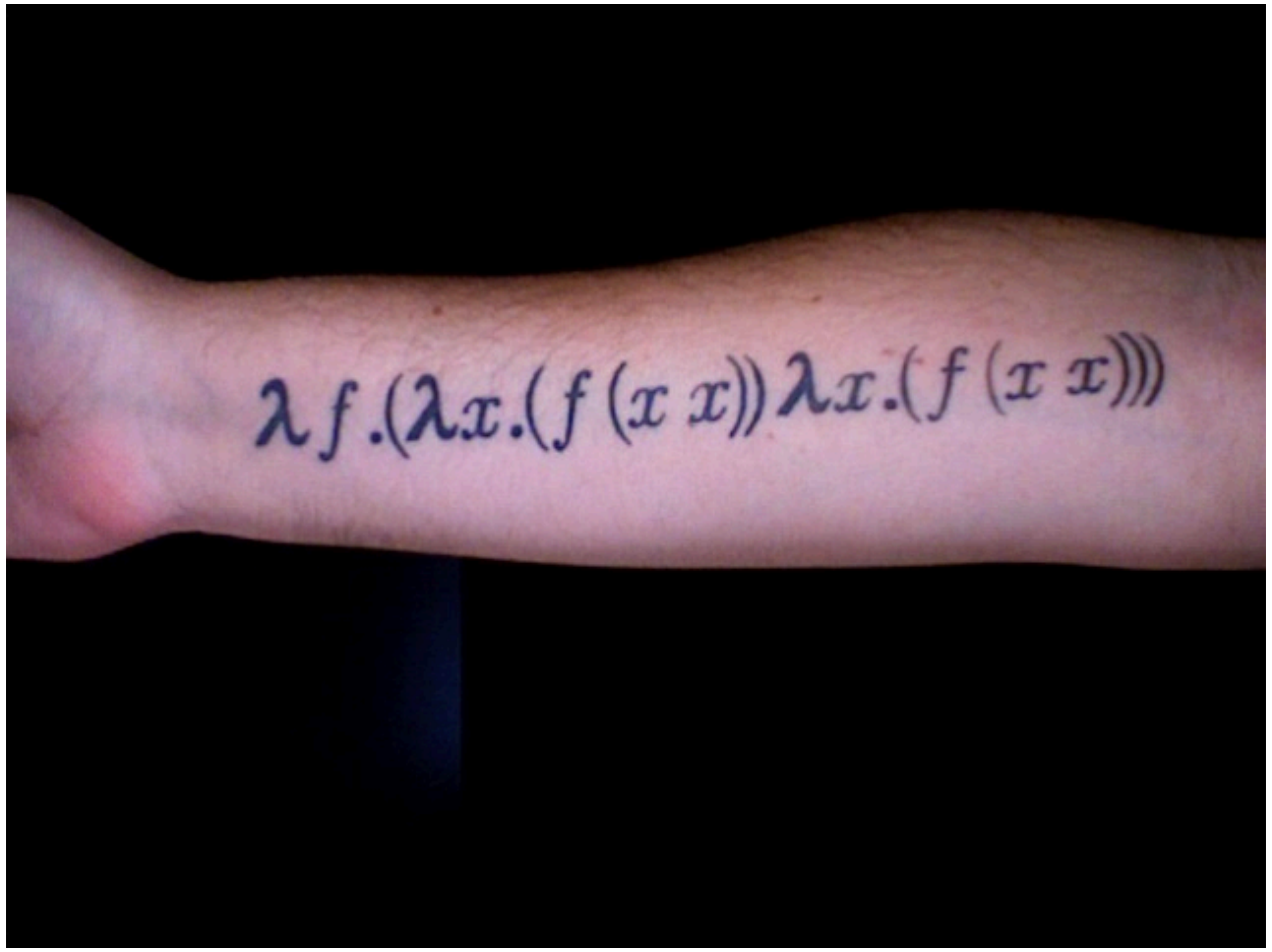
- Combinators



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

Semantics

- Combinators



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

Semantics

Kleene star is possessive, so `.* "x"` *always fails*

- Combinators
- Lisp-like macros

```
{!"x" .}* "x"
```

Semantics

Kleene star is possessive, so `.* "x"` *always fails*

- Combinators
- Lisp-like macros

`find:"x"` $\overset{\text{def}}{=}$ `{!"x" .}* "x"`

Can write this instead

Semantics

Kleene star is possessive, so `.* "x"` *always fails*

- Combinators
- Lisp-like macros

`find:"x"` $\overset{\text{def}}{=}$ `{!"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

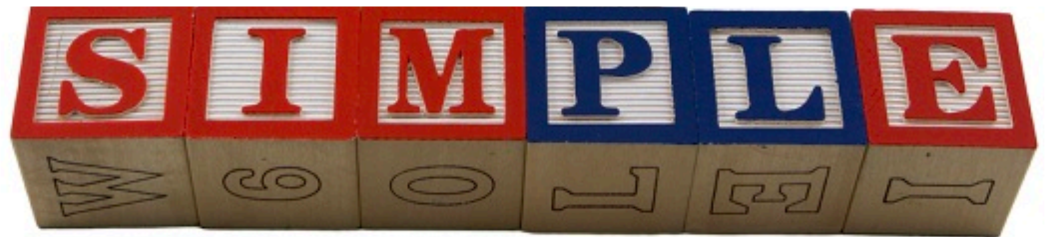
- Combinators

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

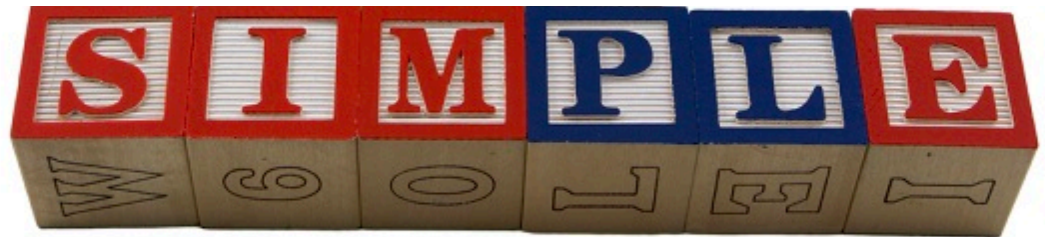
Semantics

- Combinators

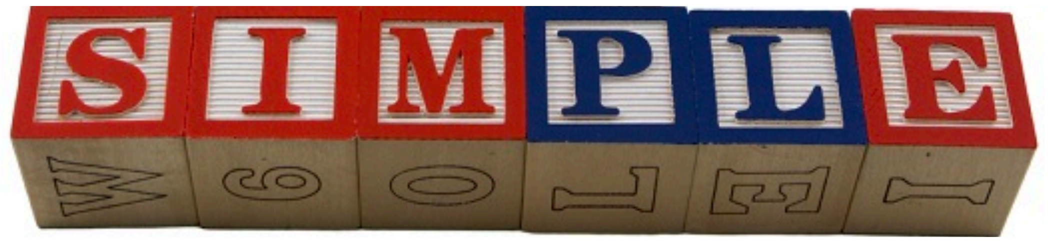
- Lisp-like macros
- Import mechanism like Go



