Replacing 50 Year Old Regex Technology with *Rosie Pattern Language*

Jamie A. Jennings, Ph.D.
Department of Computer Science
NC State University
29 June 2019

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

mnordhoff / gist:2213179

Last active 2 months ago • Report gist

Star 8 Fork 1

<> Code   Revisions 11   Stars 8   Forks 1

Embed `<script src="https://gi:`   Download ZIP

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).

gistfile1.py   Raw

```
# 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).

gistfile1.py

Raw

```
# 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
```

Raison d'être

To Do List

1. Mine data from tools
2. Make predictions that help developers

Raison d'être

To Do List

1. Mine data from tools
2. Make predictions that help developers

Raison d'être

To Do List

1. Mine data from tools
2. Make predictions that help developers

➡ My team had to write lots of regex

Raison d'être

IBM

To Do List

1. Mine data from tools
2. Make predictions that help developers

➡ My team had to write lots of regex

➡ We found that regex technology does not scale

1. # of people, over time
2. # of patterns
3. data volume and velocity

Raison d'être

To Do List

1. Mine data from tools
2. Make predictions that help developers

- ➡ My team had to write lots of regex
- ➡ We found that regex technology does not scale
  1. # of people, over time
  2. # of patterns
  3. data volume and velocity
- ➡ So I designed Rosie Pattern Language

Raison d'être

IBM

To Do List

1. Mine data from tools
2. Make predictions that help developers

JIRA

JFrog Artifactory

➡ My team had to write lots of regex

➡ We found that regex technology does not scale

1. # of people, over time
2. # of patterns
3. data volume and velocity

➡ So I designed Rosie Pattern Language

GitLab

rosie-lang.org

Raison d'être

IBM

To Do List

1. Mine data from tools
2. Make predictions that help developers

JIRA

JFrog Artifactory

➡ My team had to write lots of regex

➡ We found that regex technology does not scale

1. # of people, over time
2. # of patterns
3. data volume and velocity

➡ So I designed Rosie Pattern Language

GitLab

rosie-lang.org

NC STATE UNIVERSITY

Department of Computer Science

Current approach: regex

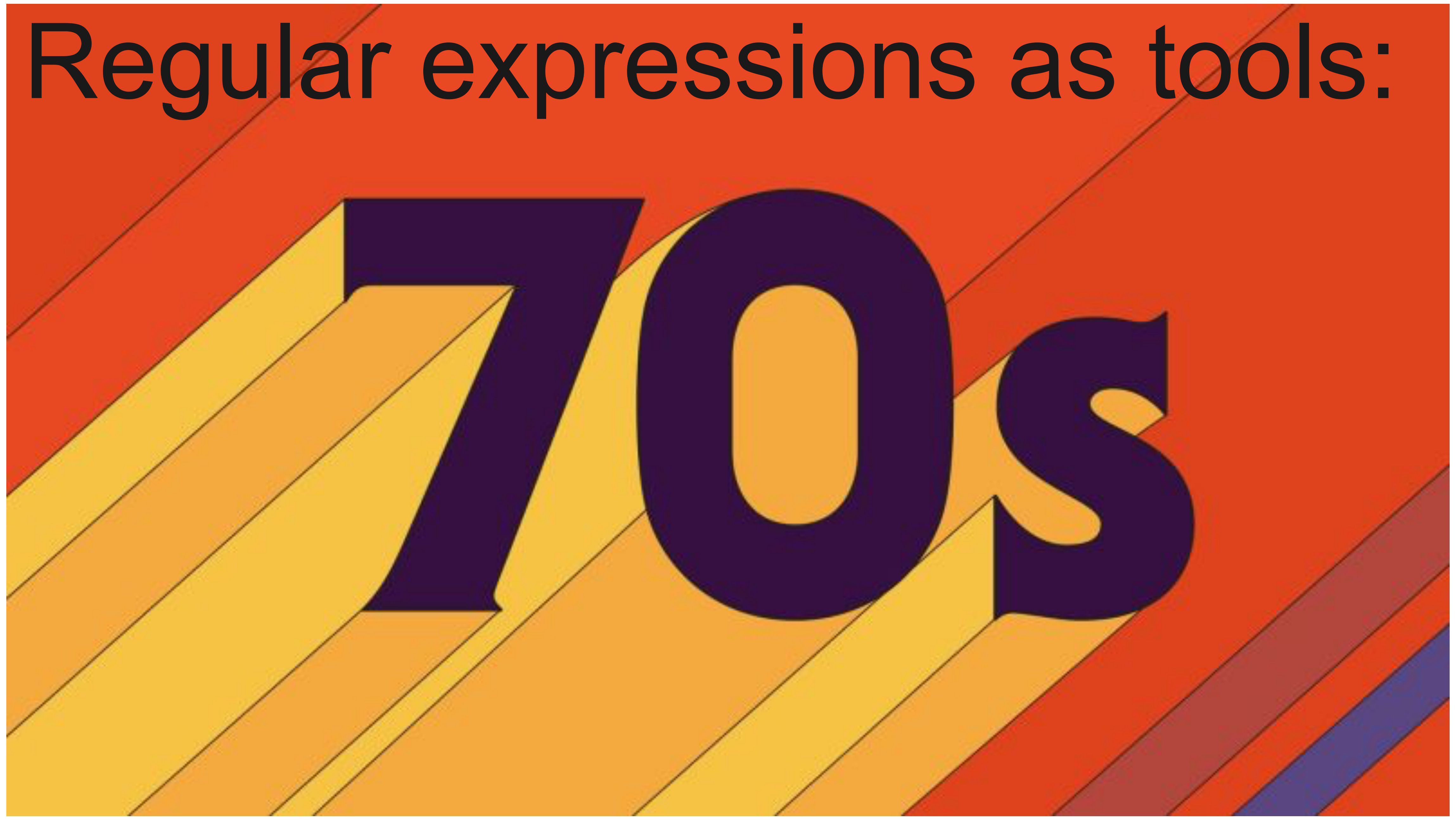
“If the only tool you have is a hammer…”

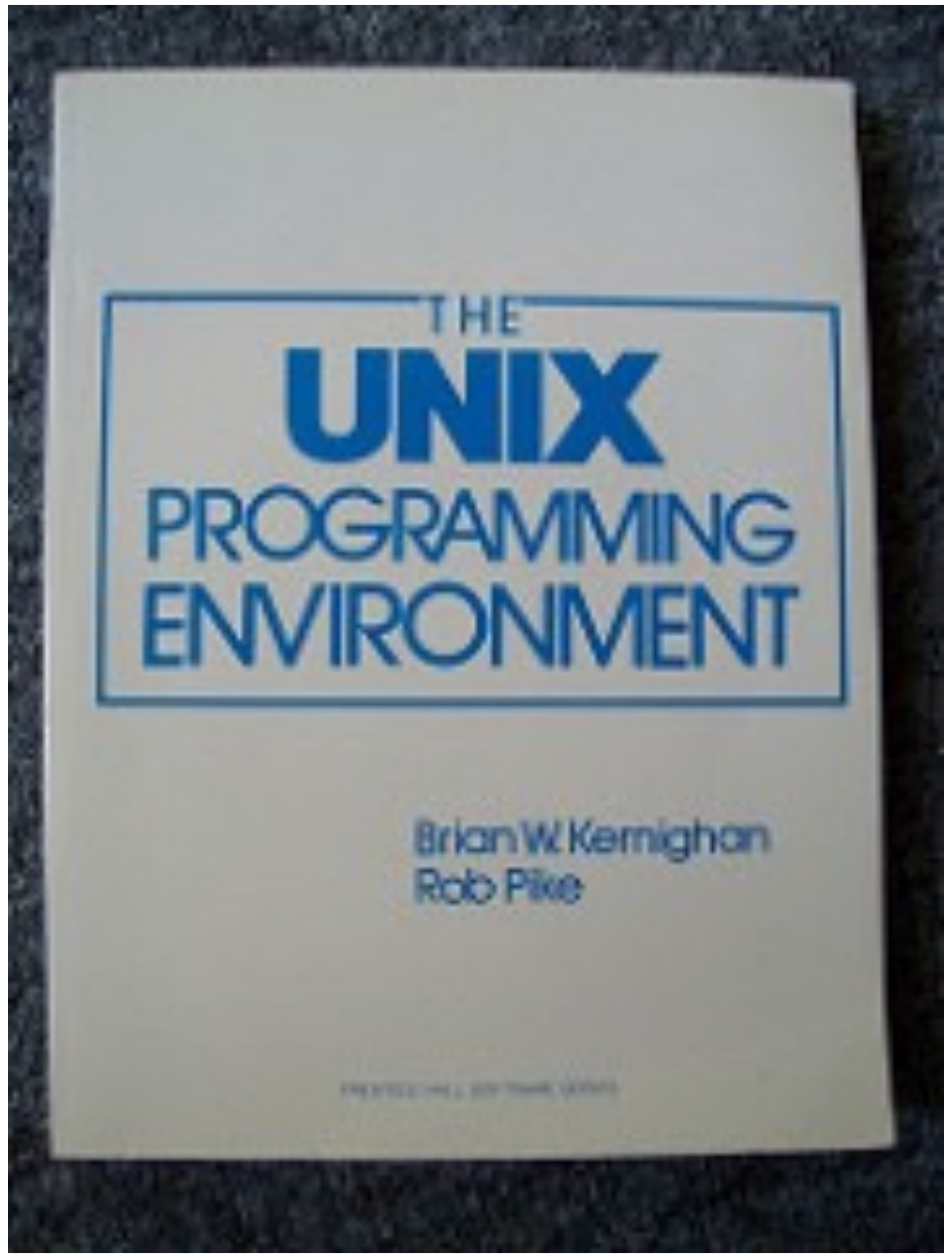
Abraham Maslow


3


3

Regular expressions as tools:

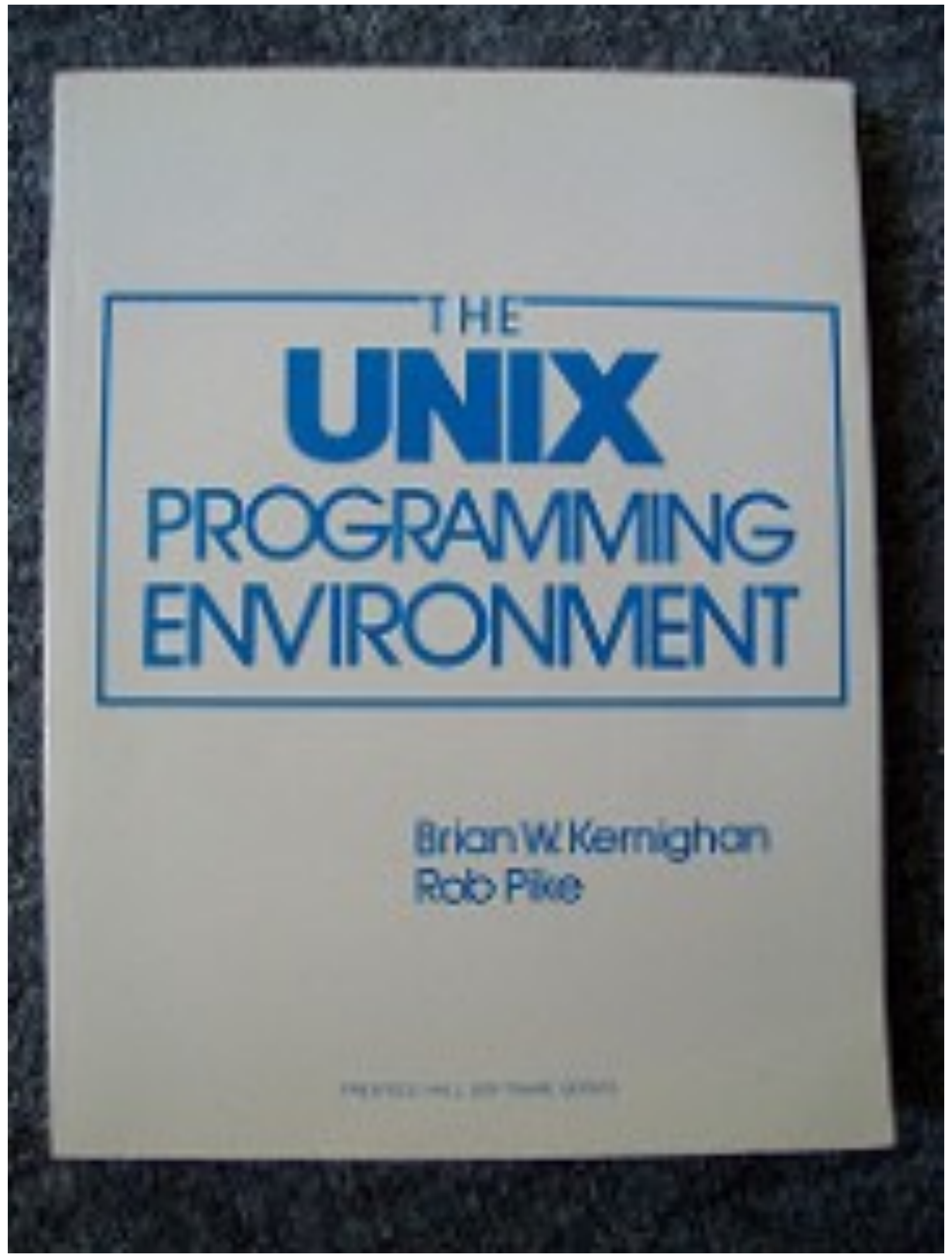




On the command line:

```
grep  -v “^#\|^’\|^\/\/”
egrep -o '((\d{1,3})([.]\d{1,3}){2}|\w+([.]\w+)+)'
sed -e ':a' -e 'N' -e '$!ba' -e 's/\n/ /g'
```

Regular expressions as tools:



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

On the command line:

```
grep  -v “^#\|^’\|^\/\/”
egrep -o '((\d{1,3})([.]\d{1,3}){2}|\w+([.]\w+)+)'
sed -e ':a' -e 'N' -e '$!ba' -e 's/\n/ /g'
```

Regex are notoriously hard to read & maintain

- Dense, **cryptic** syntax
- Semantics **vary** across implementations
- Flags that **affect** the semantics are *not part of the pattern*
- Regex **do not easily compose**

> “Some people, when confronted with a problem, think *‘I know, I'll use regular expressions.’*
> Now they have two problems.”
>
> Jamie Zawinski
> http://regex.info/blog/2006-09-15/247


2


2

Regular expressions

Match a date with slashes, like 1/1/1970:

```
^\d{1,2}\/\d{1,2}\/\d{4}$
```

Match an email address (obviously!):

```
^((?>[a-zA-Z\d!#$%&'*+\-/=?^_`{|}~]+\x20*|"((?
=[\x01-\x7f])[^"\\]|\\[\x01-\x7f])*"\x20*)*(?
<angle><))?((?!\.)(?>\.?[a-zA-Z\d!#$%&'*+\-/=?
^_`{|}~]+)+|"((?=[\x01-\x7f])[^"\\]|\\[\x01-
\x7f])*")@(((?!-)[a-zA-Z\d\-]+(?<!-)\.)+[a-zA-Z]
{2,}|\[(((?(?<!\[)\.)(25[0-5]|2[0-4]\d|[01]?\d?
\d)){4}|[a-zA-Z\d\-]*[a-zA-Z\d]:((?=[\x01-\x7f])
[^\\\[\]]|\\[\x01-\x7f])+)\])(?(angle)>)$
```

Regular expressions

Rosie Pattern Language

Match a date with slashes, like 1/1/1970:

