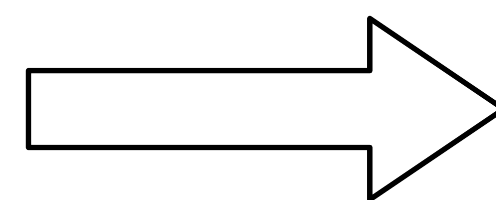
Rosie Pattern Language:

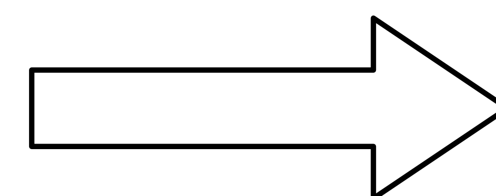
Improving on 50-Year Old Regular Expression Technology

Jamie A. Jennings, Ph.D.
Department of Computer Science
NC State University
27 September 2018

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

Meanwhile…





Raison d'être

1. My team at IBM had to write lots of regex
2. We found that regex technology does not scale
   - # patterns
   - # people
   - data size
3. So I designed Rosie Pattern Language
4. Which I’ll describe and show
5. Concluding with a roadmap, and how you can get involved

IBM Cloud DevOps Insights

DevOps Insights

Continuous Delivery
Speed with Control

Continuous Availability
Always On
with Automated Ops

Continuous Security
Protect & Defend

Open Toolchain

bluemix.net/devops

IBM Cloud DevOps Insights

DevOps Insights

Continuous Delivery
Speed with Control

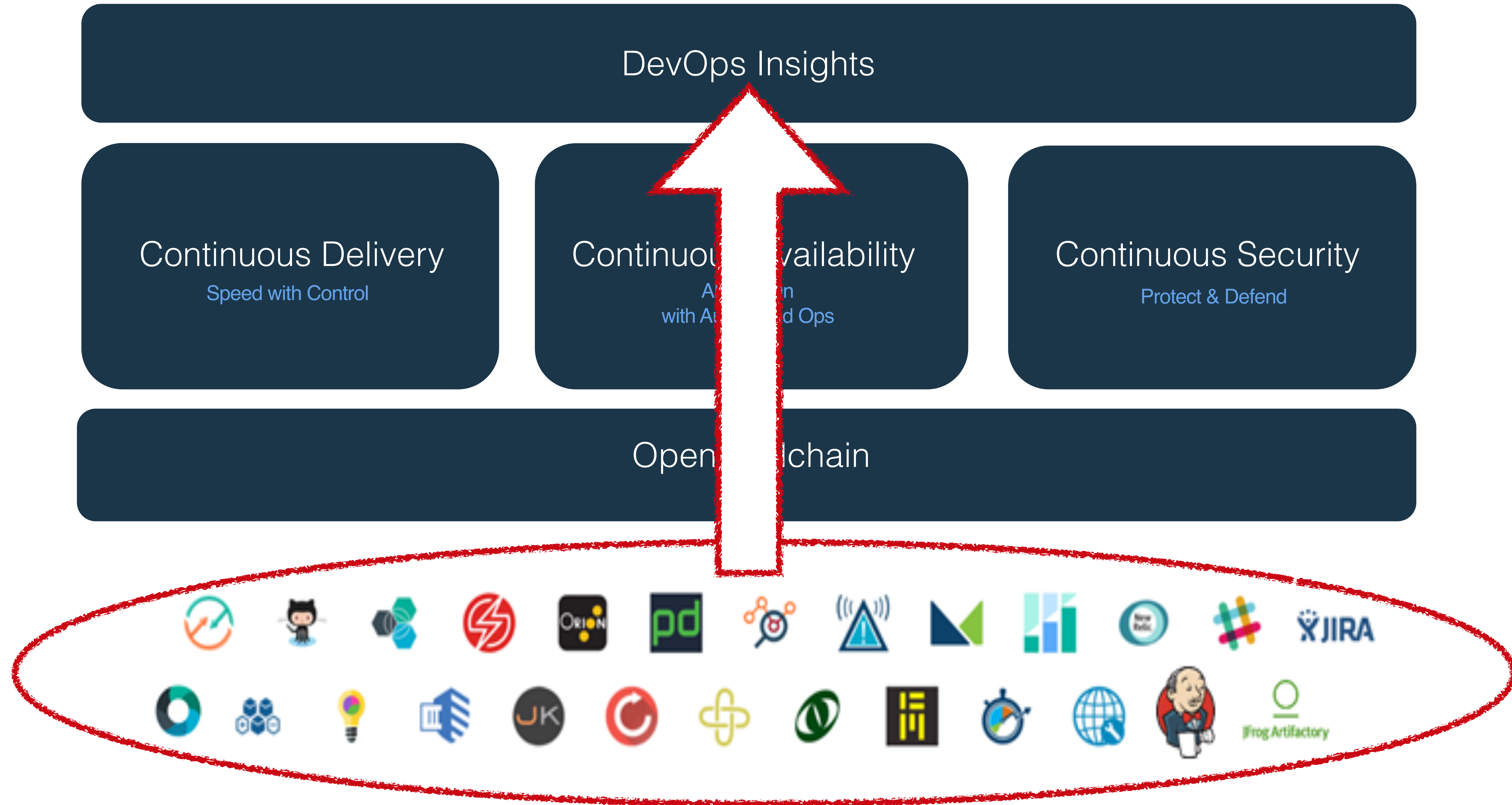
Continuous Availability
Always On
with Automated Ops