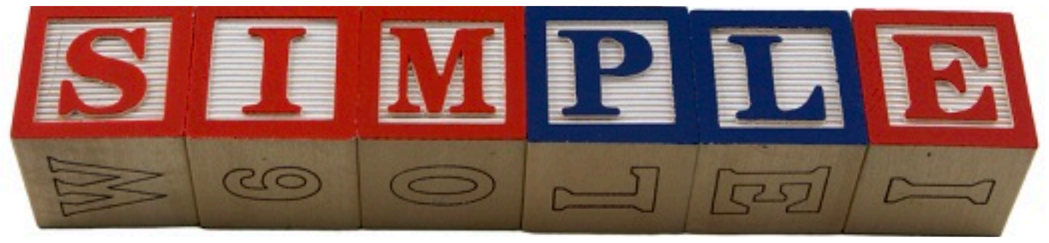
- Prelude like Haskell



- Environments like any Lisp-1



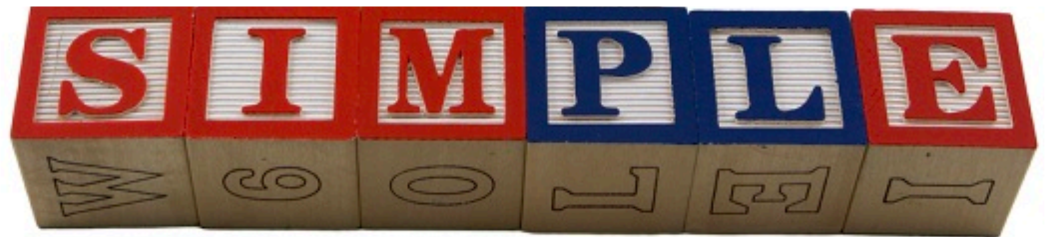
- Binding rules like Scheme



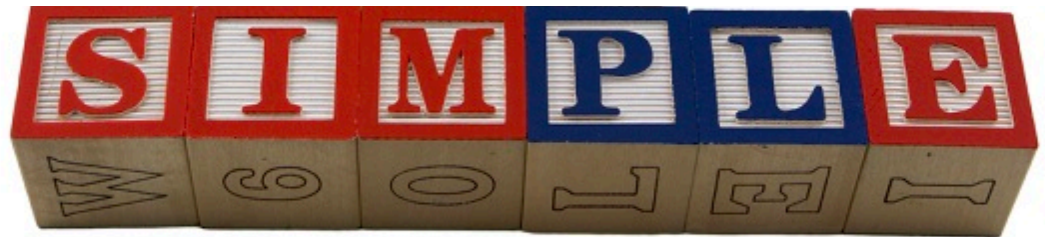
Semantics

- Combinators

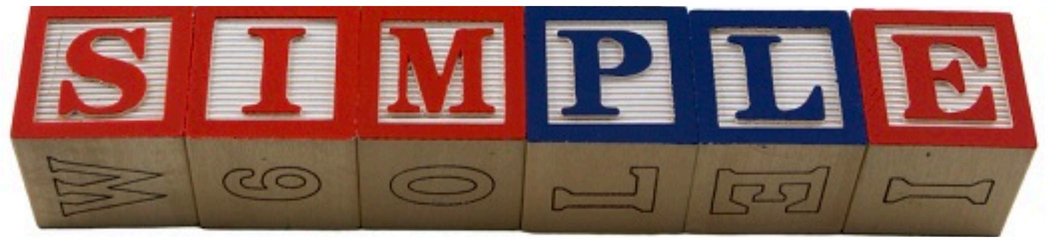
- Lisp-like macros 🙄
- Import mechanism like Go



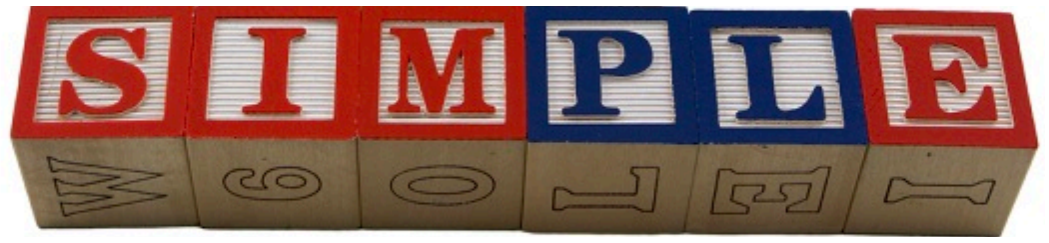
- Prelude like Haskell



- Environments like any Lisp-1



- Binding rules like Scheme



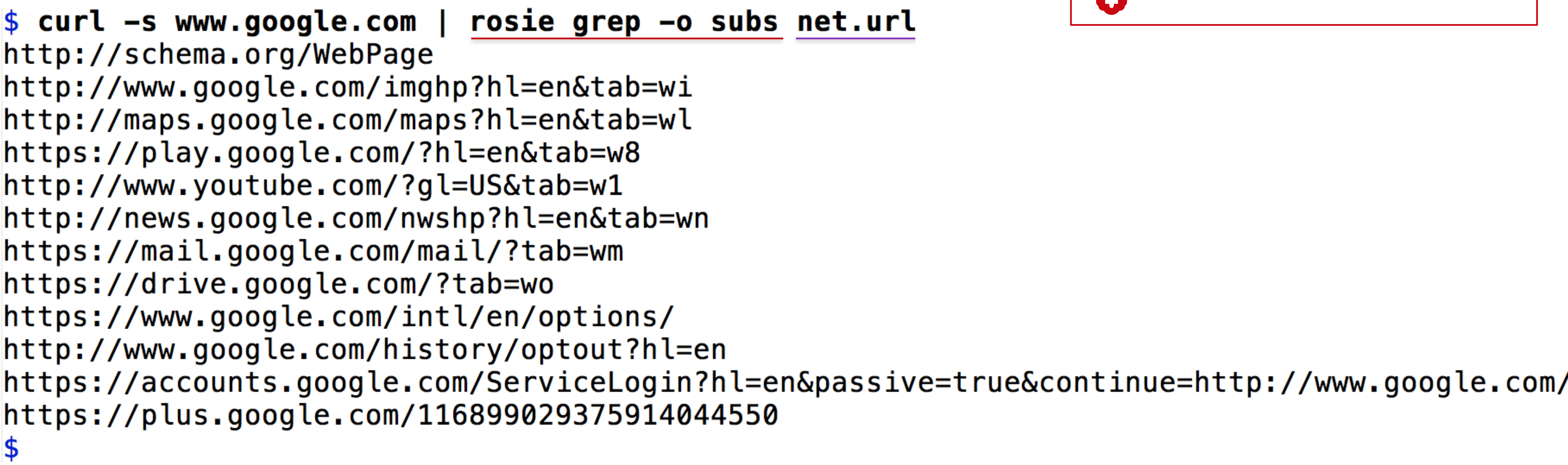
Implementation

“I want to believe”

Fox Mulder, FBI

Can your 'grep' do this?