```
^\d{1,2}\/\d{1,2}\/\d{4}$
```

```
date.slashed
```

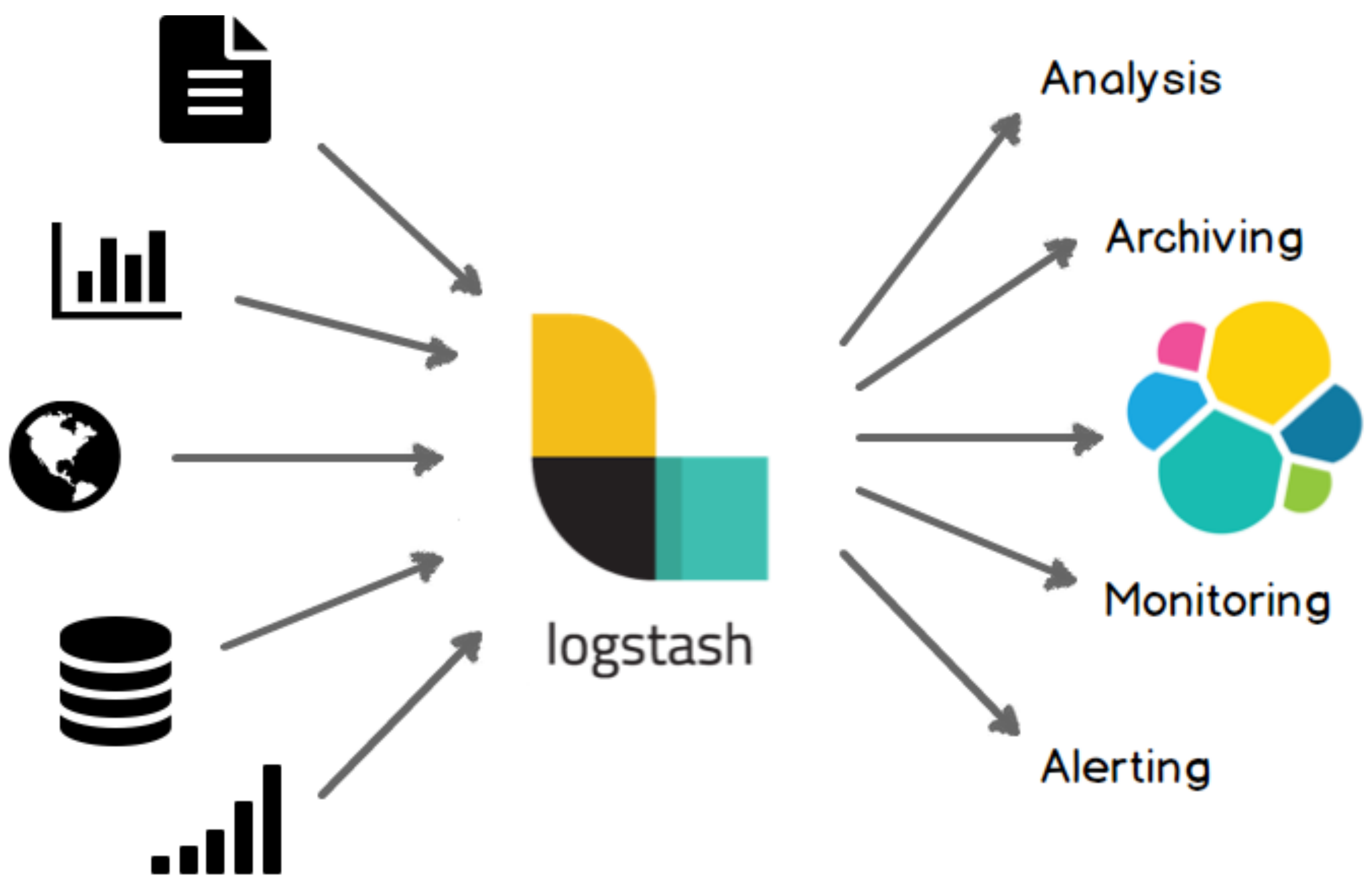
Match an email address (obviously!):

```
^((?>[a-zA-Z\d!#$%&'*+\-/=?^_`{|}~]+\x20*|"((?
=[\x01-\x7f])[^"\\]|\\[\x01-\x7f])*"\x20*)*(?
<angle><))?((?!\.)(?>\.?[a-zA-Z\d!#$%&'*+\-/=?
^_`{|}~]+)+|"((?=[\x01-\x7f])[^"\\]|\\[\x01-
\x7f])*")@(((?!-)[a-zA-Z\d\-]+(?<!-)\.)+[a-zA-Z]
{2,}|\[(((?(?<!\[)\.)(25[0-5]|2[0-4]\d|[01]?\d?
\d)){4}|[a-zA-Z\d\-]*[a-zA-Z\d]:((?=[\x01-\x7f])
[^\\\[\]]|\\[\x01-\x7f])+)\])(?(angle)>)$
```

```
net.email
```

Other regex collections? Grok does this.

Grok sits on top of regular expressions, so any regular expressions are valid in grok as well. The regular expression library is Oniguruma, and you can see the full supported regexp syntax on the Oniguruma site.

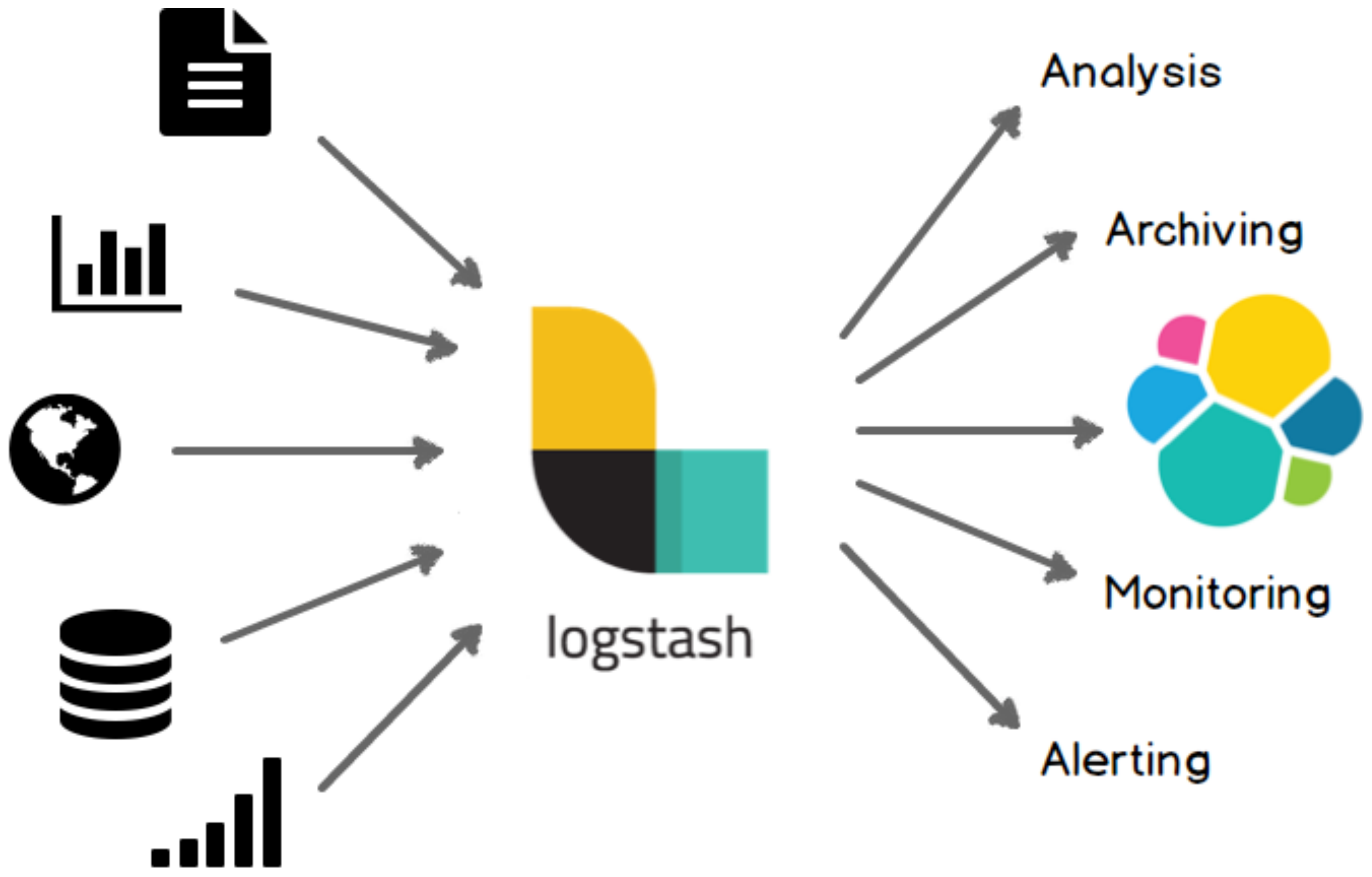


Other regex collections? Grok does this.

> Grok sits on top of regular expressions, so any regular expressions are valid in grok as well. The regular expression library is Oniguruma, and you can see the full supported regexp syntax on the Oniguruma site.

Caveats

- Name collisions? Some versions will use the first seen, some the last
- No packages, hierarchy, or dependencies
- They are still **unreadable** and **unmaintainable**!



And they don’t play well with dev tools

```
grok$ diff orig copy
18c18
< QUOTEDSTRING (?>(?<!\\)(?>"(?>\\.|[^\\"]+)+"|""|(?>'(?>\\.|[^\\']+)+')|''|(?>`(?>\\.|[^\\`]+)+`)|``))
---
> QUOTEDSTRING (?>(?(?<!\)(?>"(?>\\.|[^\\"]+)+"|"|(?>'(?>\\.|[^\\']+)+')|''|(?>`(?>\\.|[^\\']+)+`)|``))
26c26
< IPV6 ((([0-9A-Fa-f]{1,4}:){7}([0-9A-Fa-f]{1,4}|:))|(([0-9A-Fa-f]{1,4}:){6}(:[0-9A-Fa-f]{1,4}|((25[0-5]|2[0-4]\d|1\d\d|[1-9]
?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3})|:))|(([0-9A-Fa-f]{1,4}:){5}(((:[0-9A-Fa-f]{1,4}){1,2})|:((25[0-5]|2[0-4]\d|1\d\
d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3})|:))|(([0-9A-Fa-f]{1,4}:){4}(((:[0-9A-Fa-f]{1,4}){1,3})|((:[0-9A-Fa-f]{1,
4})?:((25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3}))|:))|(([0-9A-Fa-f]{1,4}:){3}(((:[0-9A-Fa-f]{
1,4}){1,4})|((:[0-9A-Fa-f]{1,4}){0,2}:((25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3}))|:))|(([0-9
A-Fa-f]{1,4}:){2}(((:[0-9A-Fa-f]{1,4}){1,5})|((:[0-9A-Fa-f]{1,4}){0,3}:((25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d
|1\d\d|[1-9]?\d)){3}))|:))|(([0-9A-Fa-f]{1,4}:){1}(((:[0-9A-Fa-f]{1,4}){1,6})|((:[0-9A-Fa-f]{1,4}){0,4}:((25[0-5]|2[0-4]\d|1\
d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3}))|:))|(:(((:[0-9A-Fa-f]{1,4}){1,7})|((:[0-9A-Fa-f]{1,4}){0,5}:((25[0-5]
|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3}))|:)))(%.+)?
---
> IPV6 ((([0-9A-Fa-f]{1,4}:){7}([0-9A-Fa-f]{1,4}|:))|(([0-9A-Fa-f]{1,4}:){6}(:[0-9A-Fa-f]{1,4}|((25[0-5]|2[0-4]\d|1\d\d|[1-9]
?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3})|:))|(([0-9A-Fa-f]{1,4}:){5}(((:[0-9A-Fa-f]{1,4}){1,2})|:((25[0-5]|2[0-4]\d|1\d\
d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3})|:))|(([0-9A-Fa-f]{1,4}:){4}(((:[0-9A-Fa-f]{1,4}){1,3})|((:[0-9A-Fa-f]{1,
4})?:((25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3}))|:))|(([0-9A-Fa-f]{1,4}:){3}(((:[0-9A-Fa-f]{
1,4}){1,4})|((:[0-9A-Fa-f]{1,4}){0,3}:((25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3}))|:))|(([0-9
A-Fa-f]{1,4}:){2}(((:[0-9A-Fa-f]{1,4}){1,5})|((:[0-9A-Fa-f]{1,4}){0,3}:((25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d
|1\d\d|[1-9]?\d)){3}))|:))|(([0-9A-Fa-f]{1,4}:){1}(((:[0-9A-Fa-f]{1,4}){1,6})|((:[0-9A-Fa-f]{1,4}){0,3}:((25[0-5]|2[0-4]\d|1\
d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3}))|:))|(:(((:[0-9A-Fa-f]{1,4}){1,7})|((:[0-9A-Fa-f]{1,4}){0,5}:((25[0-5]
|2[0-4]\d|1\d\d|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d\d|[1-9]?\d)){3}))|:)))(%.+)?
grok$
```

And they don’t play well with dev tools

RPL has syntax like a programming language

→ It reads like code
→ It diffs like code
→ It debugs like code


1


1

Regex performance is surprisingly variable

Regular expression matching **can be very efficient**: linear time in the size of the input.

"The worst-case exponential-time backtracking strategy [is] used almost everywhere [but grep and RE2], including ed, sed, Perl, PCRE, and Python."

(Russ Cox https://swtch.com/~rsc/regexp/regexp2.html)

Regex performance is surprisingly variable

Matching this 29-character string takes around <u>36 seconds</u> in Perl*

$input = “aaaaaaaaaaaaaaaaaaaaaaaaaaaaa”;

$re = “a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa”;

And this more realistic example takes around <u>65 seconds</u> in Perl*

$input = “1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
Bronze,Bronze,Gold,Silver”;

$re = “^(.*?,){29}Gold”;

(*) Perl 5.16.3 darwin-thread-multi-2level

Regex performance is surprisingly variable

In RPL, expressions are greedy and possessive.

→ Backtracking is explicit
→ To get exponential backtracking, you write it that way
→ Today (v1.1.x) such RPL patterns have exponential size

RPL makes it difficult to be accidentally inefficient.

(*) Perl 5.16.3 darwin-thread-multi-2level

Rosie Pattern Language

"All progress depends on the unreasonable [woman]"

George Bernard Shaw, paraphrased

RPL is designed 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 is designed like a programming language

- Comments
- Modules
- Identifiers
- Whitespace
- Quoted literals
- Unit tests
- Macros (not shown)

```
---- -*- 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}]"
```

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
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
$
```