Continuous Security
Protect & Defend

Open Toolchain

bluemix.net/devops

IBM Cloud DevOps Insights



bluemix.net/devops

Planet-wide

“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

Planet-wide

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

IBM

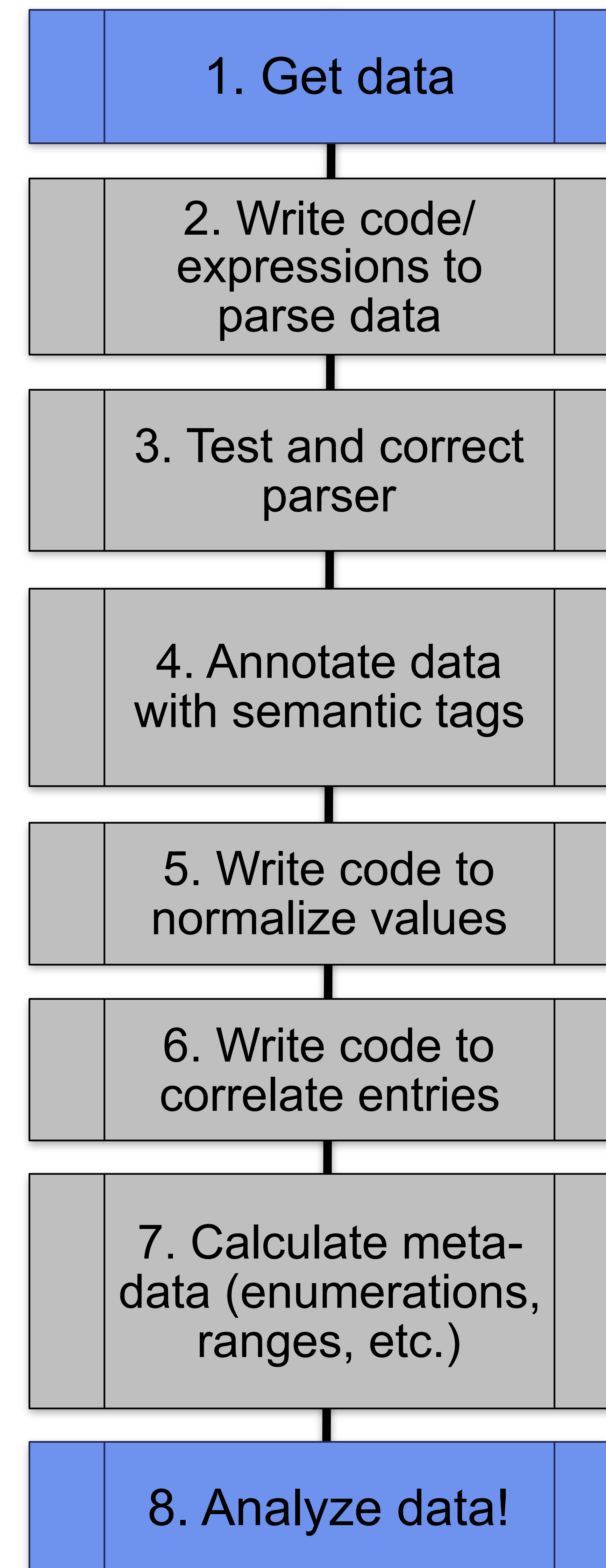
Estimates are that less than 0.5% of data is ever analyzed!