NAMED PATTERNS



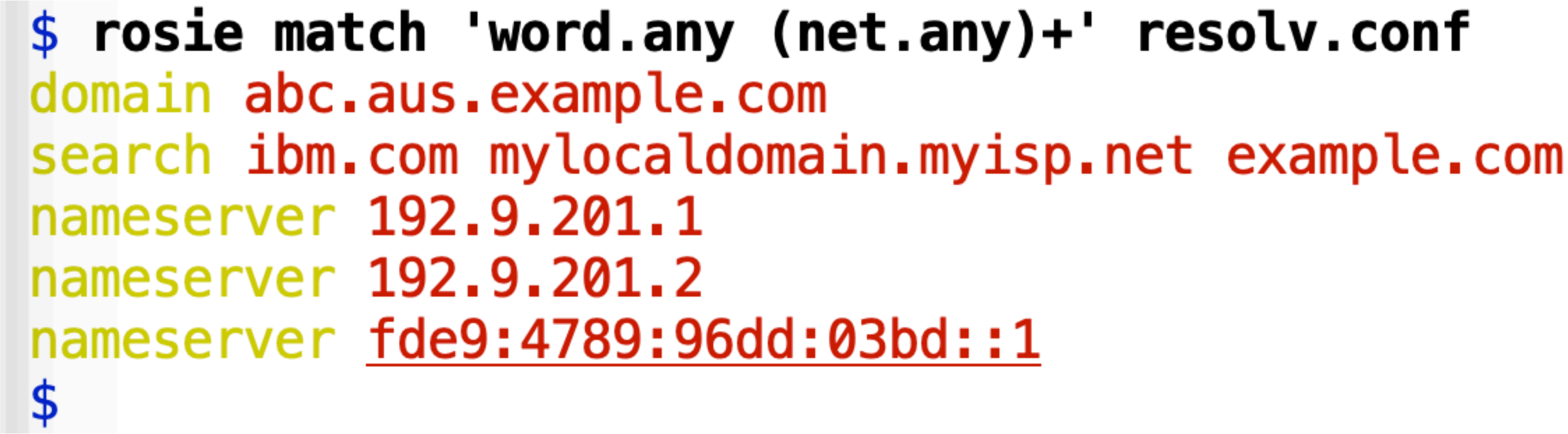
```
$ curl -s www.google.com | rosie grep -o subs net.url
http://schema.org/WebPage
http://www.google.com/imghp?hl=en&tab=wi
http://maps.google.com/maps?hl=en&tab=wl
https://play.google.com/?hl=en&tab=w8
http://www.youtube.com/?gl=US&tab=w1
http://news.google.com/nwshp?hl=en&tab=wn
https://mail.google.com/mail/?tab=wm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

-o Output format
subs ==> sub-matches

net.url
==> package net, pattern url

Can your 'grep' do this?

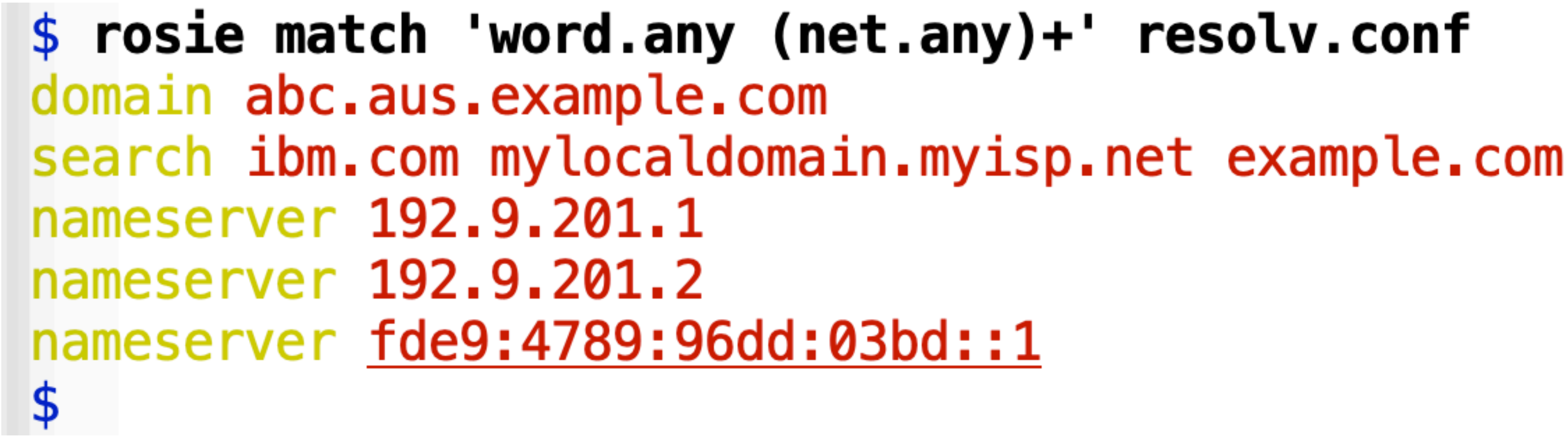
Customizable Output highlighting



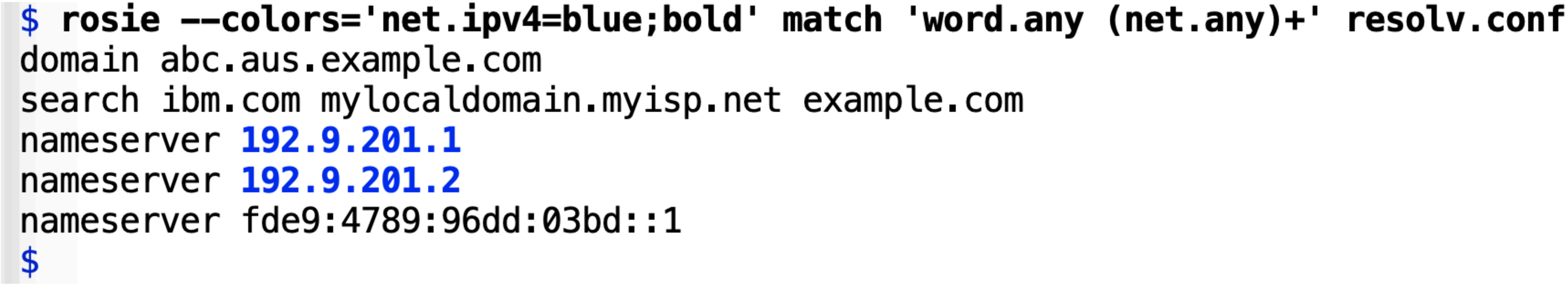
```
$ rosie match 'word.any (net.any)+' resolv.conf
domain abc.aus.example.com
search ibm.com mylocaldomain.myisp.net example.com
nameserver 192.9.201.1
nameserver 192.9.201.2
nameserver fde9:4789:96dd:03bd::1
$
```

Can your ‘grep’ do this?

Customizable Output highlighting



```
$ 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
$
```

Can your ‘grep’ do this?

Customizable Output Highlighting

```
$ sed -n 46,49p /var/log/system.log
Jul 30 10:18:42 Jamies-Compabler com.apple.xpc.launchd[1] (com.apple.CoreSimulator.CoreSimulatorService
[669]): Service exited due to signal: Killed: 9 sent by com.apple.CoreSimulator.CoreSimu[669]
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 i
mplemented in both /System/Library/PrivateFrameworks/CloudDocs.framework/Versions/A/CloudDocs (0x7fff8b
848c88) and /System/Library/PrivateFrameworks/CloudDocsDaemon.framework/XPCServices/ContainerMetadataEx
tractor.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
```

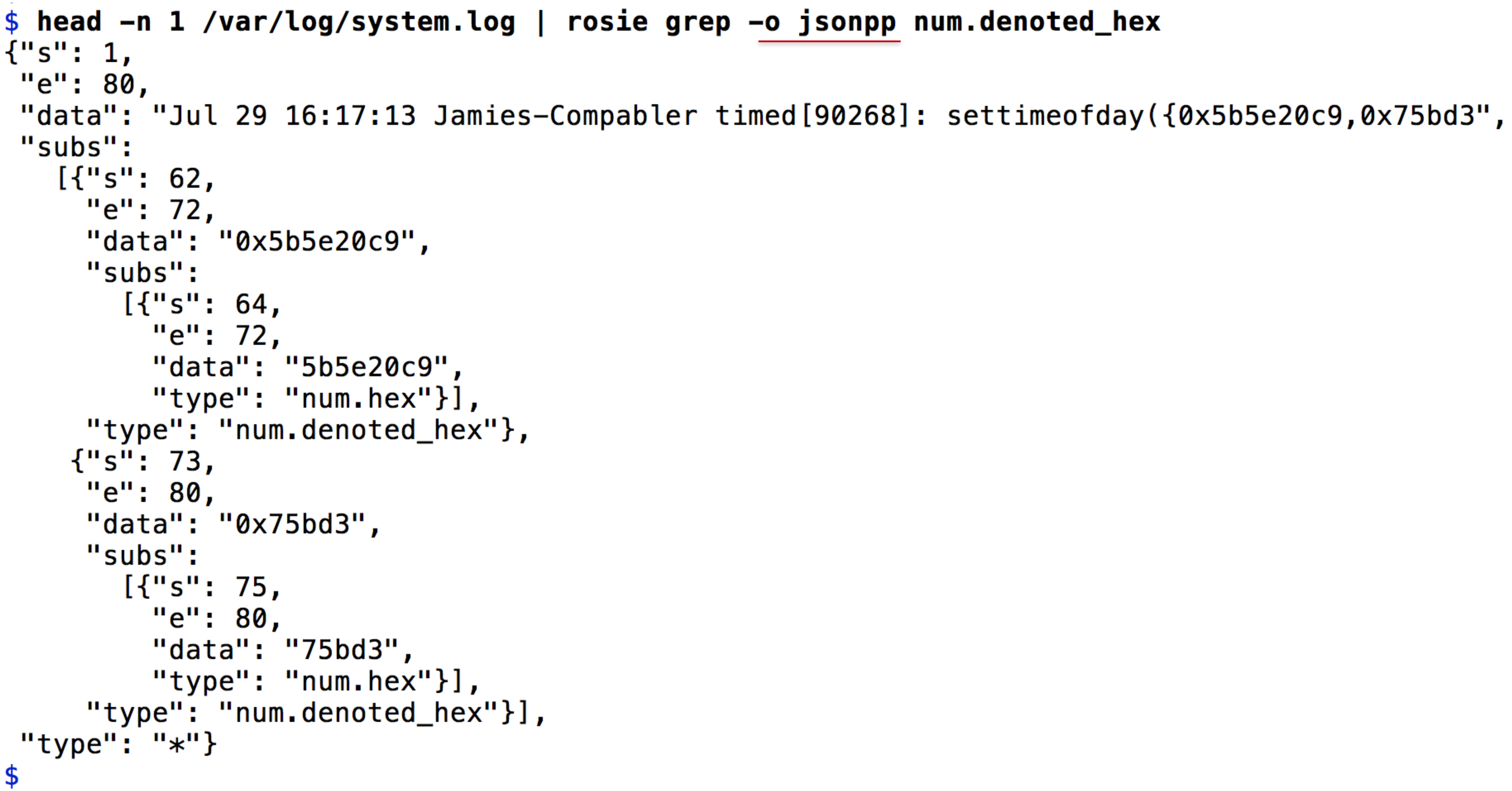
Can your ‘grep’ do this?

Customizable Output Highlighting