NAMED PATTERNS

-o Output format
subs ==> sub-matches

pattern net.url_common
==> 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
$
$ 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": "*"}
$
```

Matching line

num.denoted_hex

num.hex, a sub-match

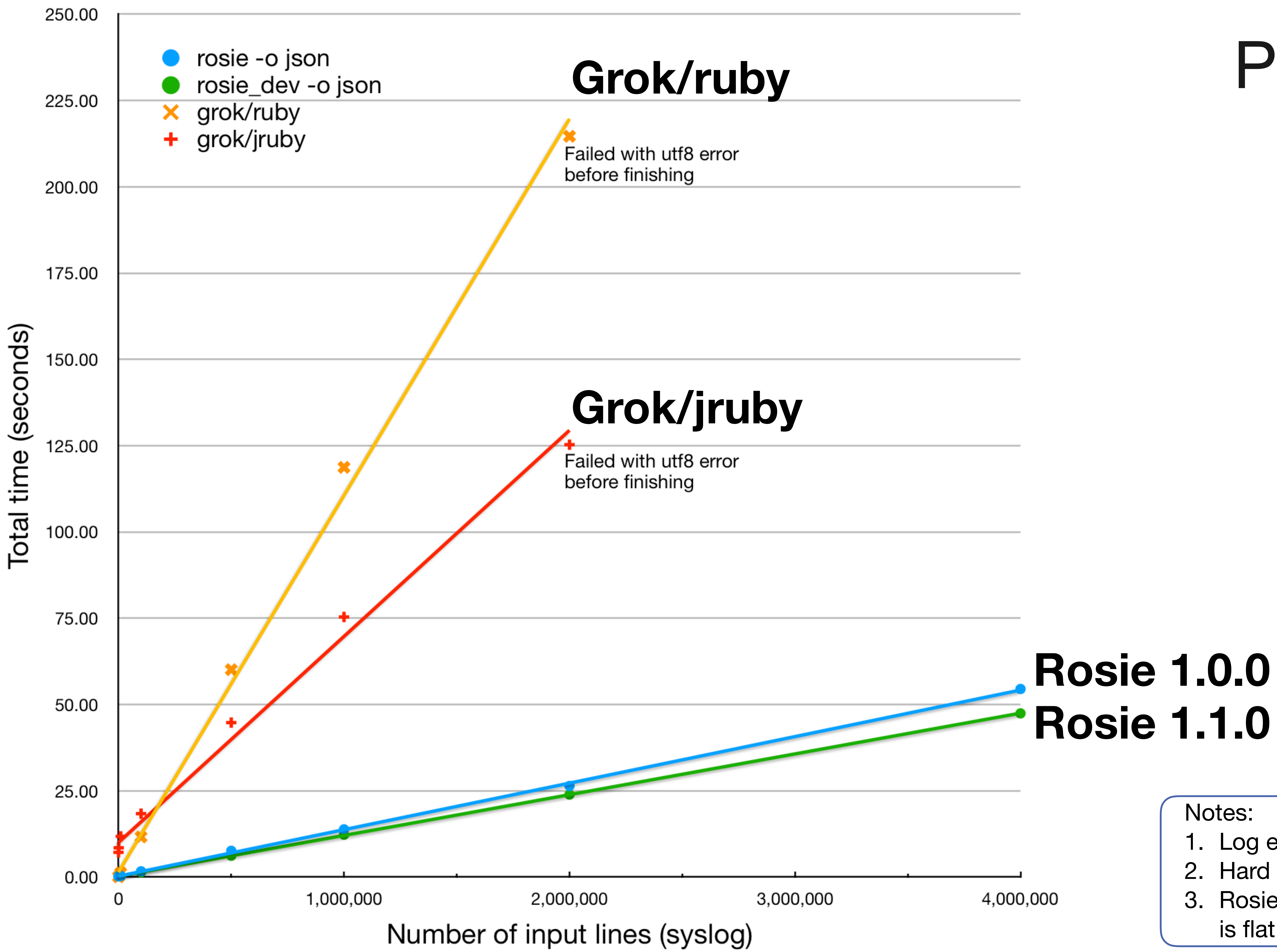
Rosie Implementation
Pattern Library
+
Your patterns
RPL Compiler
Matching Engine
json
color
boolean
{"s": 1,
"e": 12,
"type": "net.any",
"data": "192.168.0.1",
"subs":
[{"s": 1,
"e": 12,
"type": "net.ip",
"data": "192.168.0.1",
"subs":
[{"s": 1,
"e": 12,
"type": "net.ipv4",
"data": "192.168.0.1"}]
}]
}
192.168.0.1
1
big data
storage
search
software
future
development
analysis
based
san
petabytes
internet
technologies
technology
process
research
information
world
science
large
sets
new
billion
million
learning
management

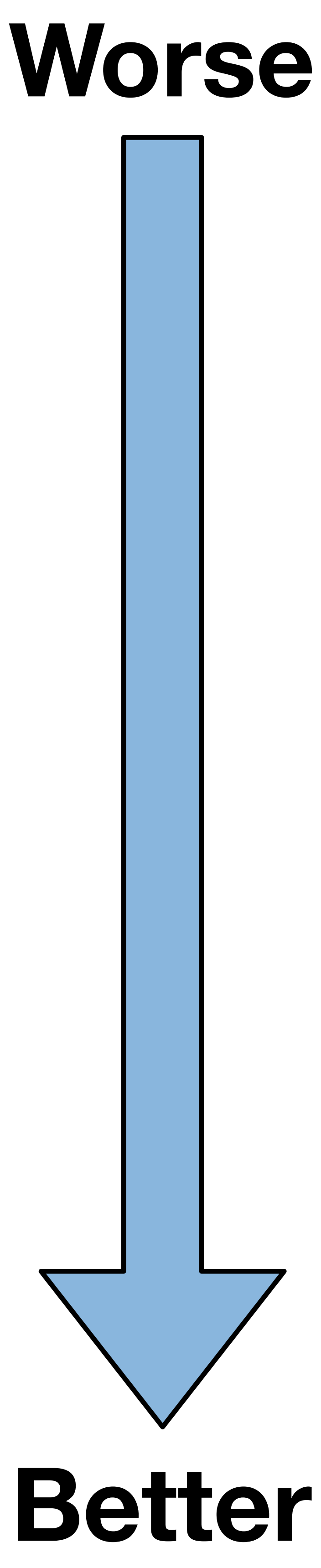
Performance

“I want to believe”

Fox Mulder, FBI

Performance





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.

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>
```

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
```

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
```

Executable unit tests

```
----
---- net.rpl     Rosie Pattern Language patterns for hostnames, ip addresses, and such
----

package net
import num

[snip]

ipv4 = ip_address_v4
-- test ipv4 accepts "0.0.0.0", "1.2.234.123", "999.999.999.999"
-- test ipv4 rejects "1234.1.2.3", "1.2.3", "111.222.333.", "111.222.333..444"

ipv6 = ipv6_mixed / ip_address_v6
-- test ipv6 includes ipv4 "::192.9.5.5", "::FFFF:129.144.52.38"
-- test ipv6 excludes ipv4 "1080::8:800:200C:417A", "2010:836B:4179::836B:4179"
```

Executable unit tests

```
$ rosie test /usr/local/lib/rosie/rpl/*.rpl
/usr/local/lib/rosie/rpl/all.rpl
    all 4 tests passed
/usr/local/lib/rosie/rpl/csv.rpl
    no tests found
/usr/local/lib/rosie/rpl/date.rpl
    all 89 tests passed
/usr/local/lib/rosie/rpl/id.rpl
    all 51 tests passed
/usr/local/lib/rosie/rpl/json.rpl
    all 45 tests passed
/usr/local/lib/rosie/rpl/net.rpl
    all 125 tests passed
/usr/local/lib/rosie/rpl/num.rpl
    all 80 tests passed
/usr/local/lib/rosie/rpl/os.rpl
    no tests found
/usr/local/lib/rosie/rpl/time.rpl
    all 85 tests passed
/usr/local/lib/rosie/rpl/ts.rpl
    all 27 tests passed
/usr/local/lib/rosie/rpl/word.rpl
    all 20 tests passed
$
```

- ☑ Part of the documentation
- ☑ Regression when making changes
- ☑ Use them in app build/compile stage

Formal basis

"Language is a process of free creation [though] its laws and principles are fixed"

Noam Chomsky

Comparison to regex: RPL is based on a different formalism

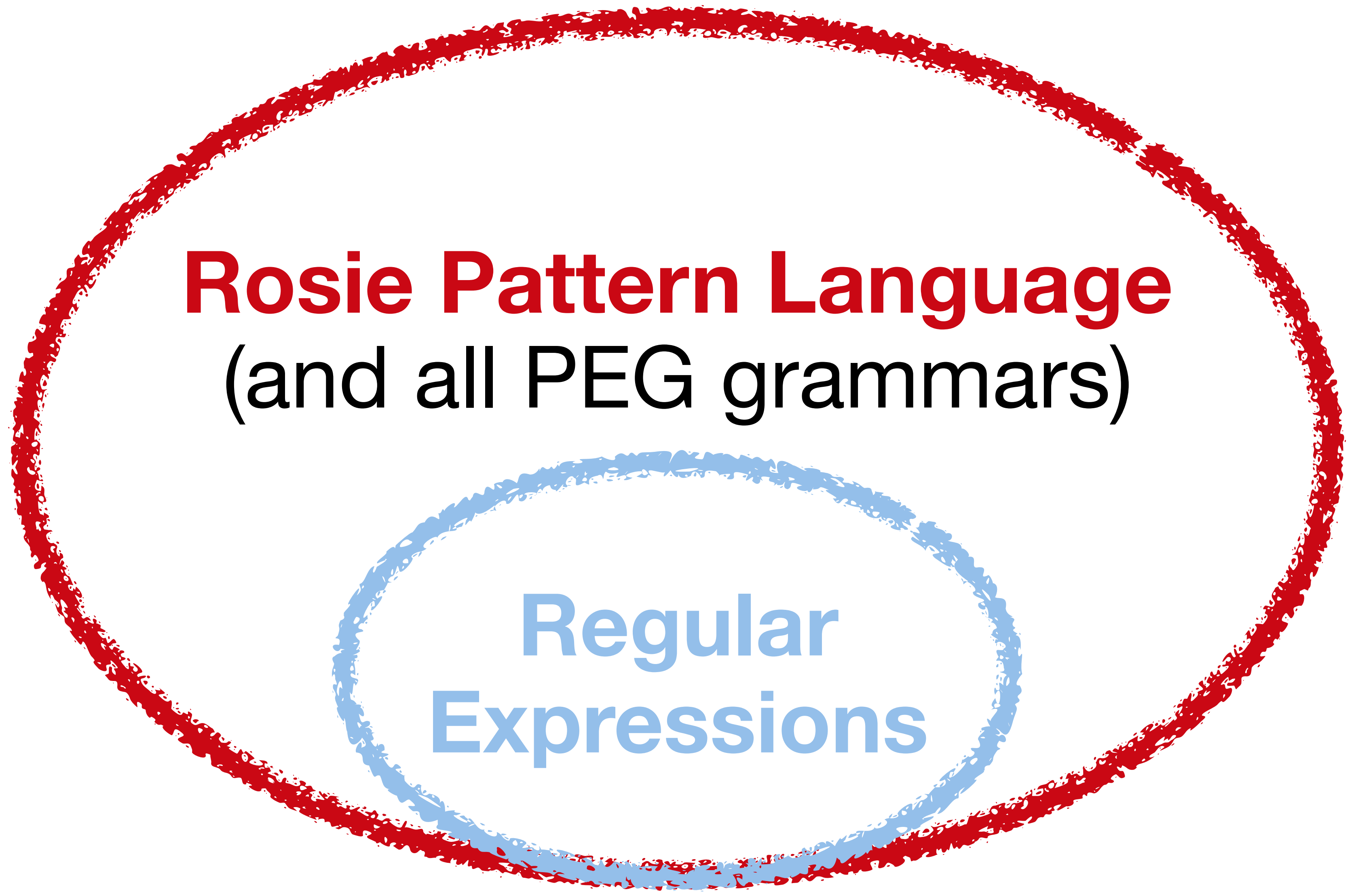
Comparison to regex: RPL is based on a different formalism

Parsing Expression Grammars

Comparison to regex: RPL is based on a different formalism

Parsing Expression Grammars

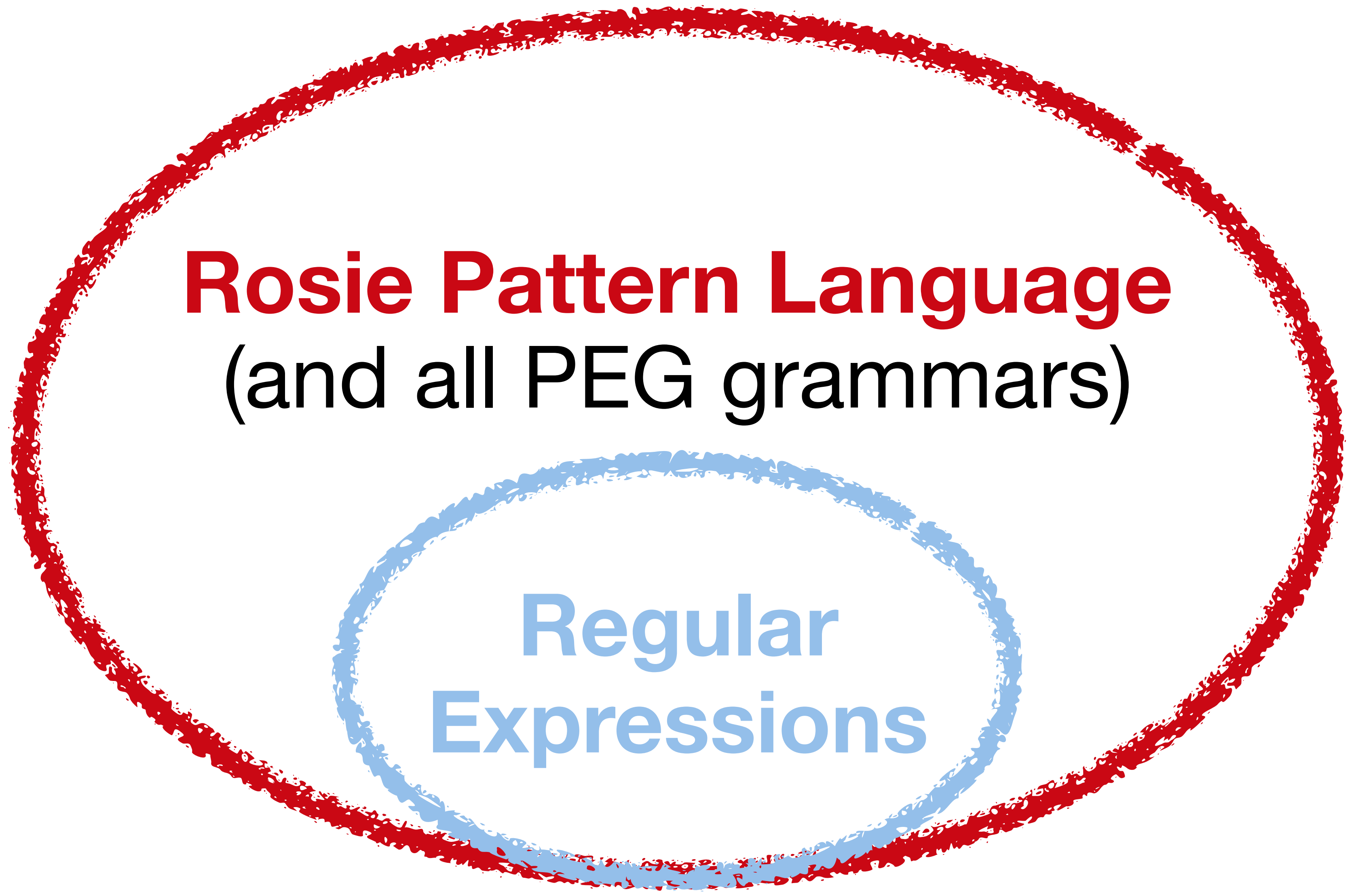
- Strictly more powerful than regular expressions



Comparison to regex: RPL is based on a different formalism

Parsing Expression Grammars

- Strictly more powerful than regular expressions
- Supports recursive pattern definitions

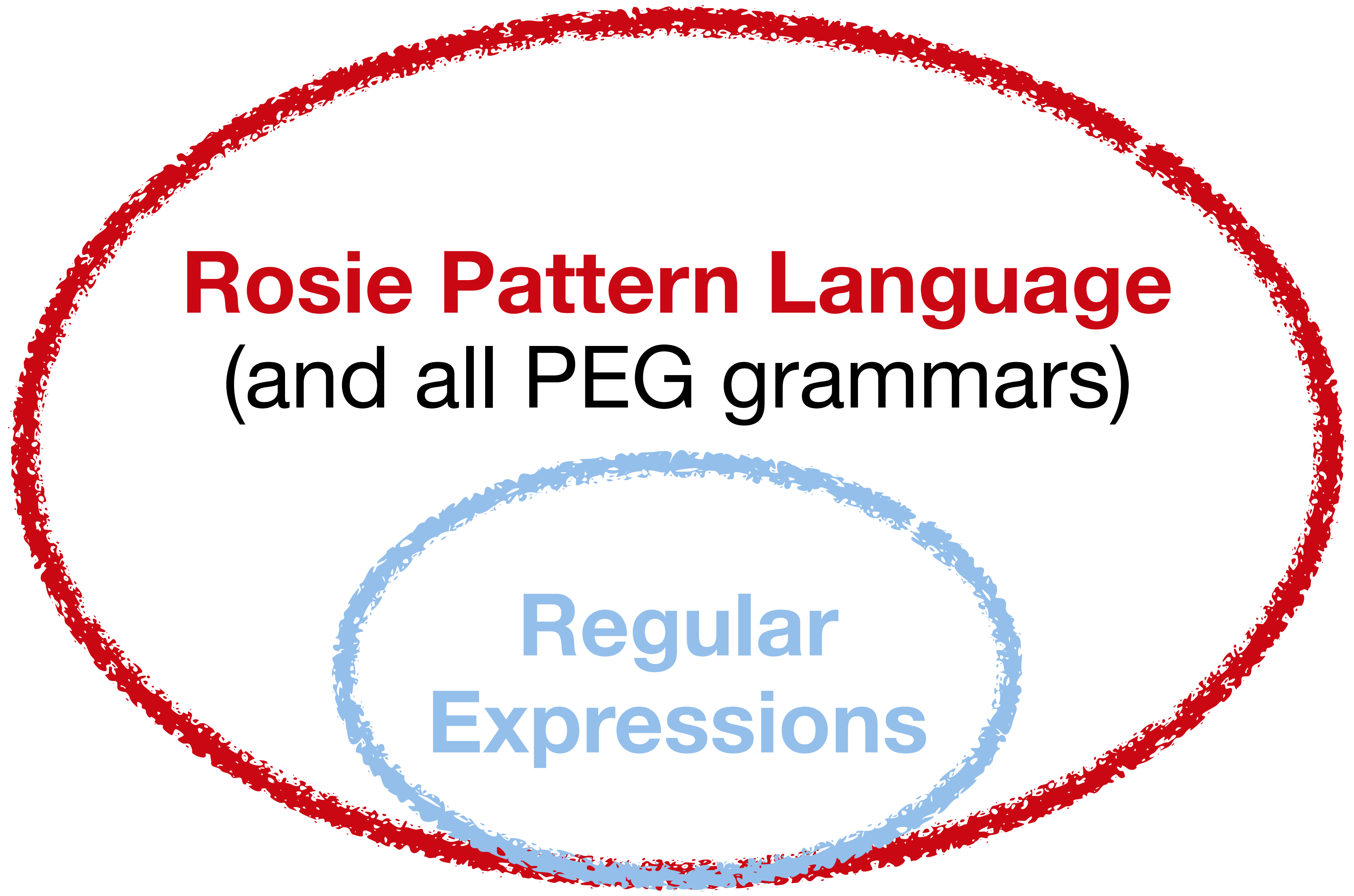



```
grammar
   bal = { "(" bal? ")" }+
end
```

Comparison to regex: RPL is based on a different formalism

Parsing Expression Grammars

- Strictly more powerful than regular expressions
- Supports recursive pattern definitions



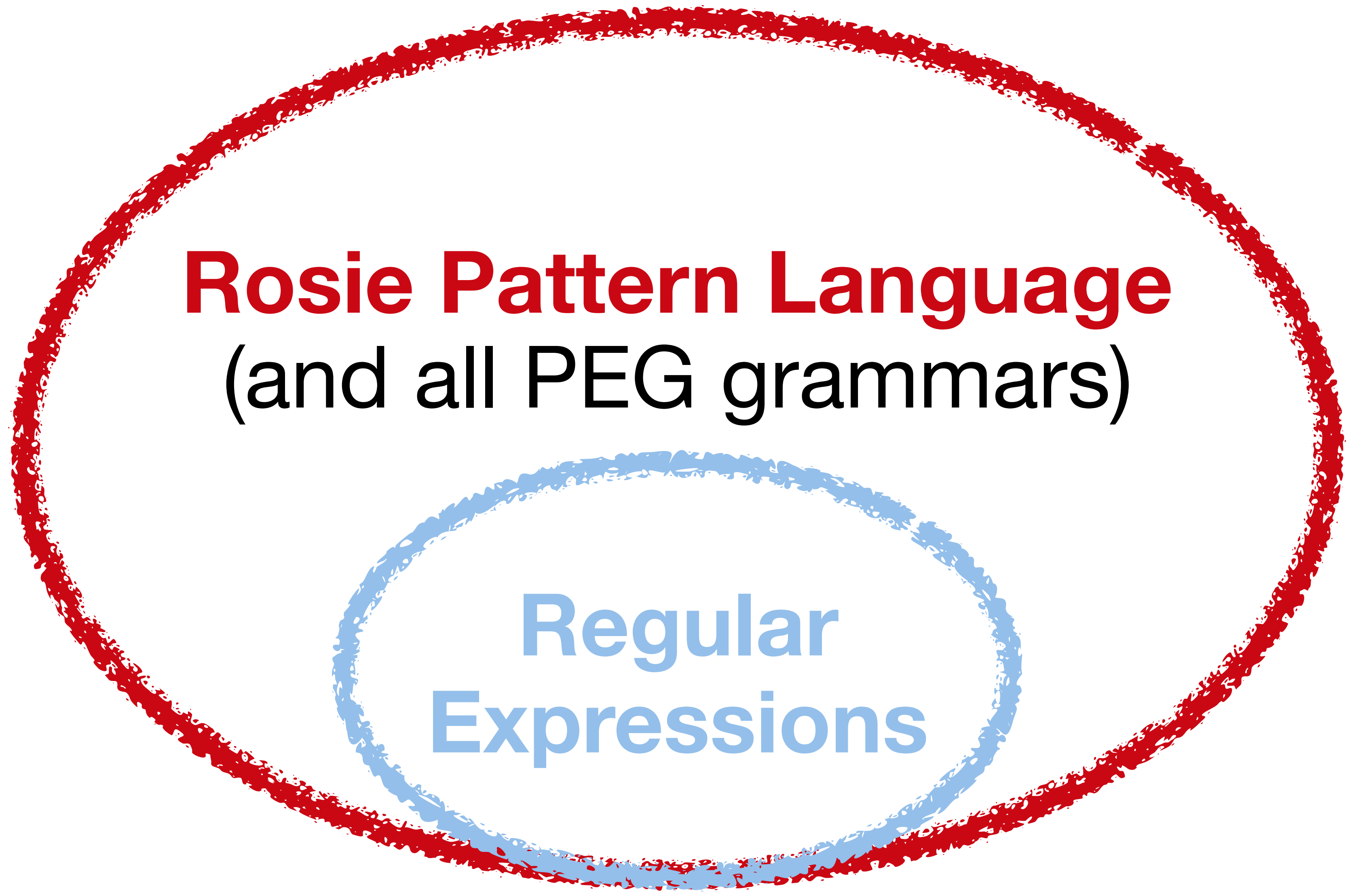

```
grammar
   bal = { "(" bal? ")" }+
end
```

Perl: `(^(\((?-1)?\))+$)`

Comparison to regex: RPL is based on a different formalism

Parsing Expression Grammars

- Strictly more powerful than regular expressions
- Supports recursive pattern definitions
- *Packrat* implementation guarantees linear time

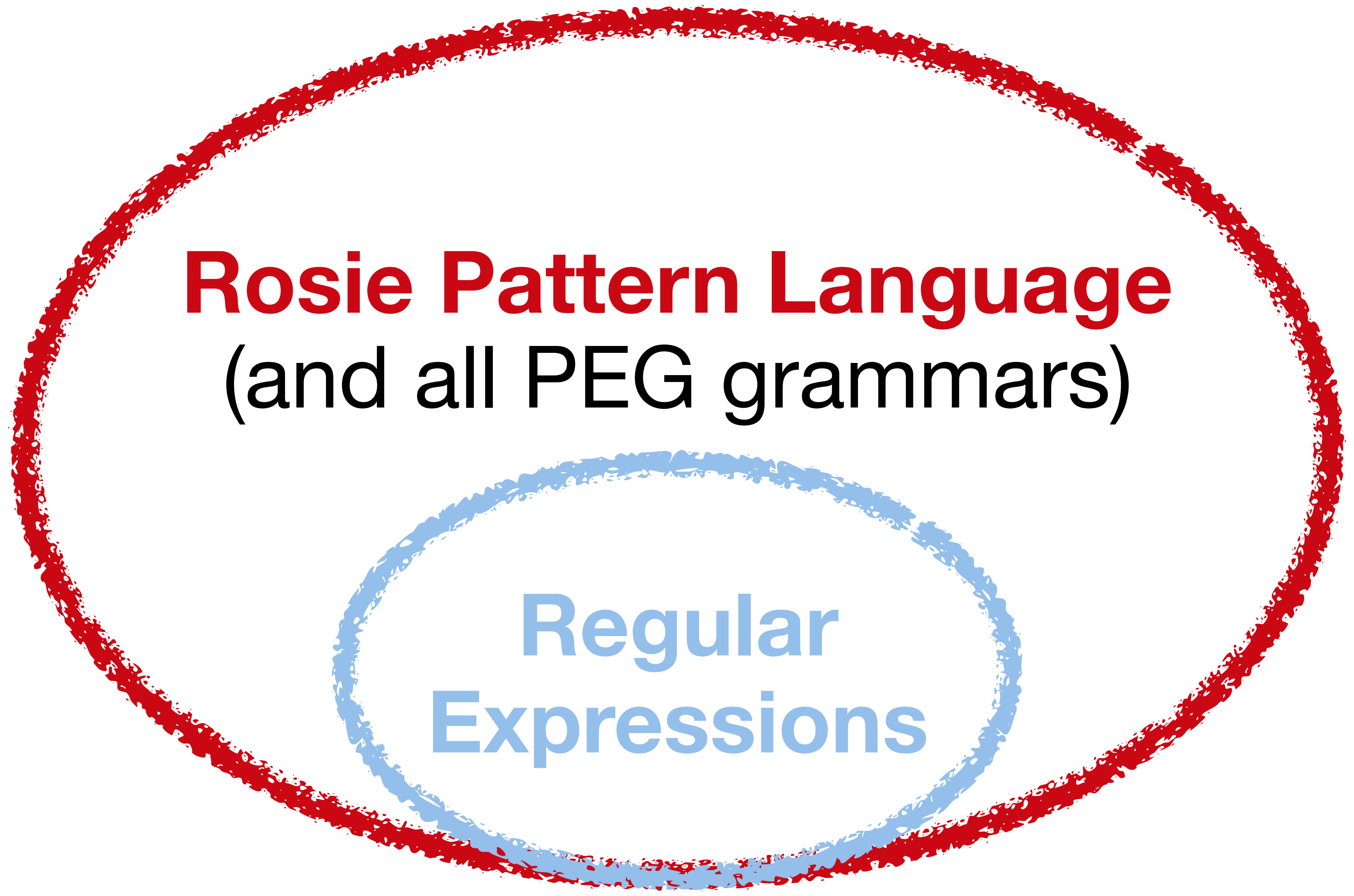



```
grammar
   bal = { "(" bal? ")" }+
end
```

Comparison to regex: RPL is based on a different formalism

Parsing Expression Grammars

- Strictly more powerful than regular expressions
- Supports recursive pattern definitions
- *Packrat* implementation guarantees linear time
- Rosie uses a *Matching VM* implementation
  - Uses less space
  - Linear time for non-grammar, non-lookaround

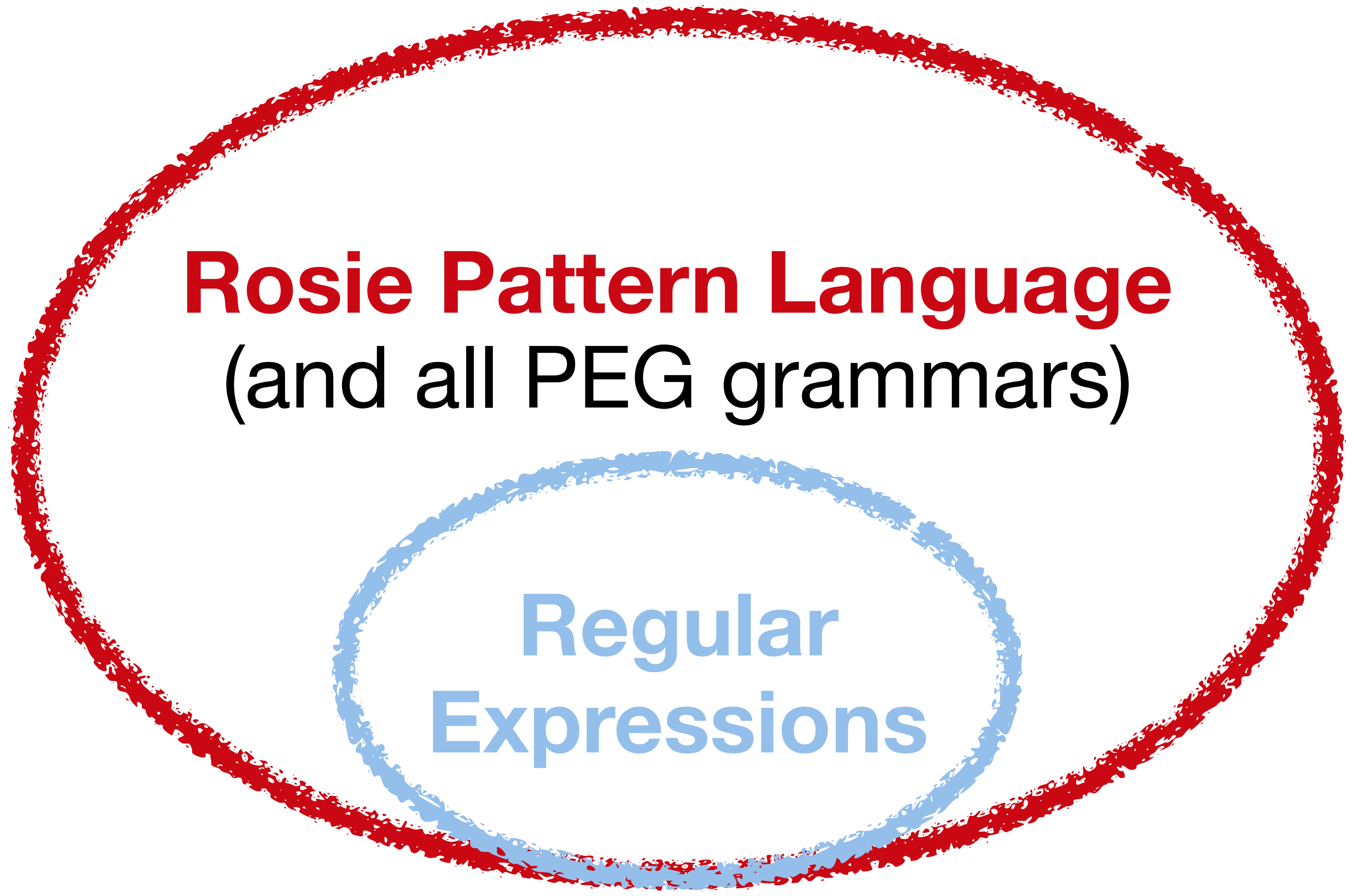



```
grammar
   bal = { "(" bal? ")" }+
end
```

Comparison to regex: RPL is based on a different formalism

Parsing Expression Grammars

- Strictly more powerful than regular expressions
- Supports recursive pattern definitions
- *Packrat* implementation guarantees linear time
- Rosie uses a *Matching VM* implementation
  - Uses less space
  - Linear time for non-grammar, non-lookaround
- Expressions are greedy and possessive




```
grammar
   bal = { "(" bal? ")" }+
end
```

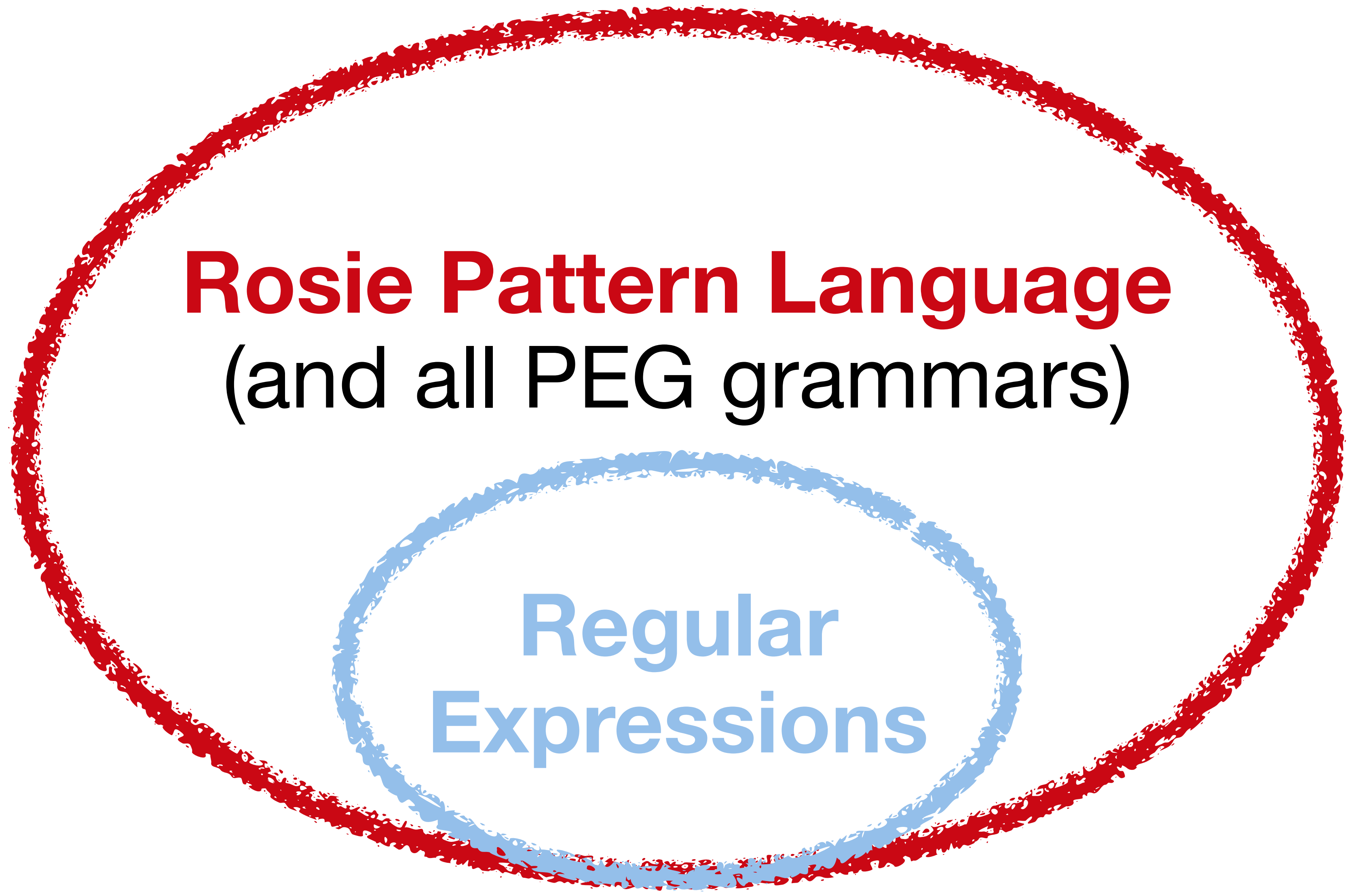
`.* "x"` *always fails!*

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

Comparison to regex: RPL is based on a different formalism

Parsing Expression Grammars

- Strictly more powerful than regular expressions
- Supports recursive pattern definitions
- *Packrat* implementation guarantees linear time
- Rosie uses a *Matching VM* implementation
  - Uses less space
  - Linear time for non-grammar, non-lookaround
- Expressions are greedy and possessive




```
grammar
   bal = { "(" bal? ")" }+
end
```

`.* "x"` *always fails!*

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

```
find:"x"
```

Comparison to regex: RPL syntax is familiar

RPL gives a concrete syntax for PEGs

- Sequences built by adjacency, plus tokenizing
- Ordered choice expressed by /
- Repetition operators * **+ ? {n,m}**
- Look ahead **>**, look behind **<**, negation **!**
- Character sets with more strict syntax but extended composability
- *Plus declarations, packages, macros*

Expression	**Matches**
`{"a" "b"}`	`ab`
`("a" "b")`	`a b`
`"a" "b"`	`a b`

Regex	**RPL**
`a|b`	`"a" / "b"`
`a|bc`	`"ac" / "bc"`

`[[_ ] Script.Greek]+`

matches

`_Γειά σου Κόσμε_`

Using Rosie in programs

Today:

THE C PROGRAMMING LANGUAGE

python™

Haskell

Go

Once and future:

node JS®

Ruby PROGRAMMING Language

Java™ ORACLE®

Lua

Clojure

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

Faster

- Dev time:
  - ✓ library of patterns
  - ✓ composable patterns
  - ✓ 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)

Thank you!