```
$ sed -n 46,49p /var/log/system.log
Jul 30 10:18:42 Jamies-Compabler com.apple.xpc.launchd[1] (com.apple.CoreSimulator.CoreSimulatorService
[669]): Service exited due to signal: Killed: 9 sent by com.apple.CoreSimulator.CoreSimu[669]
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 i
mplemented in both /System/Library/PrivateFrameworks/CloudDocs.framework/Versions/A/CloudDocs (0x7fff8b
848c88) and /System/Library/PrivateFrameworks/CloudDocsDaemon.framework/XPCServices/ContainerMetadataEx
tractor.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
$
$ sed -n 46,49p  /var/log/system.log | rosie match all.things
Jul 30 10:18:42 Jamies-Compabler com.apple.xpc.launchd[1] (com.apple.CoreSimulator.CoreSimulatorService
[669]): Service exited due to signal: Killed: 9 sent by com.apple.CoreSimulator.CoreSimu[669]
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 i
mplemented in both /System/Library/PrivateFrameworks/CloudDocs.framework/Versions/A/CloudDocs (0x7fff8b
848c88) and /System/Library/PrivateFrameworks/CloudDocsDaemon.framework/XPCServices/ContainerMetadataEx
tractor.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?

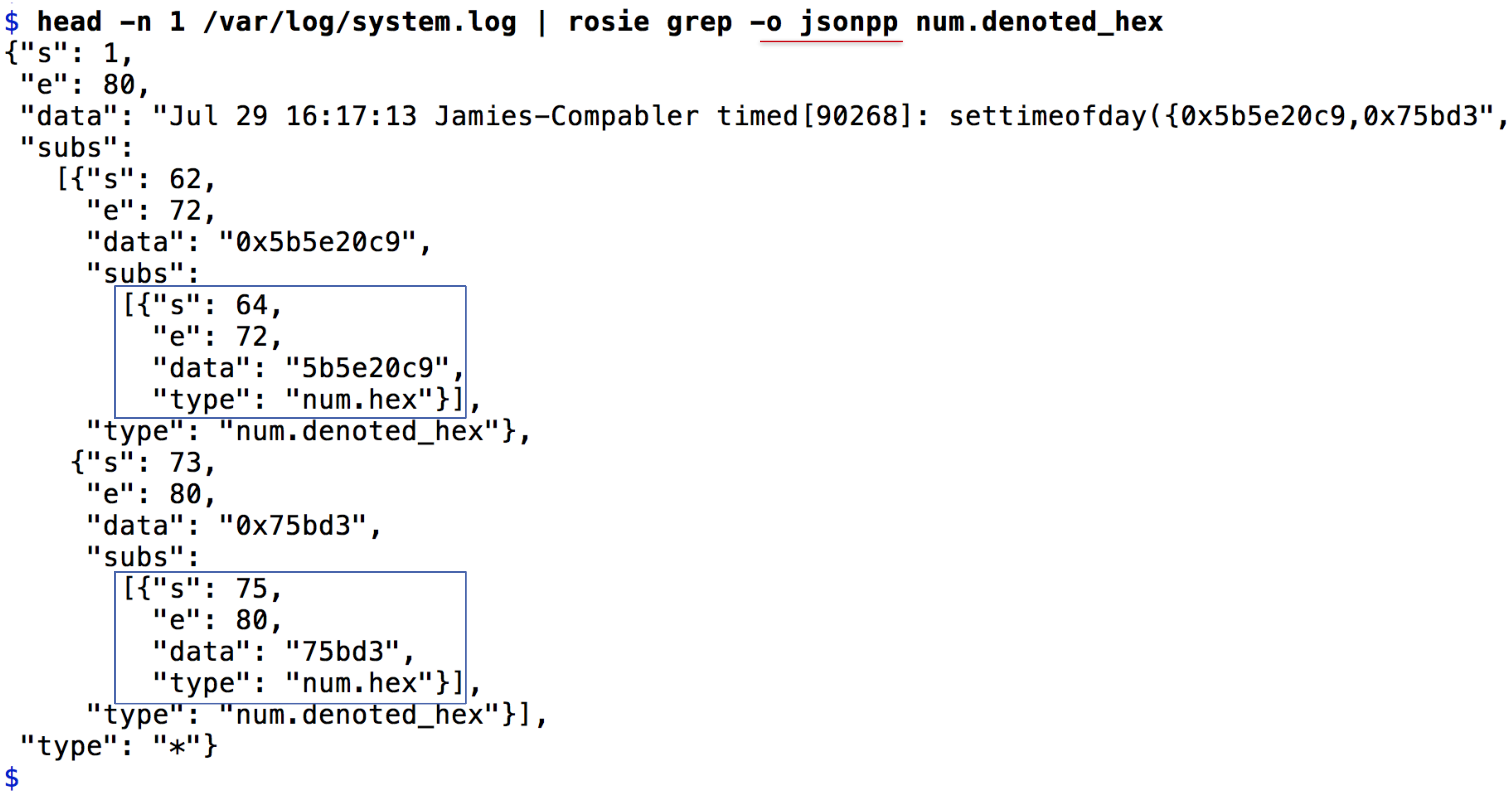
Structured output option



```
$ 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?

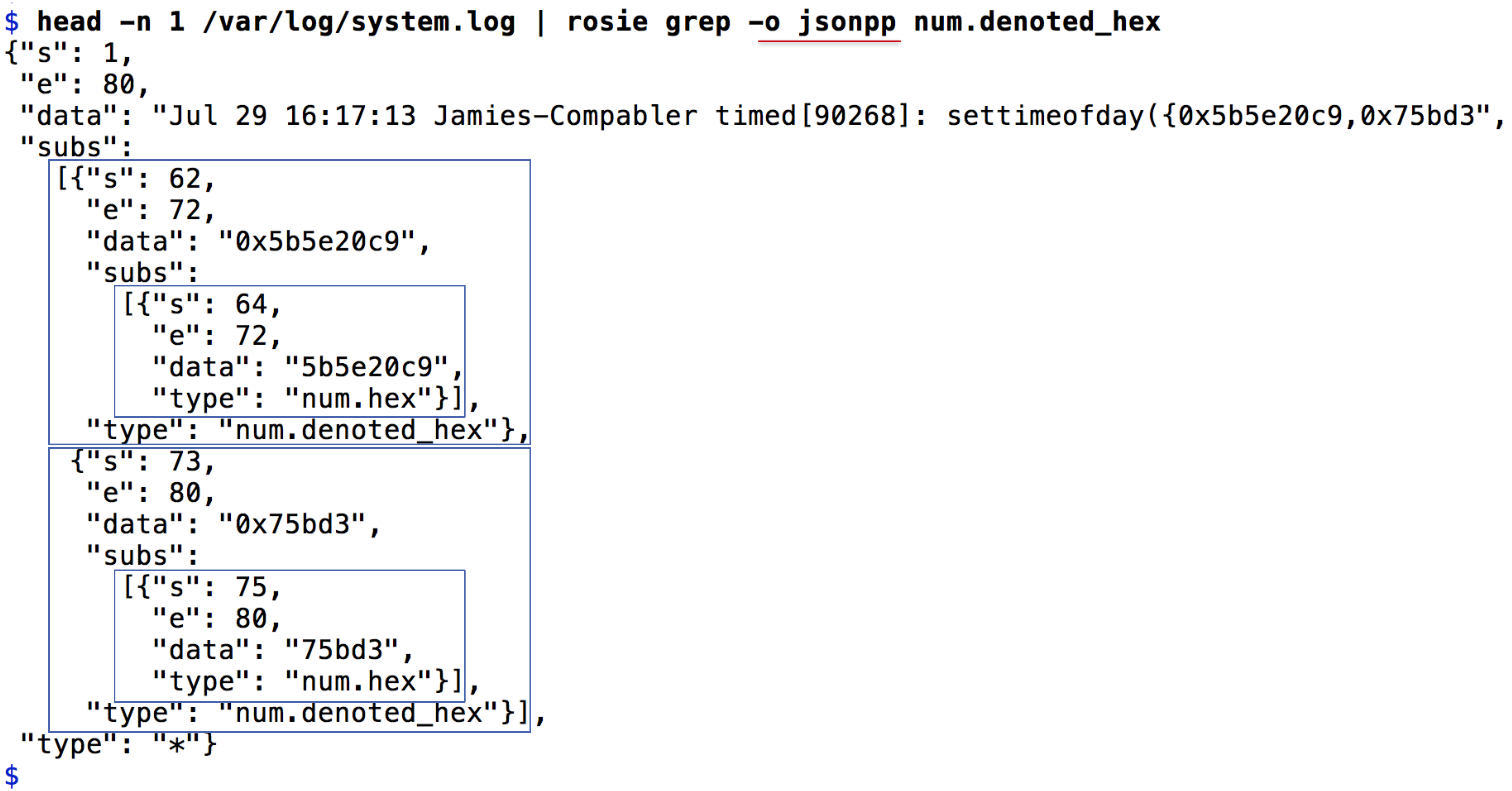
STRUCTURED OUTPUT OPTION



```
$ 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?