Additional slides follow…

Patterns in the standard library (v1.0.0)

- Collections
  - `net.any, date.any,` etc.
  - `all.things`
- Commonly needed
  - int, float, hex, and other numbers
  - several kinds of identifiers
  - path names for Unix and Windows
  - GUIDs
- Network patterns
  - ip address (v4, v6, mixed), domain name, email address, url, URI, MAC, HTTP
- Timestamps
  - RFC3339, RFC2822, and more than a dozen other common formats

- CSV data
  - delimiters: , ; |
  - quoted fields: “foo” or ‘bar’
  - escapes: "" or \" or \"\”
- JSON data
  - full parse
  - match nested and balanced {} []

- Source code features
  - 10 popular languages
- De-structuring
  - E.g. “CSC316” ==> “CSC”, “316”
  - E.g. “(1.2, 3.77, 0)” ==> “1.2”, “3.77”, “0”
- Log files
  - syslog constituents (covers most log files)
  - Java exceptions, Python tracebacks

Community

Join the Rosie community!

```
make;
make install (optional)
```

Contribute Patterns

- Domain-specific
- Authoritative
  - E.g. from RFC
- Non-English patterns!
- "Looks like" (recognizers)
- Byte-encoded data?

Write Tools

- Package info
- Better trace (compact)
- Linter
- Notebook (Jupyter?)
- Integrations
  - scikit-learn
  - Spark

Implement features

- Optimizations
- Language-specific libs
  - Improve or create
  - Python, R, Go, Java, …
- User-written extensions
  - Output encoders
  - Macros
  - Character sets

Join the Rosie community!

```
make;
make install
```
(optional)

Or: brew install rosie

Also: pip install rosie

Contribute Patterns

- Domain-specific
- Authoritative
  - E.g. from RFC
- Non-English patterns!
- “Looks like” (recognizers)
- Byte-encoded data?

Write Tools

- Package info
- Better trace (compact)
- Linter
- Notebook (Jupyter?)
- Integrations
  - scikit-learn
  - Spark

Implement features

- Optimizations
- Language-specific libs
  - Improve or create
  - Python, R, Go, Java, …
- User-written extensions
  - Output encoders
  - Macros
  - Character sets

1. Mining source code repositories

- “Micro-grammar” approach:

  How to build static checking systems using orders of magnitude less code by Brown, Nötzli, Engler

- NCSU students: 6 features x 10 languages

Features → Languages ↓	Comments	Dependencies	Class / Struct Defs	Function Defs	Error Handling	String Literals	Function Bodies	Class Bodies
Java	✓	✓	✓	✓	✓	✓		
C	✓	✓	✓	✓	✓	✓		
C++	✓	✓	✓	✓	✓	✓		
C#	✓	✓	✓	✓	✓	✓		
Python	✓	✓	✓	✓	✓	✓		
JScript	✓	✓	✓	✓	✓	✓		
Ruby	✓	✓	✓	✓	✓	✓		
R	✓	✓	✗	✓	✓	✓		
Go	✓	✓	✓	✓	✓	✓		
Bash	✓	✗	✗	✓	✓	✓		
VB	✓	✓	✓	✓	✓	✓		

1. Mining source code repositories

- “Micro-grammar” approach:

  How to build static checking systems using orders of magnitude less code by Brown, Nötzli, Engler

- NCSU students: 6 features x 10 languages

Features → Languages ↓	Comments	Dependencies	Class / Struct Defs	Function Defs	Error Handling	String Literals	Function Bodies	Class Bodies
Java	✓	✓	✓	✓	✓	✓		
C	✓	✓	✓	✓	✓	✓		
C++	✓	✓	✓	✓	✓	✓		
C#	✓	✓	✓	✓	✓	✓		
Python	✓	✓	✓	✓	✓	✓		
JScript	✓	✓	✓	✓	✓	✓		
Ruby	✓	✓	✓	✓	✓	✓		
R	✓	✓	✗	✓	✓	✓		
Go	✓	✓	✓	✓	✓	✓		
Bash	✓	✗	✗	✓	✓	✓		
VB	✓	✓	✓	✓	✓	✓		

2. Application log processing (streaming or batch)

```
$ tail -n 3 /var/log/system.log | rosie match all.things
Jul 29 09:48:58 Jamies-Compabler com.apple.xpc.launchd[1] (com.apple.quicklook): Service only ran for
Jul 29 09:48:59 Jamies-Compabler com.apple.xpc.launchd[1] (com.apple.quicklook[91387]): Endpoint has be
Jul 29 09:48:59 Jamies-Compabler kcm[91389]: DEPRECATED USE in libdispatch client: Setting timer interv
```

1. Mining source code repositories

- "Micro-grammar" approach:

  How to build static checking systems using orders of magnitude less code by Brown, Nötzli, Engler

- NCSU students: 6 features x 10 languages

eatures → anguages ↓	Comments	Dependencies	Class / Struct Defs	Function Defs	Error Handling	String Literals	Function Bodies	Class Bodies
Java	✓	✓	✓	✓	✓	✓		
C	✓	✓	✓	✓	✓	✓		
C++	✓	✓	✓	✓	✓	✓		
C#	✓	✓	✓	✓	✓	✓		
Python	✓	✓	✓	✓	✓	✓		
JScript	✓	✓	✓	✓	✓	✓		
Ruby	✓	✓	✓	✓	✓	✓		
R	✓	✓	✗	✓	✓	✓		
Go	✓	✓	✓	✓	✓	✓		
Bash	✓	✗	✗	✓	✓	✓		
VB	✓	✓	✓	✓	✓	✓		

2. Application log processing (streaming or batch)

```
$ tail -n 3 /var/log/system.log | rosie match all.things
Jul 29 09:48:58 Jamies-Compabler com.apple.xpc.launchd[1] (com.apple.quicklook): Service only ran for 0
Jul 29 09:48:59 Jamies-Compabler com.apple.xpc.launchd[1] (com.apple.quicklook[91387]): Endpoint has be
Jul 29 09:48:59 Jamies-Compabler kcm[91389]: DEPRECATED USE in libdispatch client: Setting timer interv
```

3. Secure engineering principle: *Parse everything!*

The most critical risk in every OWASP report since 2003: Injection attacks (unvalidated input)

Best practice: Whitelist valid input, which requires parsing every input

RPL: Familiar concepts (and syntax)

RPL expression	Matches
`pat*`	Zero or more copies of `pat`
`pat+`	One or more copies of `pat`
`pat?`	Zero or one copies of `pat`
`pat{n}`	Exactly `n` copies of `pat`

RPL expression	Meaning
`[:name:]`	Named character set (see *note [a]*)
`[:^name:]`	Complement of a named character set
`[x-y]`	Range of characters, from `x` to `y` (see *note [b]*)
`[^x-y]`	Complement of a character range
`[...]`	List of characters (in place of `...`)
`[^...]`	Complement of the character list `...`
`[cs1 cs2 ...]`	Union of character sets `cs1`, `cs2`, etc. (E.g. `[[a-f][0-9]]`)
`[^ cs1 cs2 ...]`	Complement of a union of character sets

RPL expression	Meaning
`> pat`	Look ahead at `pat` (predicate: consumes no input)
`< pat`	Look behind at `pat` (predicate: consumes no input)
`!pat`	Not `pat`, i.e. not looking at `pat`. Same as `!>pat`.

RPL expression	Meaning
`p / q`	Ordered choice: match `p`, and `p` fails, match `q`

RPL: Familiar concepts (and syntax)

RPL expression	Matches
`pat*`	Zero or more copies of `pat`
`pat+`	One or more copies of `pat`
`pat?`	Zero or one copies of `pat`
`pat{n}`	Exactly `n` copies of `pat`

RPL expression	Meaning
`[:name:]`	Named character set (see *note [a]*)
`[:^name:]`	Complement of a named character set
`[x-y]`	Range of characters, from `x` to `y` (see *note [b]*)
`[^x-y]`	Complement of a character range
`[...]`	List of characters (in place of `...`)
`[^...]`	Complement of the character list `...`
`[cs1 cs2 ...]`	Union of character sets `cs1`, `cs2`, etc. (E.g. `[[a-f][0-9]]`)
`[^ cs1 cs2 ...]`	Complement of a union of character sets

RPL expression	Meaning
`> pat`	Look ahead at `pat` (predicate: consumes no input)
`< pat`	Look behind at `pat` (predicate: consumes no input)
`!pat`	Not `pat`, i.e. not looking at `pat`. Same as `!>pat`.

RPL expression	Meaning
`p / q`	Ordered choice: match `p`, and `p` fails, match `q`

Differences

- Always greedy
- Always possessive
- Choices are ordered

RPL: Familiar concepts (and syntax)

RPL expression	Matches
`pat*`	Zero or more copies of `pat`
`pat+`	One or more copies of `pat`
`pat?`	Zero or one copies of `pat`
`pat{n}`	Exactly `n` copies of `pat`

RPL expression	Meaning
`[:name:]`	Named character set (see *note [a]*)
`[:^name:]`	Complement of a named character set
`[x-y]`	Range of characters, from `x` to `y` (see *note [b]*)
`[^x-y]`	Complement of a character range
`[...]`	List of characters (in place of `...`)
`[^...]`	Complement of the character list `...`
`[cs1 cs2 ...]`	Union of character sets `cs1`, `cs2`, etc. (E.g. `[[a-f][0-9]]`)
`[^ cs1 cs2 ...]`	Complement of a union of character sets

RPL expression	Meaning
`> pat`	Look ahead at `pat` (predicate: consumes no input)
`< pat`	Look behind at `pat` (predicate: consumes no input)
`!pat`	Not `pat`, i.e. not looking at `pat`. Same as `!>pat`.

RPL expression	Meaning
`p / q`	Ordered choice: match `p`, and `p` fails, match `q`

Differences

- Always greedy
- Always possessive
- Choices are ordered

To regex, and beyond!

- Structured output, not flat
- Sane syntax
- A few key concepts
- Auto tokenization
- Package system
- Macros
- Unit tests
- REPL
- Trace output

Roadmap

"If you want to go fast, go alone.
If you want to go far, go together."

"Proverb"

Roadmap

Roadmap

Extensibility
User-written macros
User-written output encoders

Pattern generation
Algorithmic, e.g. from static analysis
Statistical / ML

Command line/scripting convenience
Traverse directories
Follow links or not, etc.

Compiler Optimizations
Common subexpression elimination
New vm instructions
Flow analysis

Regex-to-rosie converter
Re-use existing regex
Give them unit tests
Debug them

Ahead of time compilation
Fast startup
Small matching run-time (~50Kb binary)

Rosie Pattern Language

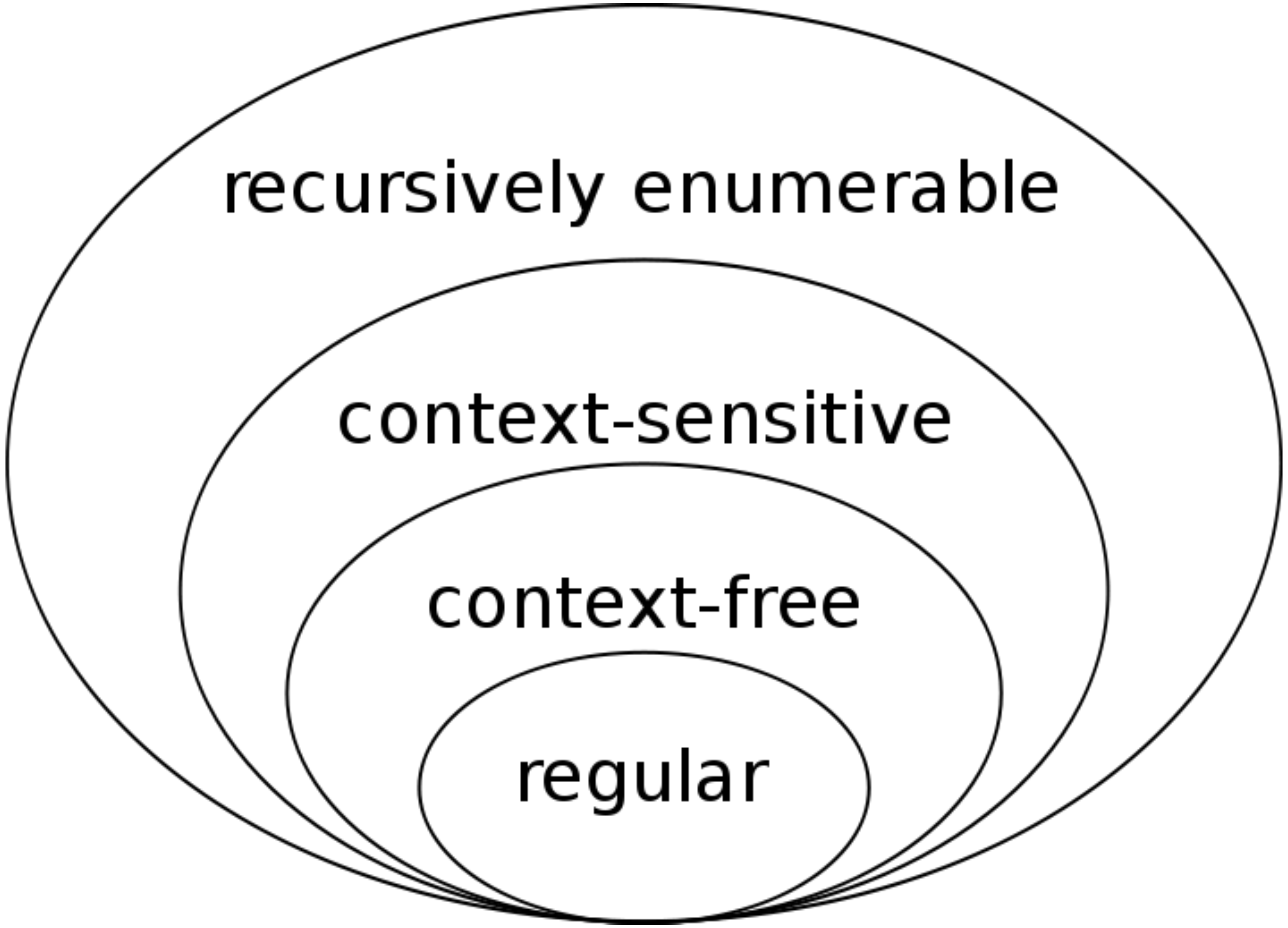
- **Pattern libraries**
  - Standard library, including full Unicode (UTF-8) support
  - Community libraries (e.g. GitHub)
  - User libraries
- **Output formats**
  - Colorized text for humans
  - JSON for programs
  - Full lines or just matches (like grep)
  - And others…
- **Development tools**
  - Command line interface, read/eval/print loop
  - Trace output
  - Unit tests (automated)
  - Packages (shareable)
- **Built for big data** but makes a better `grep`
  - Readable, maintainable
  - Works well with git/diff, pipelines (unit tests), dependency mgmt

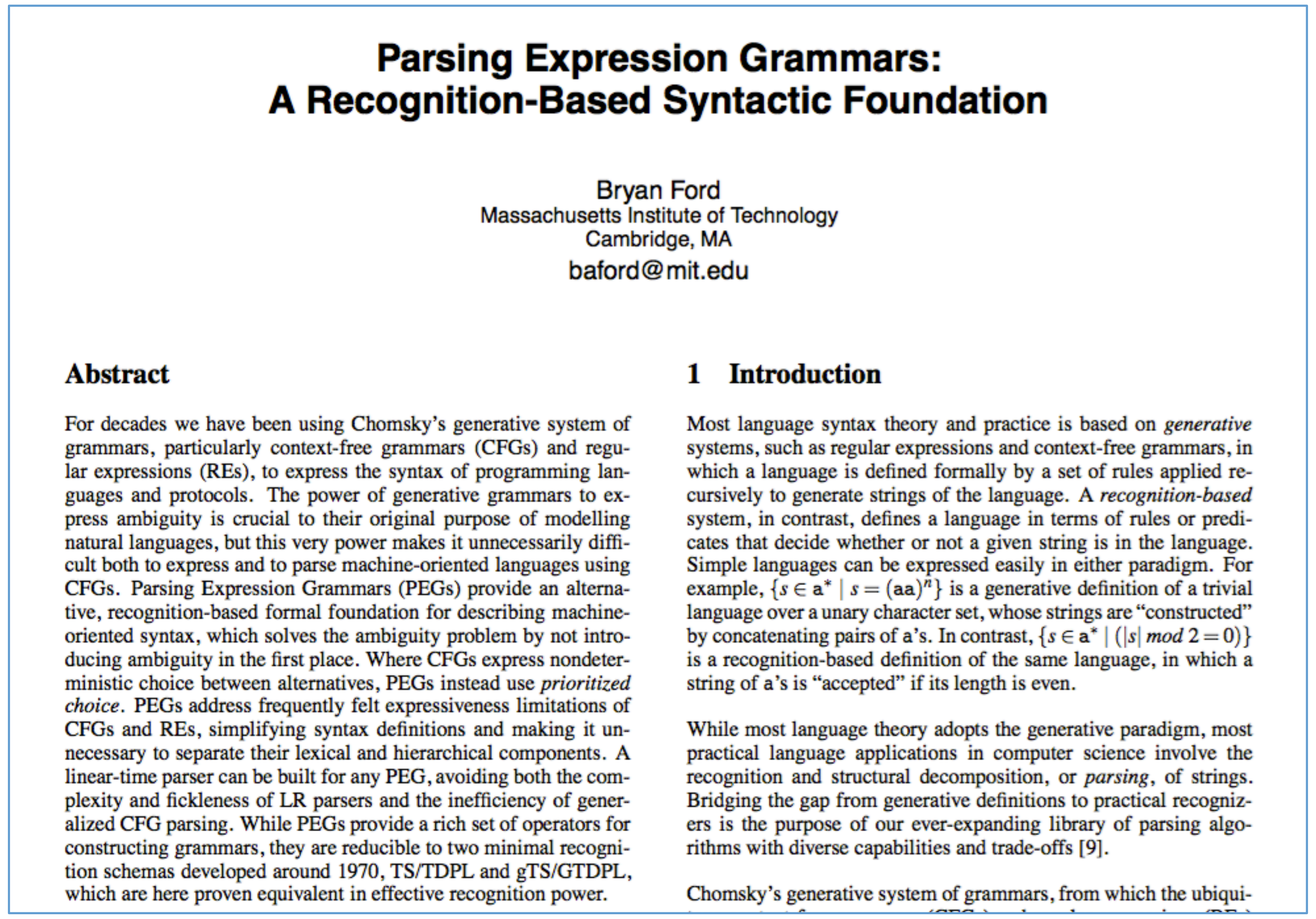
Formal basis:

- ✦ Parser combinators
- ✦ Based on Parsing Exp. Grammars
- ✦ Good performance, often linear
- ✦ Not a "packrat" implementation

Formal basis

Chomsky hierarchy





Parsing Expression Grammars: A Recognition-Based Syntactic Foundation

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

Abstract

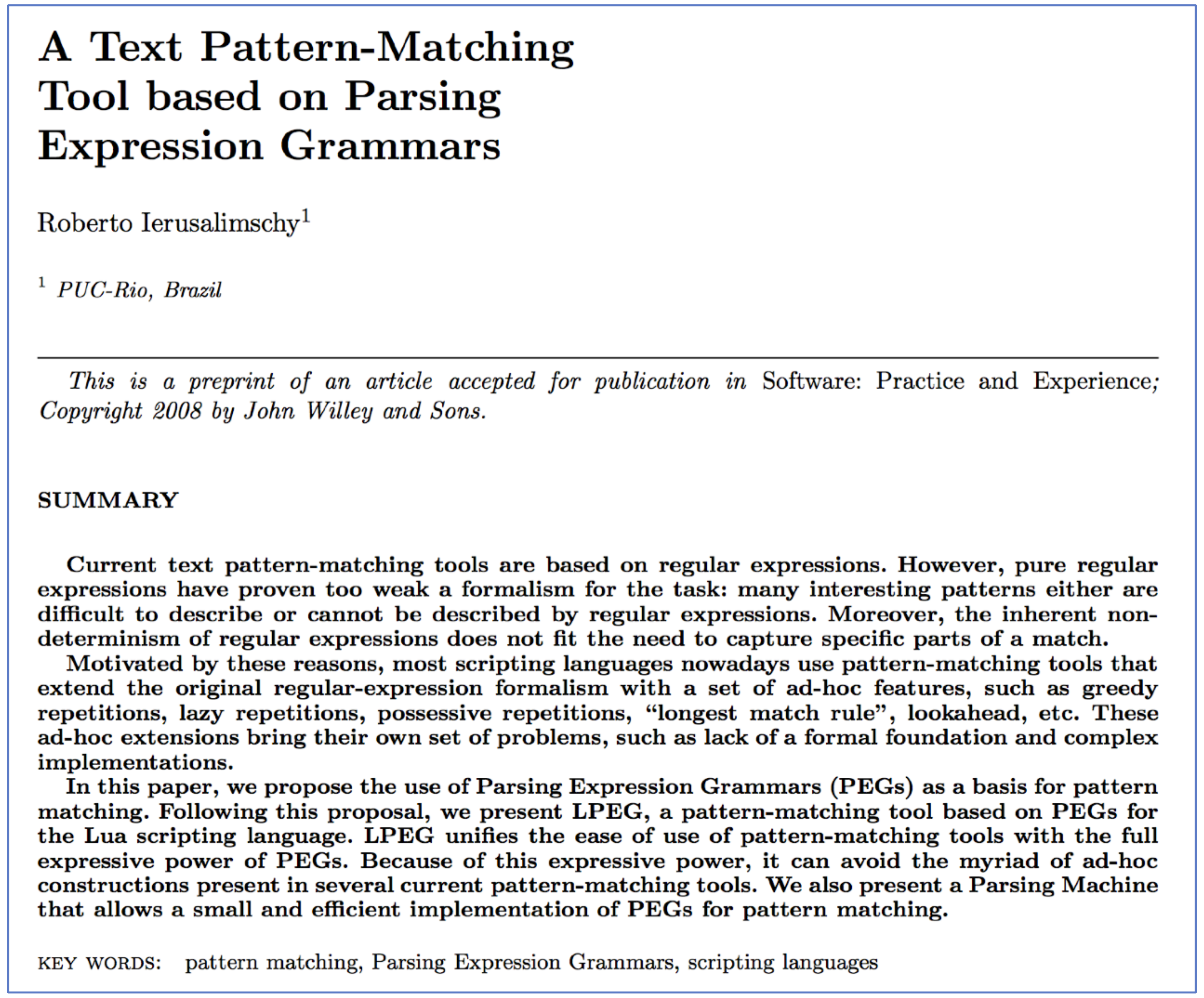
For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), to express the syntax of programming languages and protocols. The power of generative grammars to express ambiguity is crucial to their original purpose of 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.

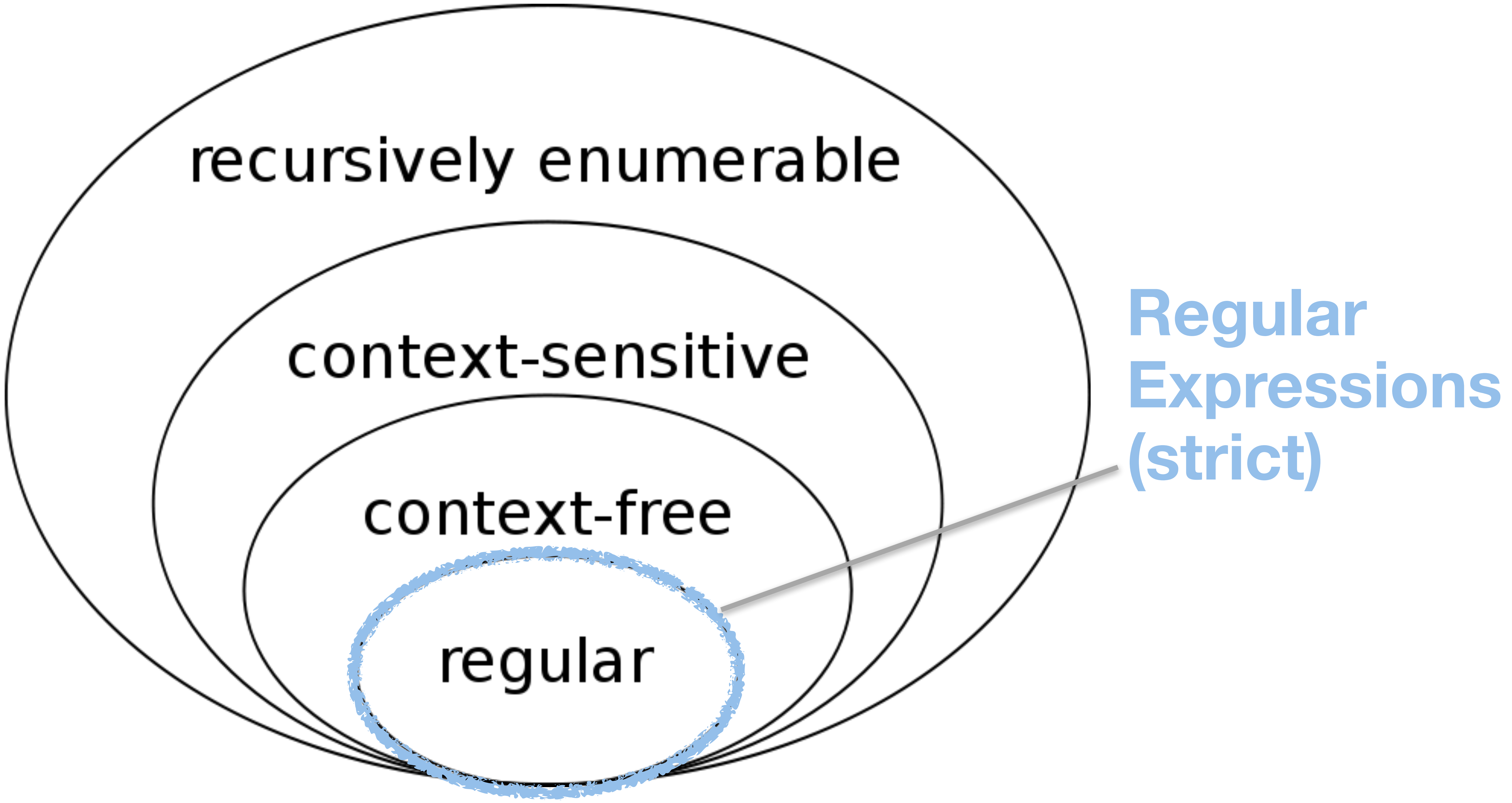
Motivated by these reasons, most scripting languages nowadays use pattern-matching tools that extend the original regular-expression formalism with a set of ad-hoc features, such as greedy repetitions, lazy repetitions, possessive repetitions, "longest match rule", lookahead, etc. These ad-hoc extensions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

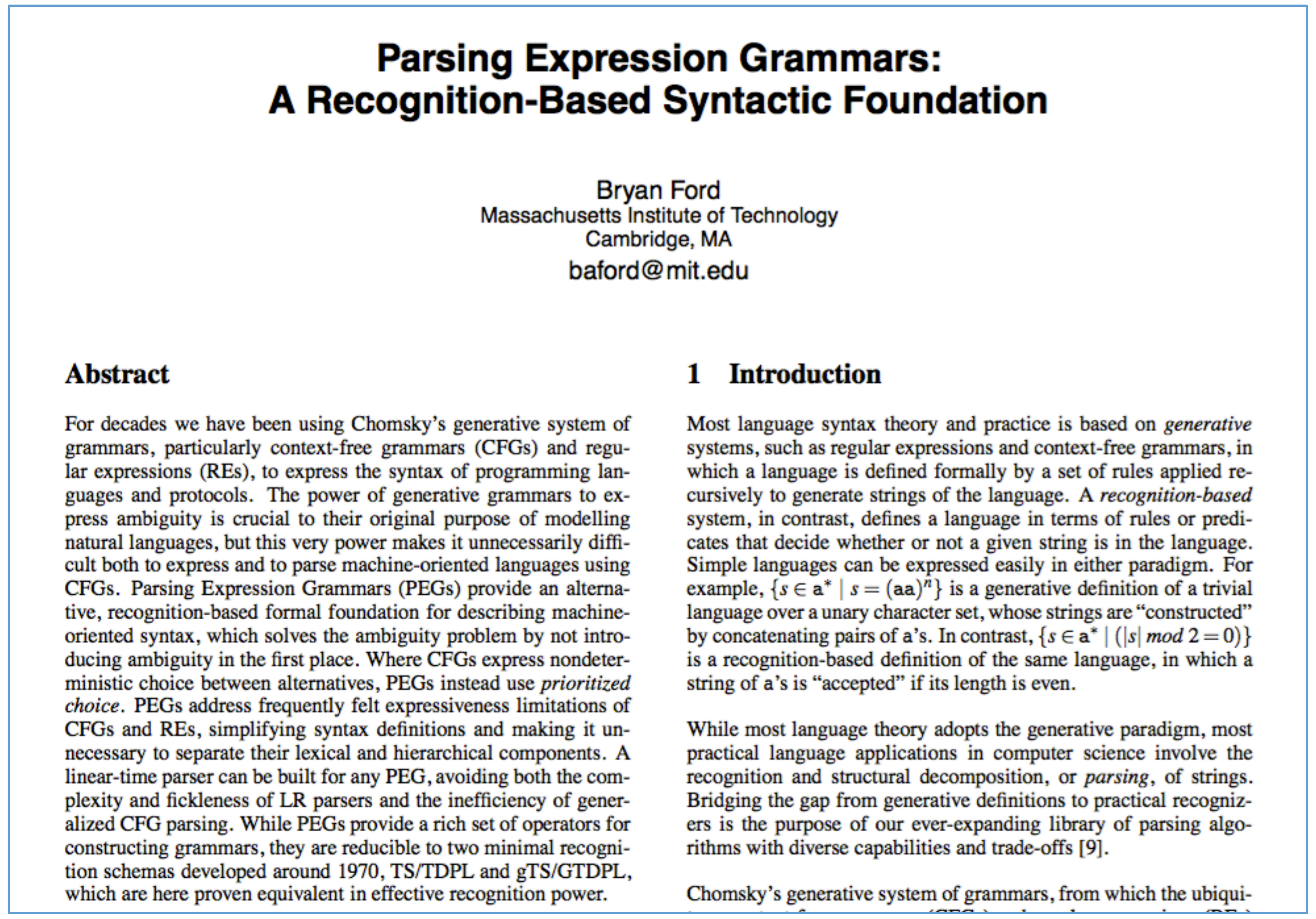
In this paper, we propose the use of Parsing Expression Grammars (PEGs) as a basis for pattern matching. Following this proposal, we present LPEG, a pattern-matching tool based on PEGs for the Lua scripting language. LPEG unifies the ease of use of pattern-matching tools with the full expressive power 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