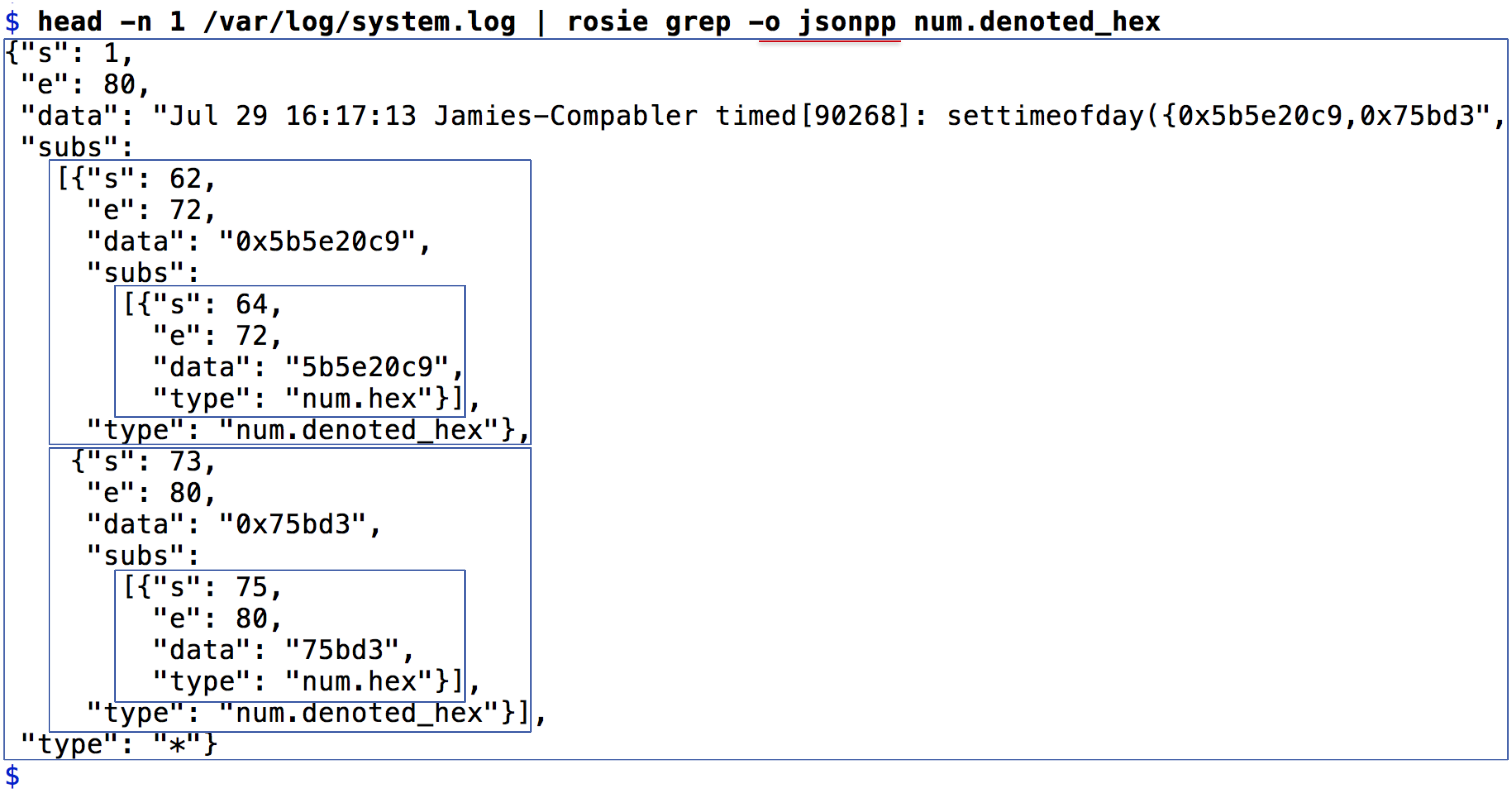
STRUCTURED OUTPUT OPTION



```
$ 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": "*"}
$
```

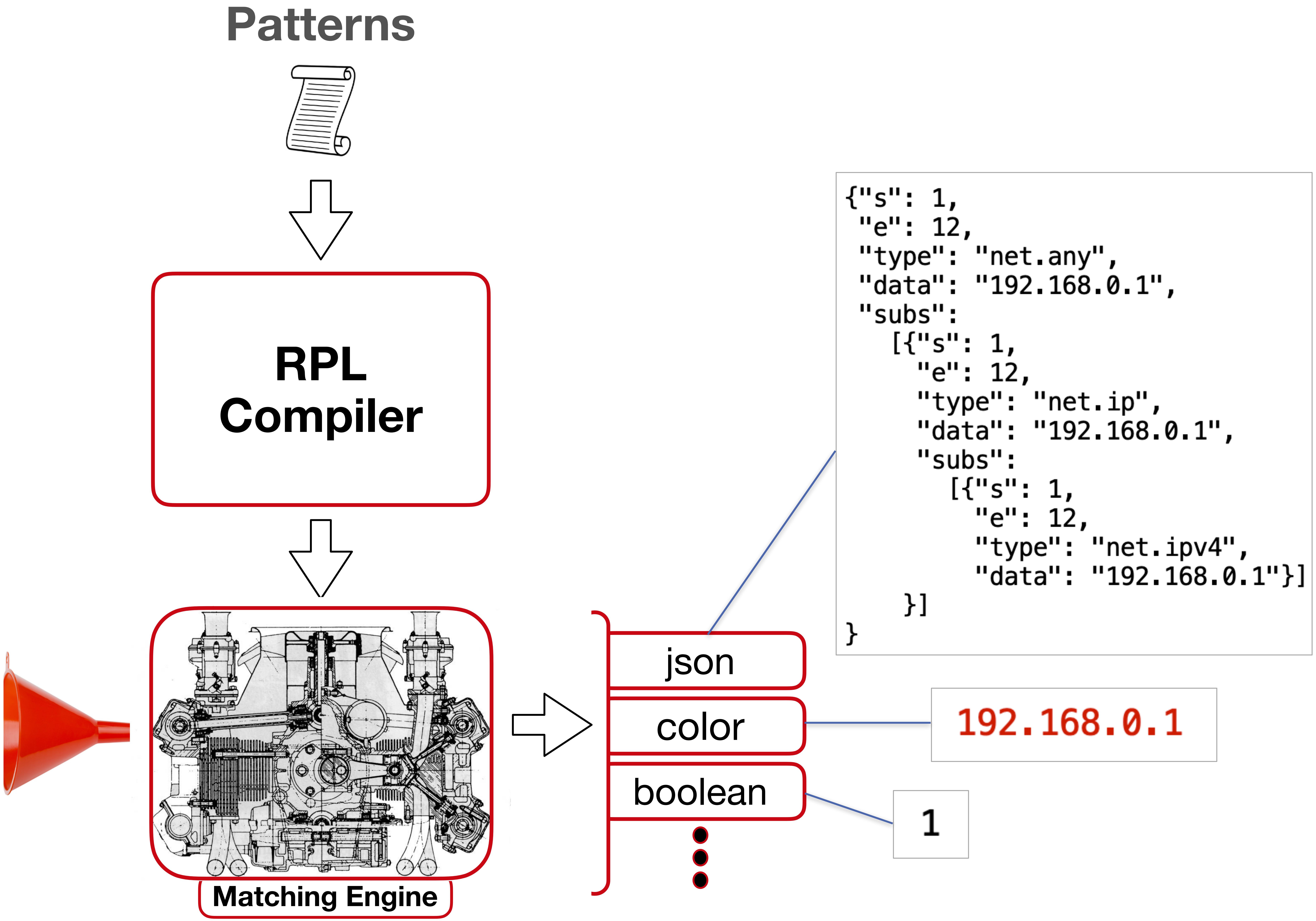
Can your ‘grep’ do this?