Parsing Expression Grammars: A Recognition-Based Syntactic Foundation

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

Abstract

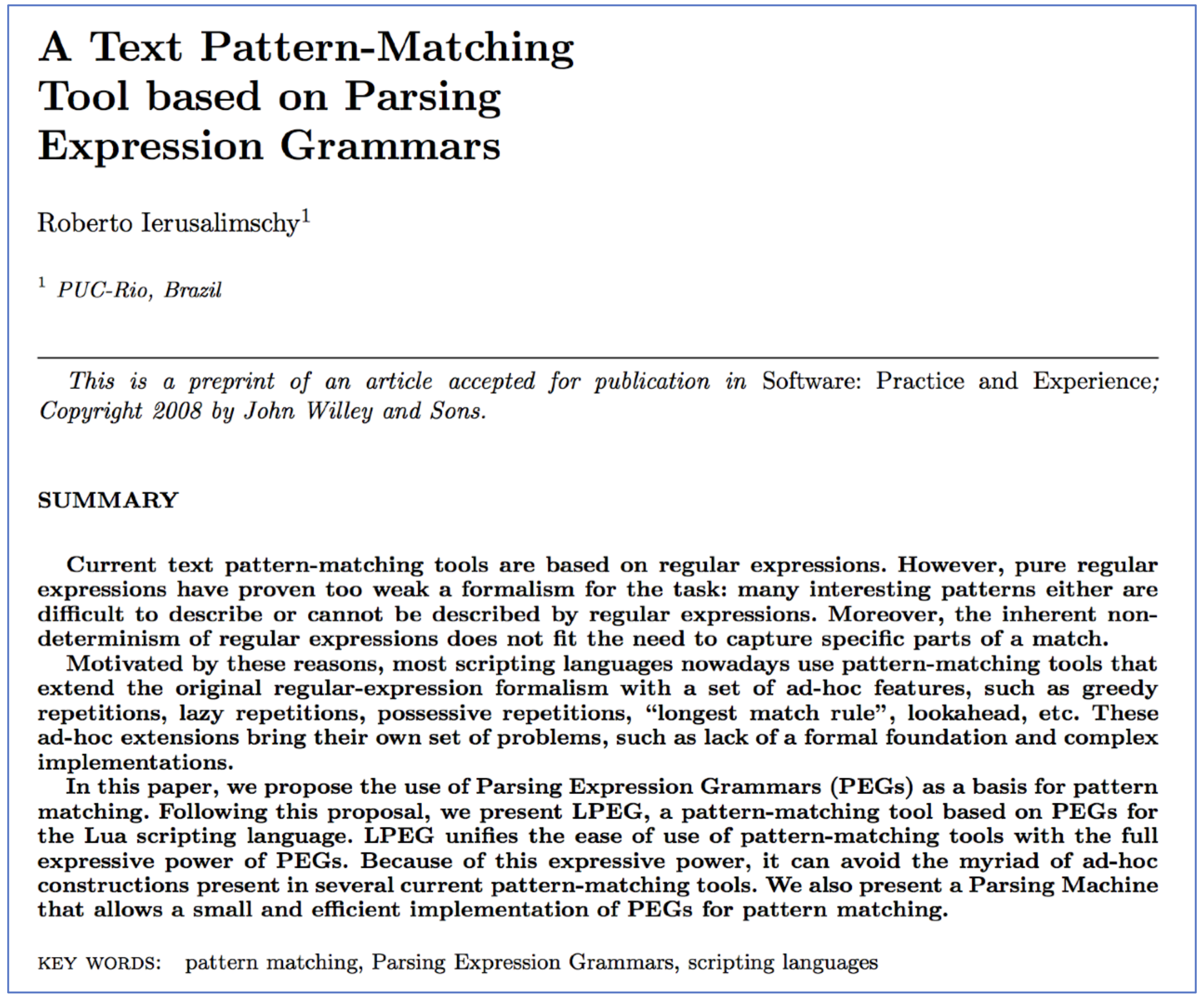
For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), to express the syntax of programming languages and protocols. The power of generative grammars to express ambiguity is crucial to their original purpose of 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, $\{s \in a^* \mid (|s| \bmod 2 = 0)\}$ 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.

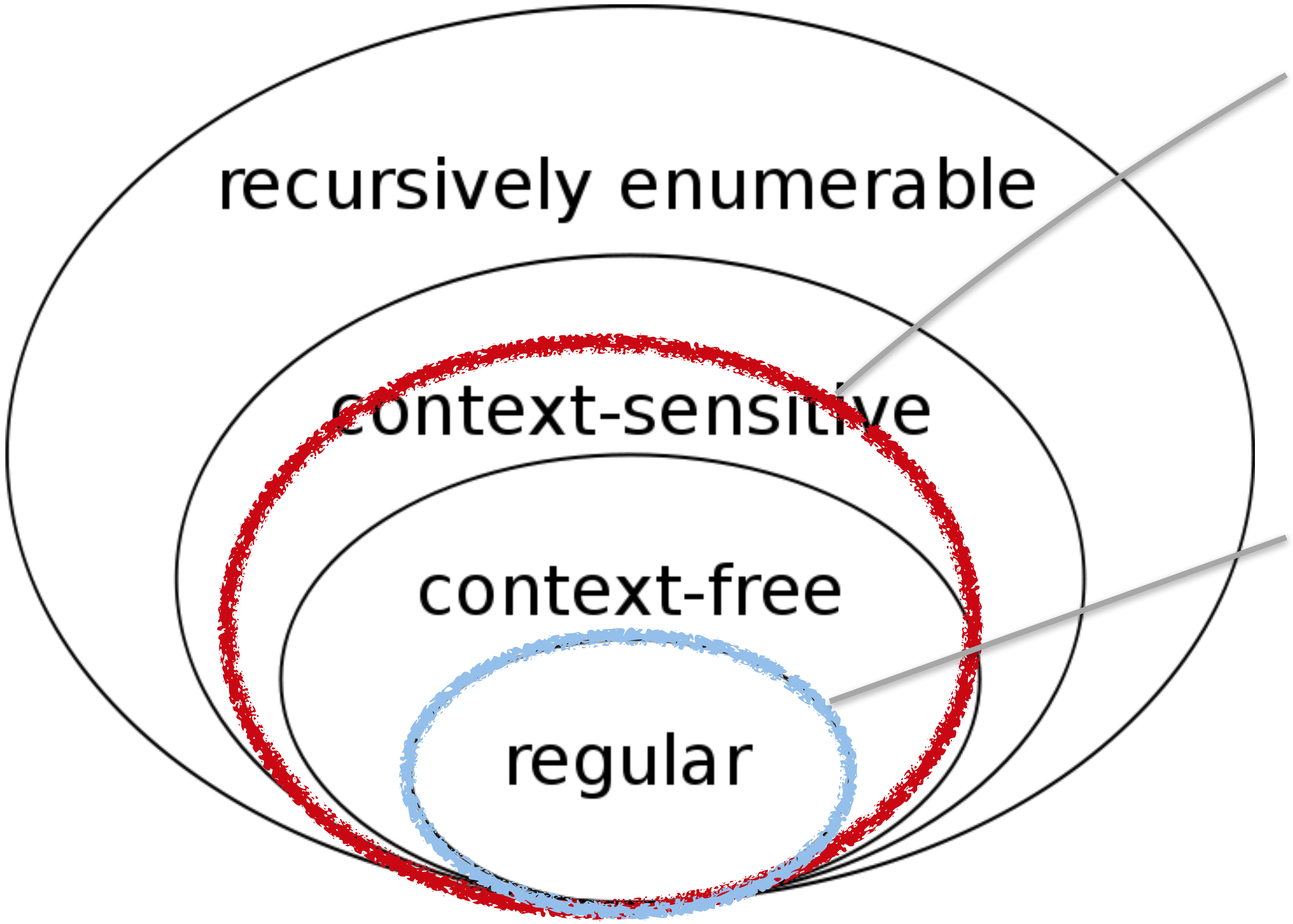
Motivated by these reasons, most scripting languages nowadays use pattern-matching tools that extend the original regular-expression formalism with a set of ad-hoc features, such as greedy repetitions, lazy repetitions, possessive repetitions, "longest match rule", lookahead, etc. These ad-hoc extensions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

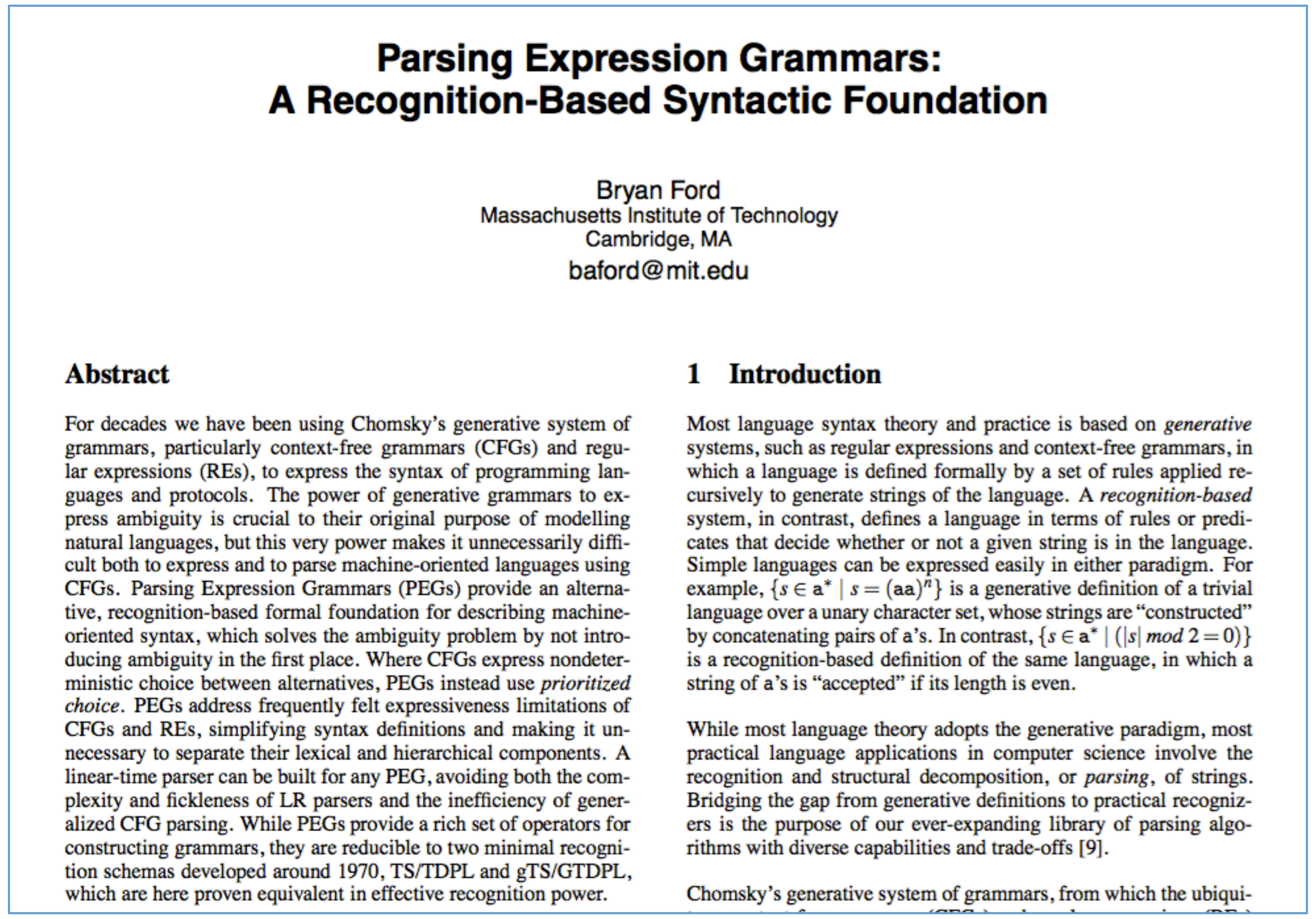
Formal basis

Chomsky hierarchy



Rosie Pattern Language
(and all PEG grammars)

Regular Expressions (strict)



Parsing Expression Grammars: A Recognition-Based Syntactic Foundation

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

Abstract

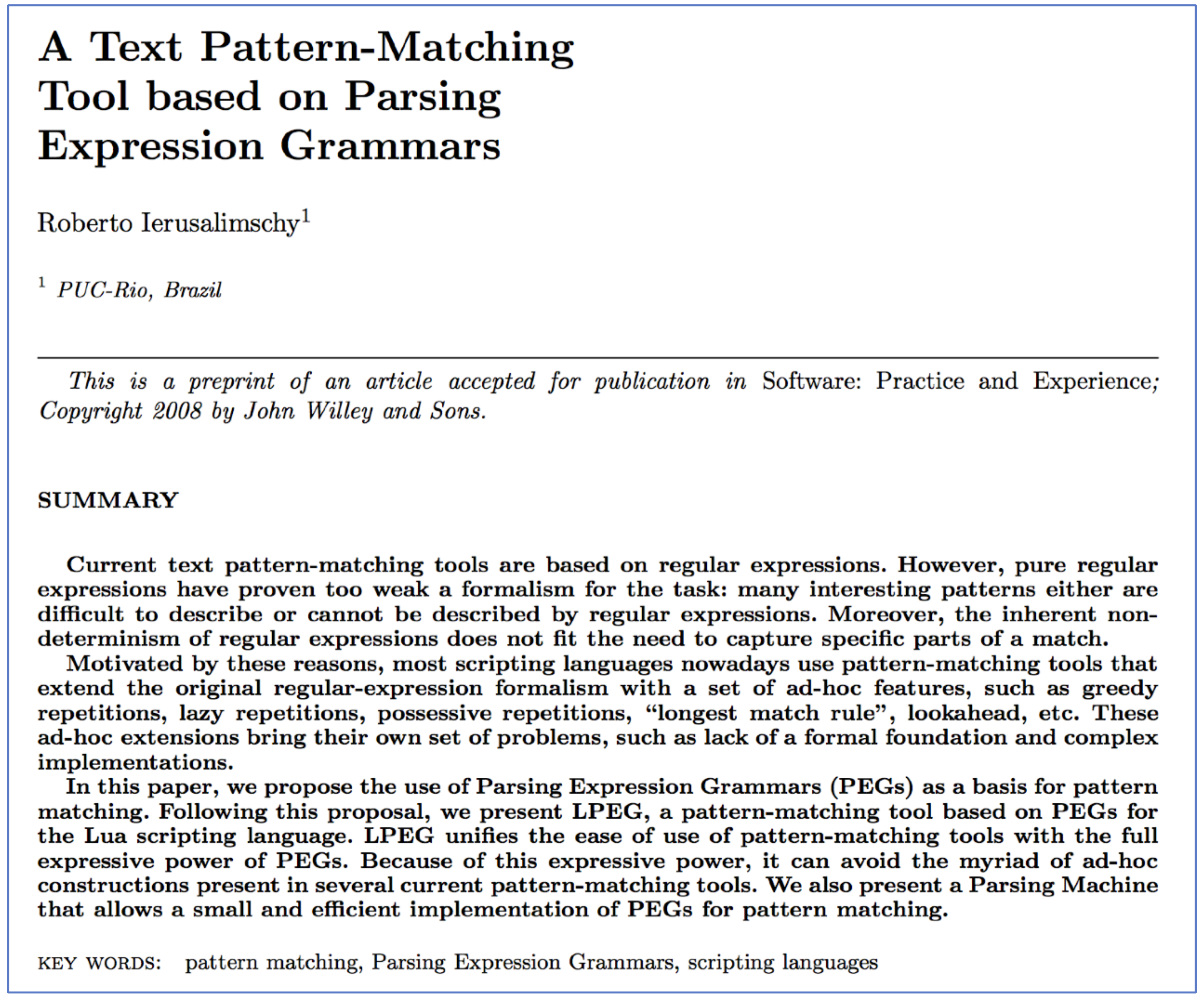
For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), to express the syntax of programming languages and protocols. The power of generative grammars to express ambiguity is crucial to their original purpose of 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.