STRUCTURED OUTPUT OPTION



```
$ 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": "*"}
$
```

Rosie Architecture



```
{"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
```

```
1
```

Rosie Architecture

Patterns
RPL Compiler
Matching Engine
json
color
boolean
192.168.0.1
1

```
{"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"}]
    }]
}
```

Rosie Architecture

Patterns

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

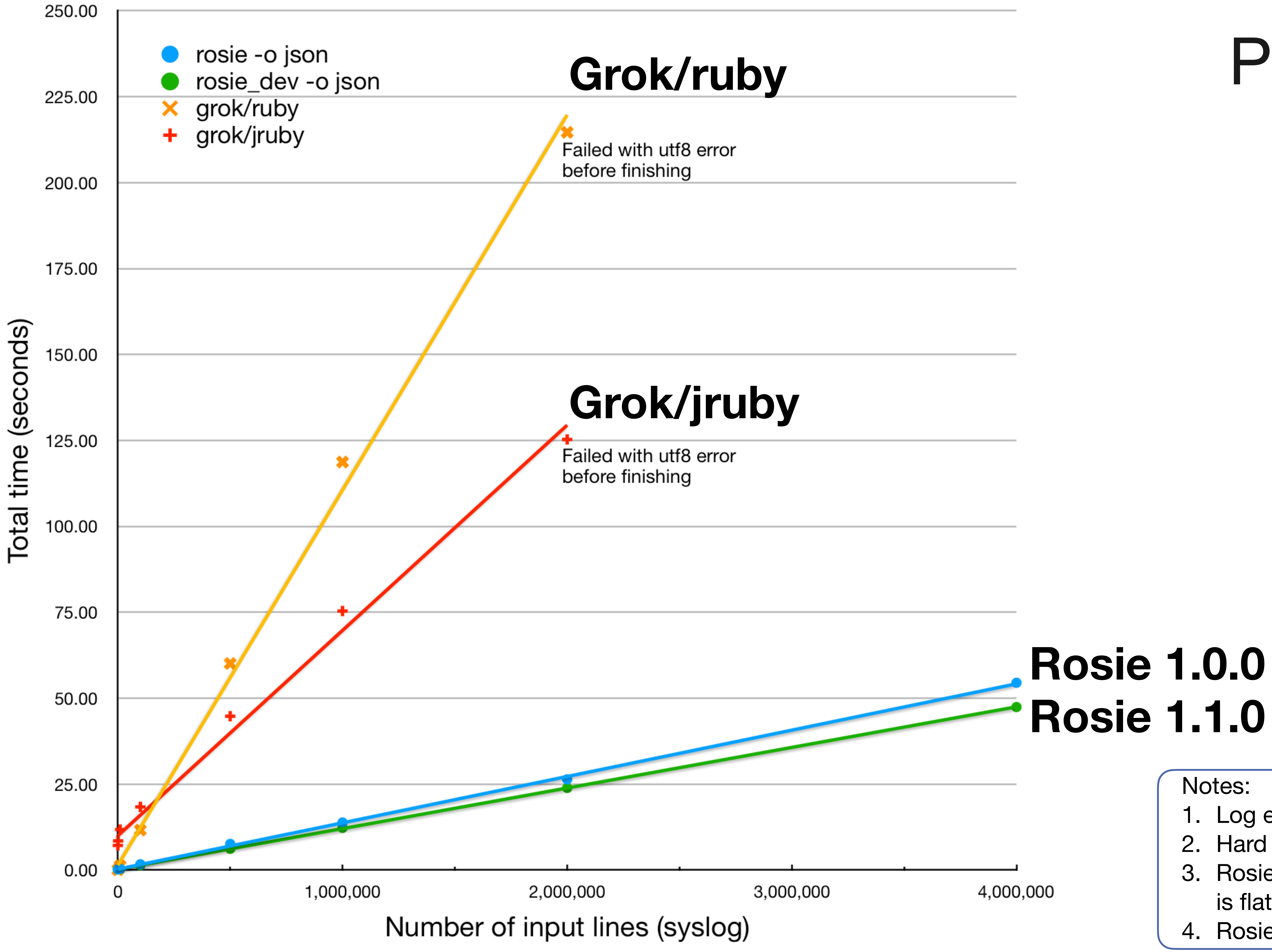
RPL Compiler
Matching Engine
json
color
boolean

```
{"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"}]
   }]
}
```

```
192.168.0.1
```

```
1
```

Performance



Grok/ruby

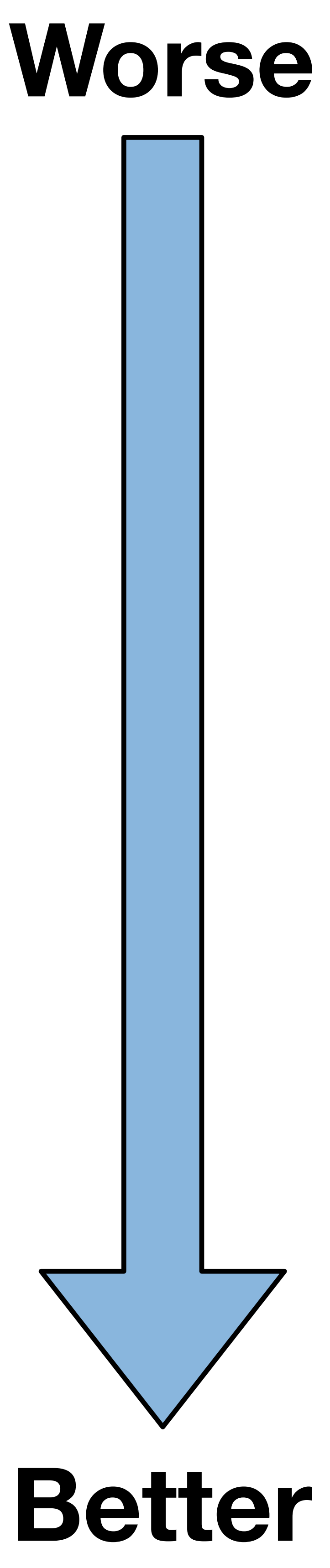
Failed with utf8 error before finishing

Grok/jruby

Failed with utf8 error before finishing

Rosie 1.0.0

Rosie 1.1.0



Worse

Better

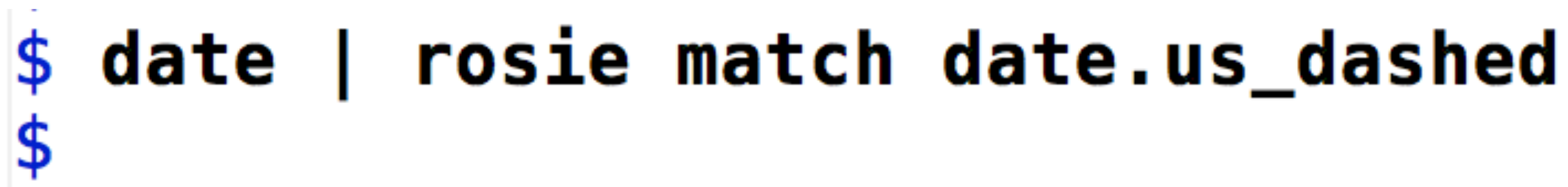
Notes:

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

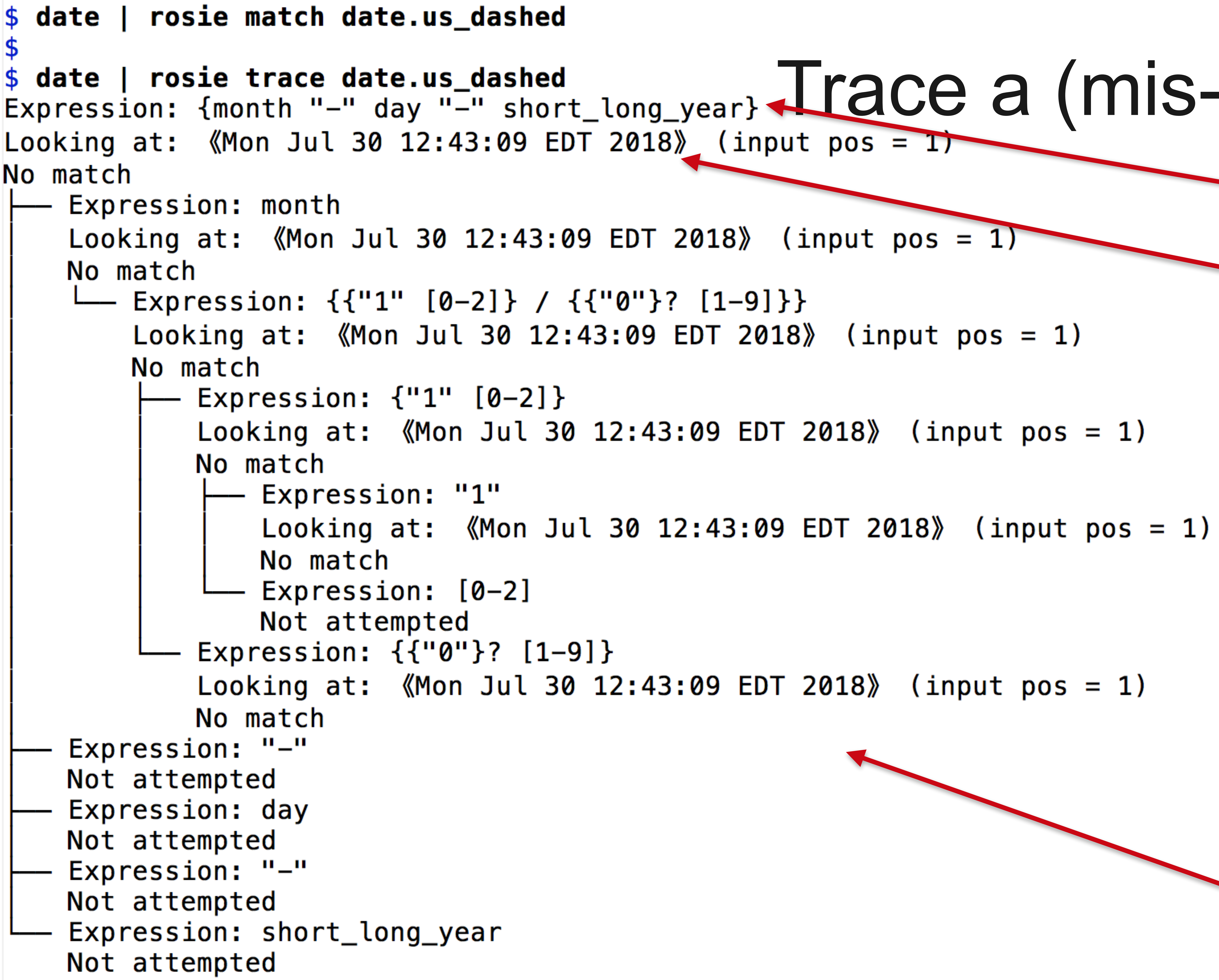
> “To err is human, but to really foul things up you need a computer.”
>
> Paul R. Ehrlich

Trace a (mis-)match



```
$ date | rosie match date.us_dashed
$
```

Trace a (mis-)match



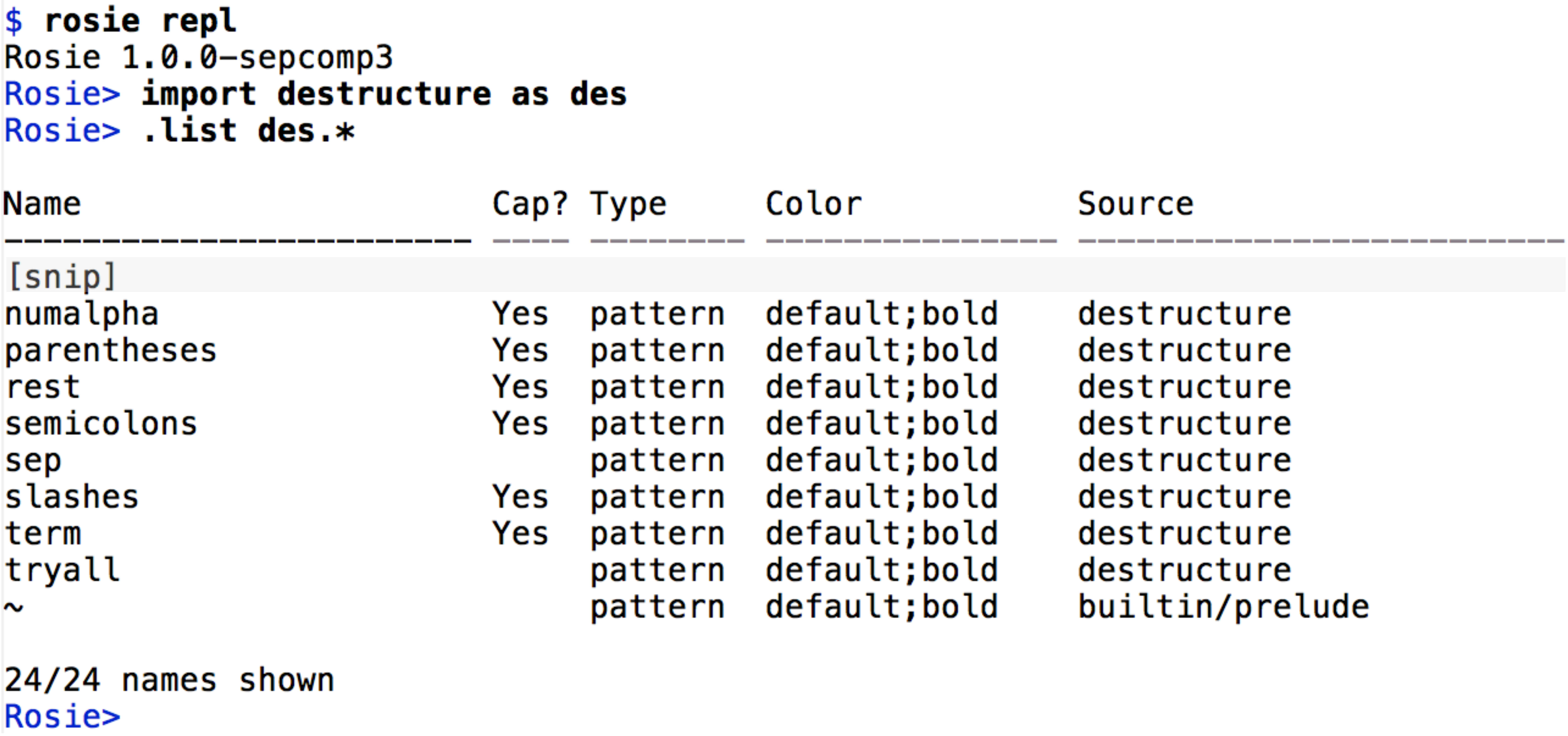
```
$ date | rosie match date.us_dashed
$
$ date | rosie trace date.us_dashed
Expression: {month "-" day "-" short_long_year}
Looking at: 《Mon Jul 30 12:43:09 EDT 2018》 (input pos = 1)
No match
├── Expression: month
│   Looking at: 《Mon Jul 30 12:43:09 EDT 2018》 (input pos = 1)
│   No match
│   └── Expression: {{"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
```