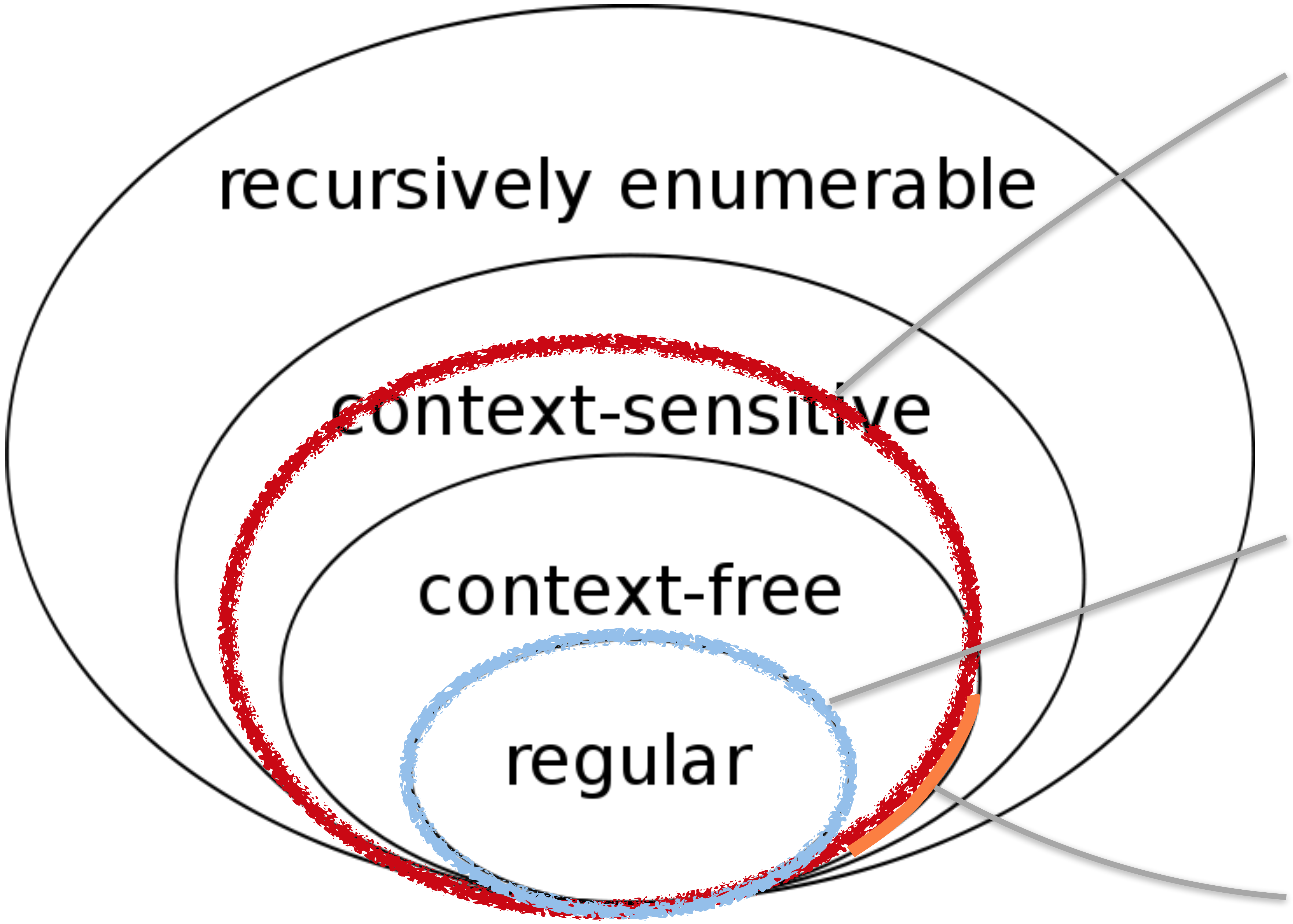
Motivated by these reasons, most scripting languages nowadays use pattern-matching tools that extend the original regular-expression formalism with a set of ad-hoc features, such as greedy repetitions, lazy repetitions, possessive repetitions, "longest match rule", lookahead, etc. These ad-hoc extensions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

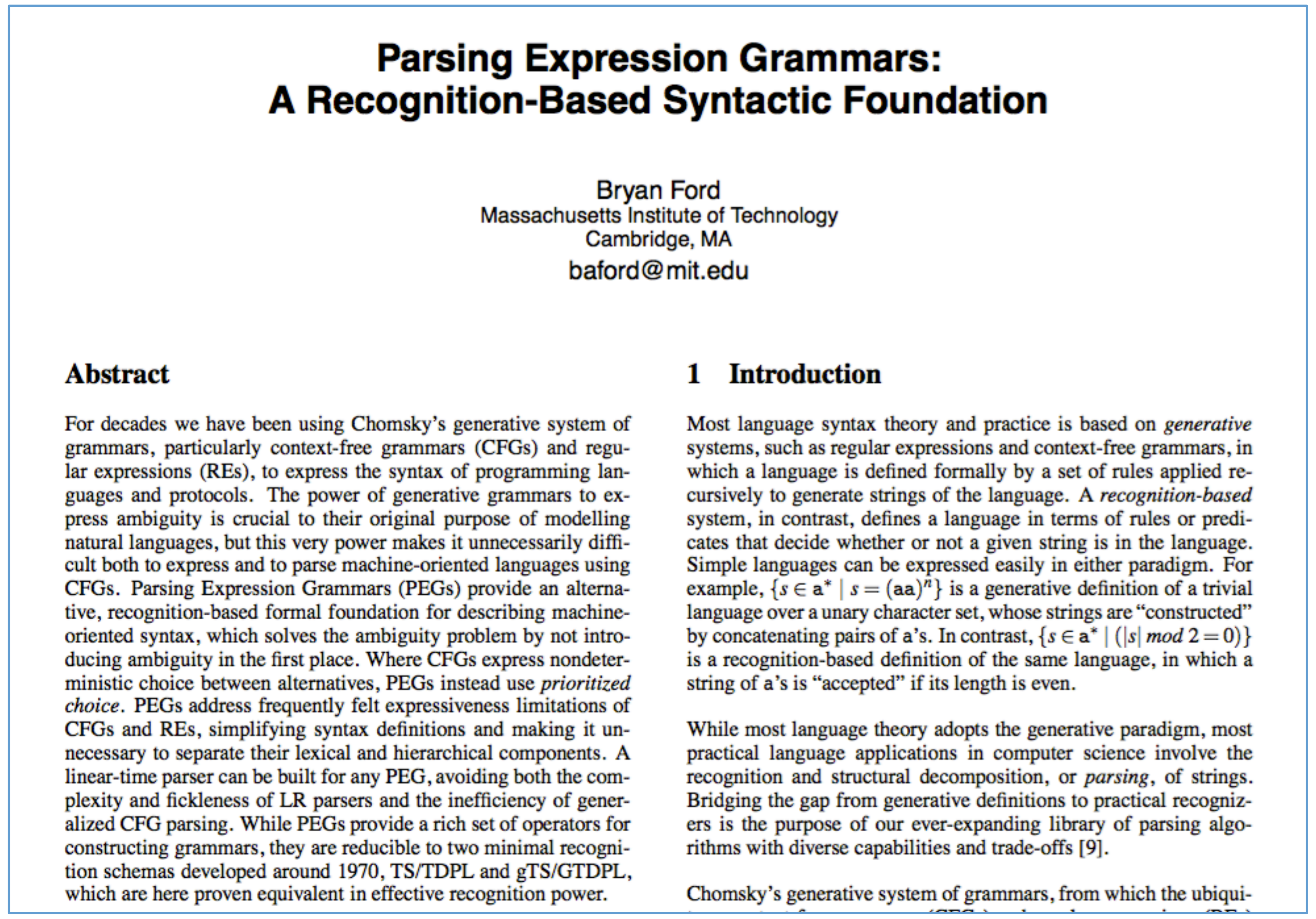


By J. Finkelstein - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=9405226

Rosie Pattern Language
(and all PEG grammars)

Regular Expressions (strict)

Open Question: PEG > CFG



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.

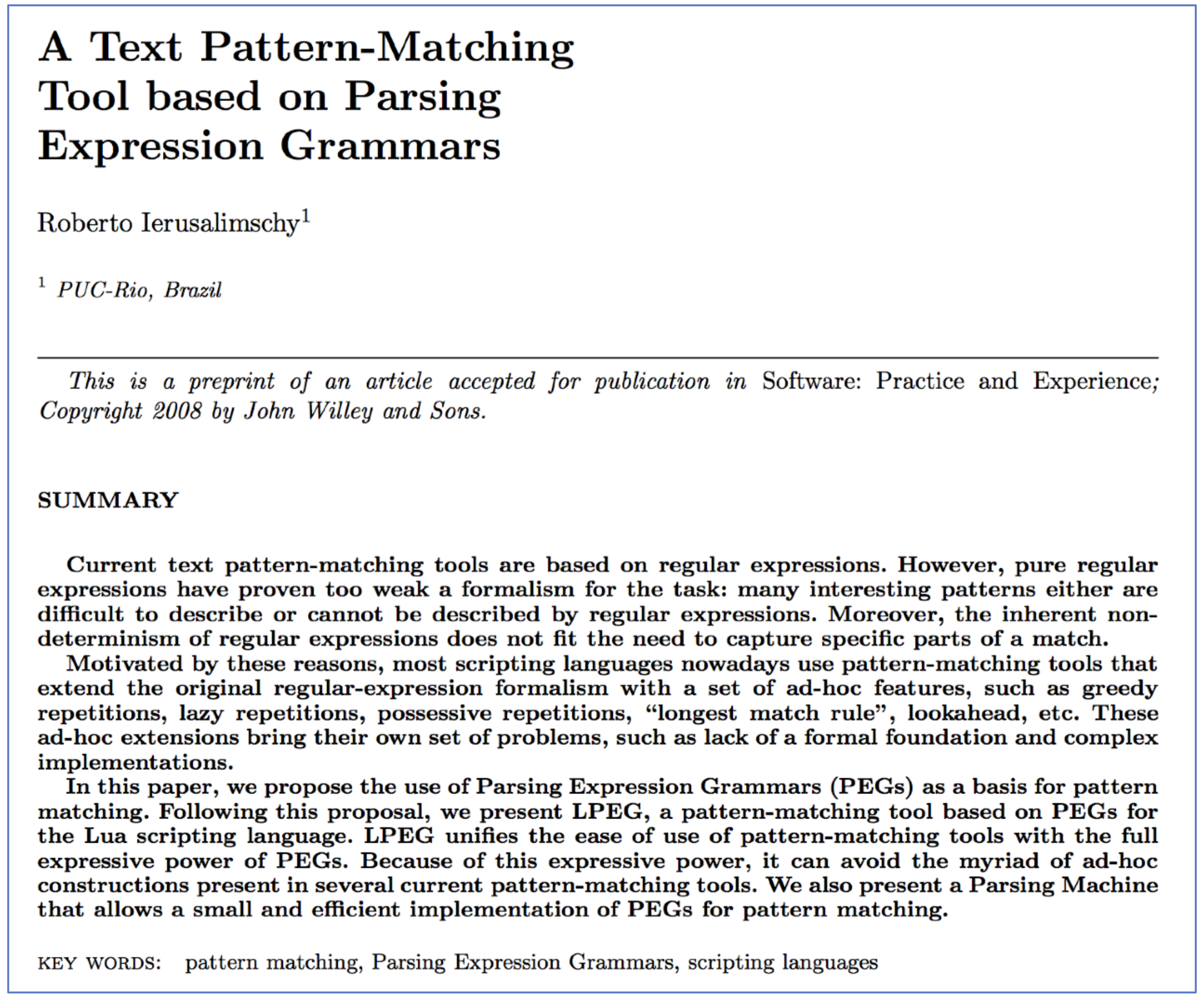
1 Introduction

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, $\{s \in \mathtt{a}^* \mid (|s| \mathit{mod}\ 2 = 0)\}$ 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.

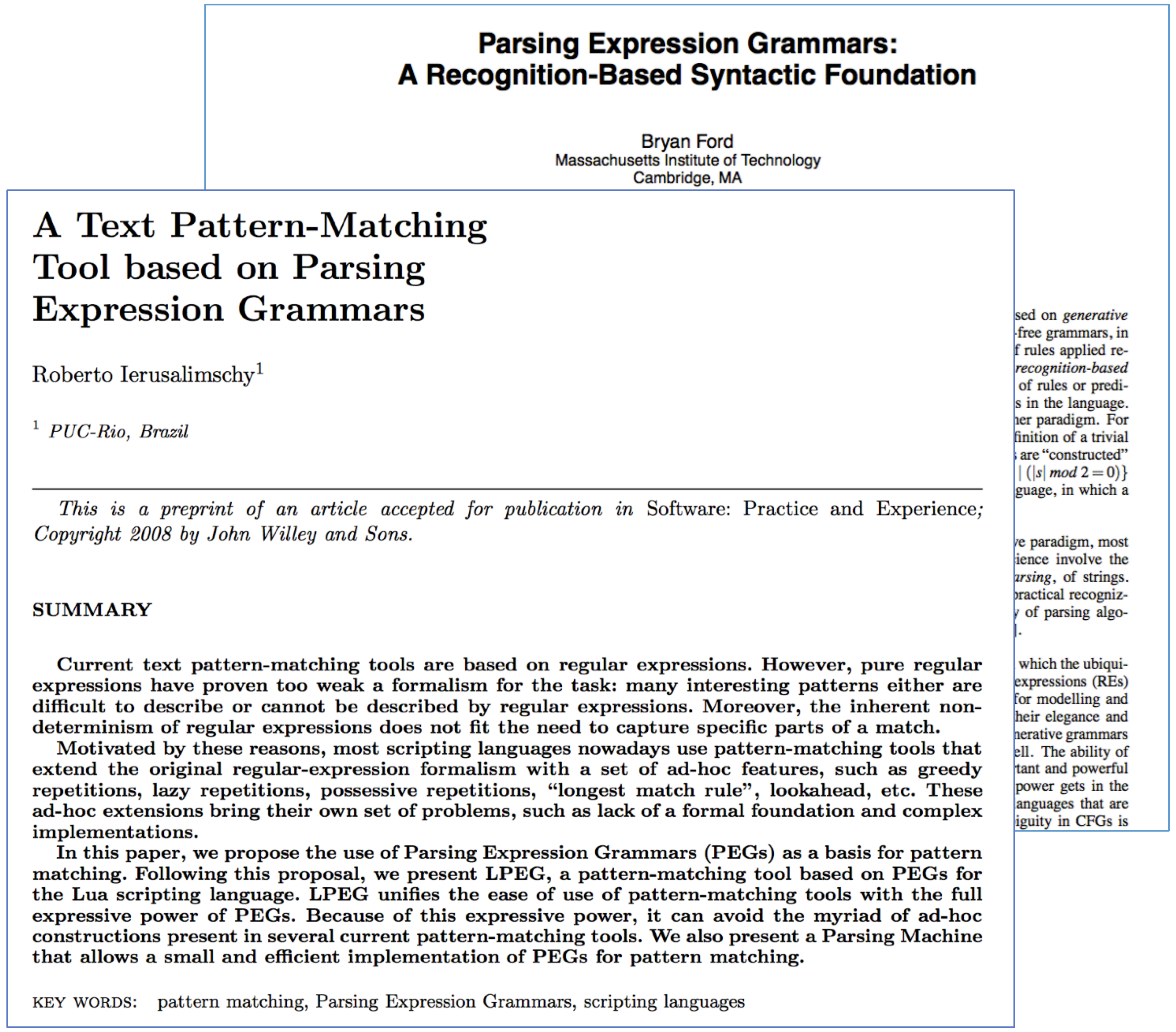
Motivated by these reasons, most scripting languages nowadays use pattern-matching tools that extend the original regular-expression formalism with a set of ad-hoc features, such as greedy repetitions, lazy repetitions, possessive repetitions, "longest match rule", lookahead, etc. These ad-hoc extensions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

The formal basis of RPL

- Rosie's operators are parser combinators
  - Based on Parsing Expression Grammars
  - Not CFG (slow!) or regex (limited!)
  - Express all deterministic (unambiguous) CFLs
  - And some non-CFLs, e.g. $a^n b^n c^n$
  - Key advantage: can match recursively structured input
- PEGs [Ford, 2004]
  - "Scanner-less parsing"
  - Linear time matching (at space cost)
  - Languages recognized by PEGs are
    - A superset of regular languages
    - All languages recognized by LL(k) and LR(k) parsers
- LPEG library [Ierusalimschy, 2008]
  - ➡ Gives a space-efficient PEG matching algorithm
  - ➡ Linear time in input size (non-grammars, no look-around)



Parsing Expression Grammars: A Recognition-Based Syntactic Foundation

Bryan Ford
Massachusetts Institute of Technology
Cambridge, MA

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

Rosie's matching engine is an enhanced version of LPEG

Rosie is self-hosting

- Rosie is a parser, and Rosie is used to parse Rosie Pattern Language
- About 115 lines of RPL (core version) to define the current RPL version
- Could support multiple versions of RPL, even different dialects
- Non-trivial user extensions to RPL can be enabled by:
  – Specifying RPL for the extension (to RPL)
  – Writing a compiler "plug-in" for the extension
  – The compiler plug-in interface has not yet been designed… *hint!*

Match non-blank, non-comment lines of RPL:

```
$ rosie match -o data '!{[:space:]*$} !{[:space:]* "--"}' rpl_1_2.rpl | wc -l
     115
```