Pattern definition

Input text

Matching steps

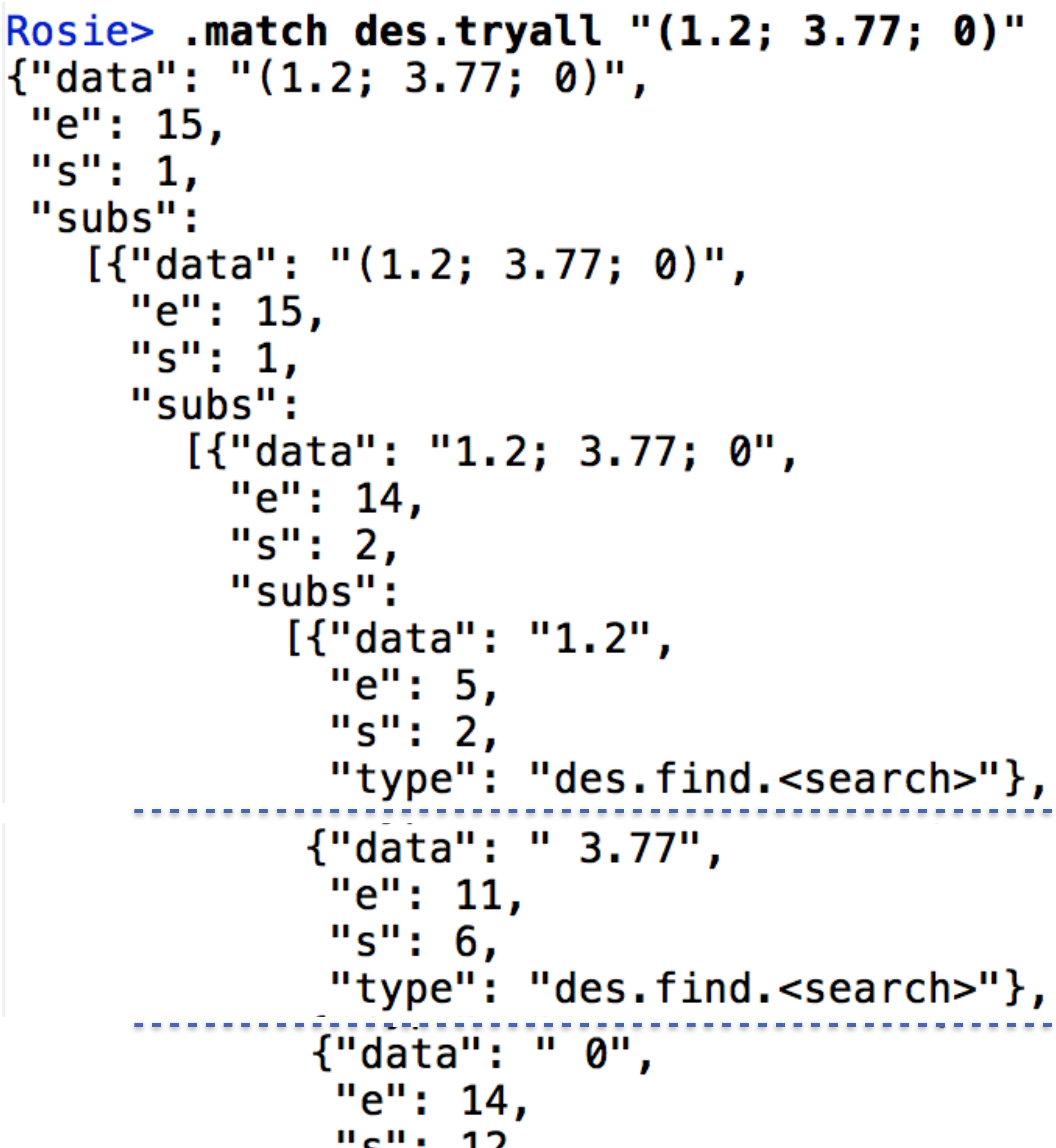
Read-eval-print loop



```
$ rosie repl
Rosie 1.0.0-sepcomp3
Rosie> import destructure as des
Rosie> .list des.*

Name                     Cap? Type     Color           Source
------------------------ ---- -------- --------------- -------------------------
[snip]
numalpha                 Yes  pattern  default;bold    destructure
parentheses              Yes  pattern  default;bold    destructure
rest                     Yes  pattern  default;bold    destructure
semicolons               Yes  pattern  default;bold    destructure
sep                           pattern  default;bold    destructure
slashes                  Yes  pattern  default;bold    destructure
term                     Yes  pattern  default;bold    destructure
tryall                        pattern  default;bold    destructure
~                             pattern  default;bold    builtin/prelude

24/24 names shown
Rosie>
```



```
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>"},
```

snip

```
            {"data": " 3.77",
             "e": 11,
             "s": 6,
             "type": "des.find.<search>"},
```

snip

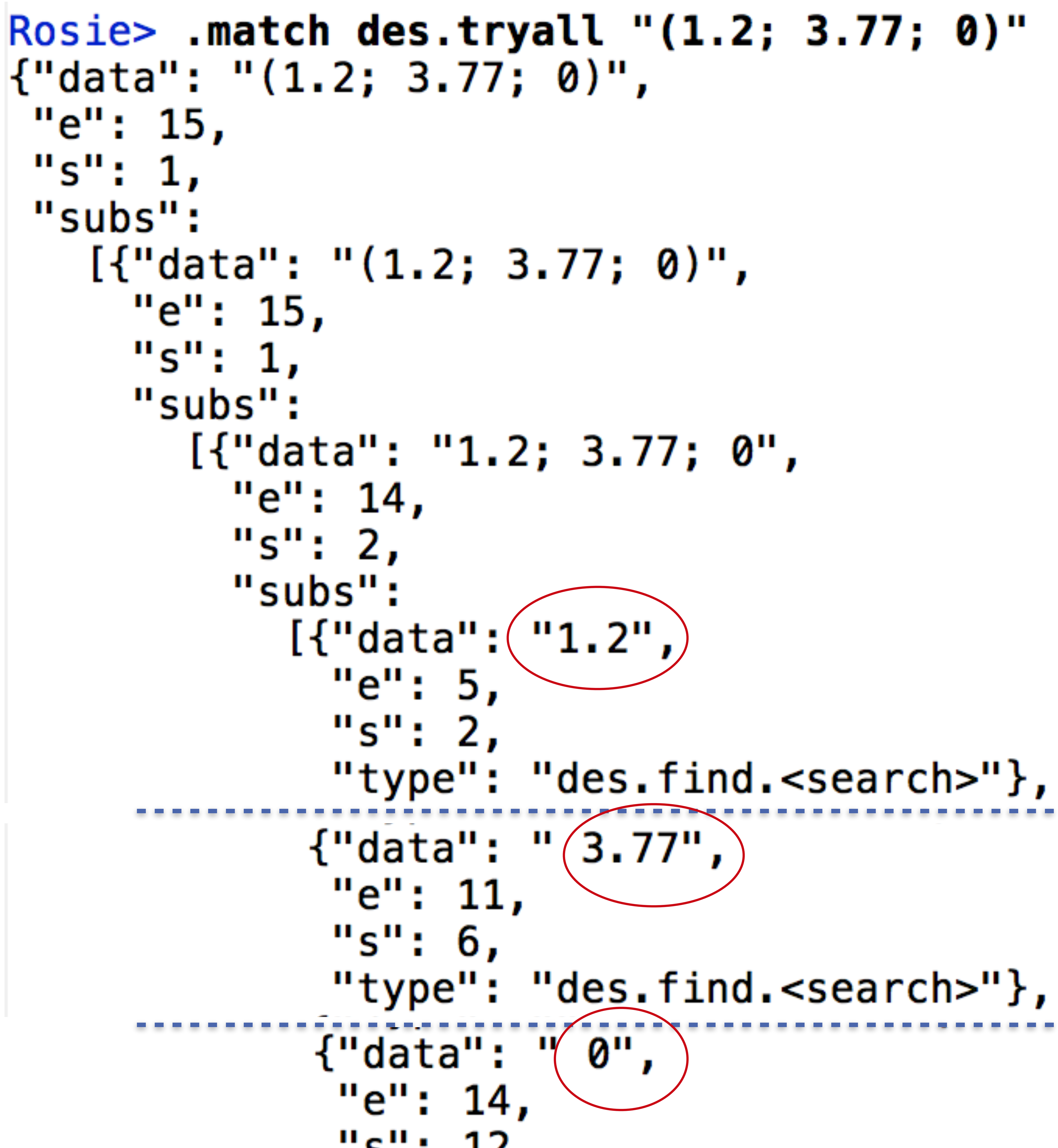
```
            {"data": " 0",
             "e": 14,
             "s": 12
```

Read-eval-print loop

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

Read-eval-print loop

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



```
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>"},
```

snip

```
            {"data": " 3.77",
             "e": 11,
             "s": 6,
             "type": "des.find.<search>"},
```

snip

```
            {"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:

THE C PROGRAMMING LANGUAGE

python™

Haskell

Go

Once and future:

node JS®

Ruby PROGRAMMING Language

Java™ ORACLE®

Lua

Clojure

Thank you!

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

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)