Antonio Regalado,
MIT Technology Review

Planet-wide

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

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



Planet-wide

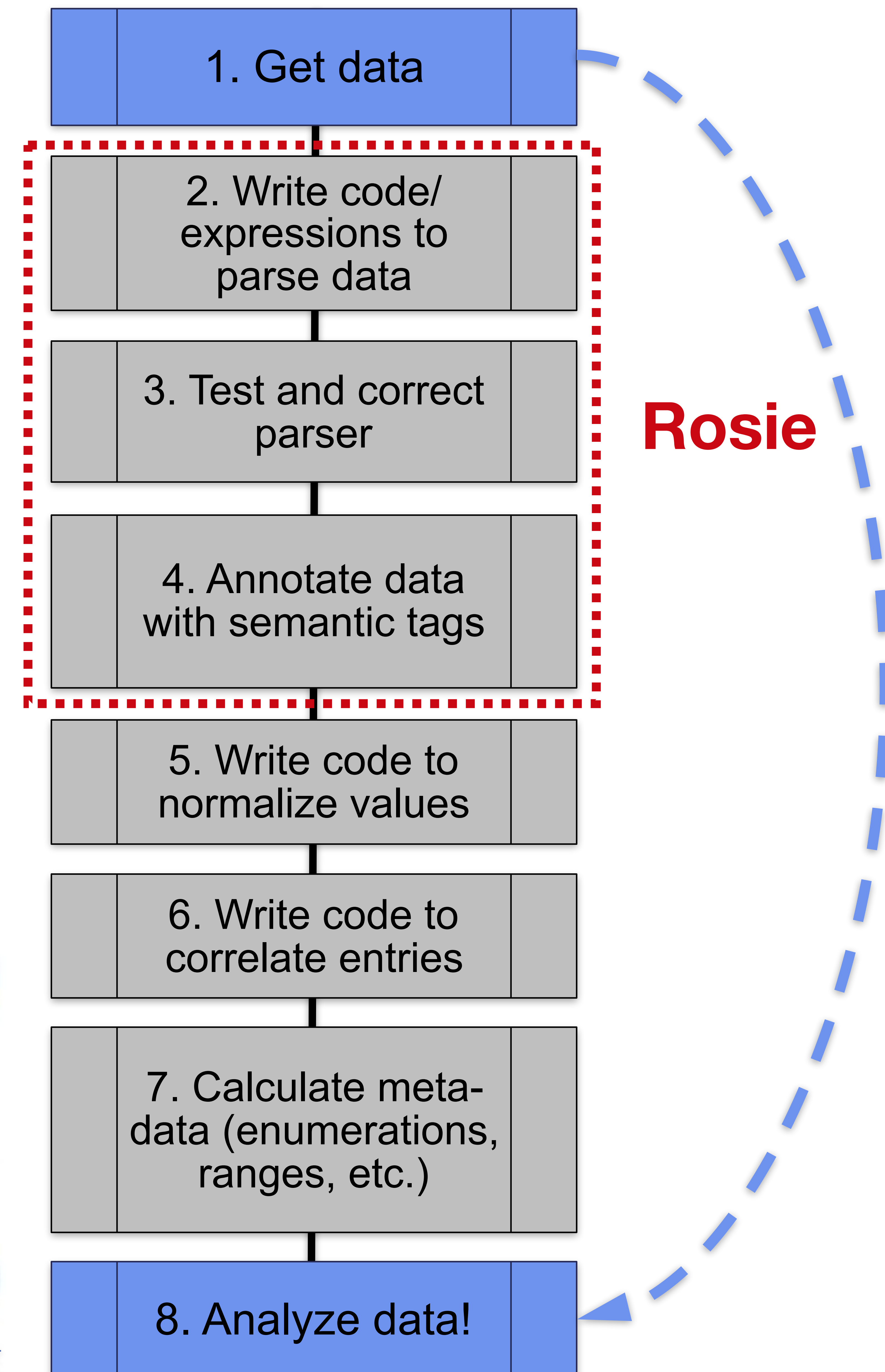
“Every day, we create 2.5 quintillion bytes of data”

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

Data AVAILABLE to an organization

Missed opportunity

Data an organization can PROCESS



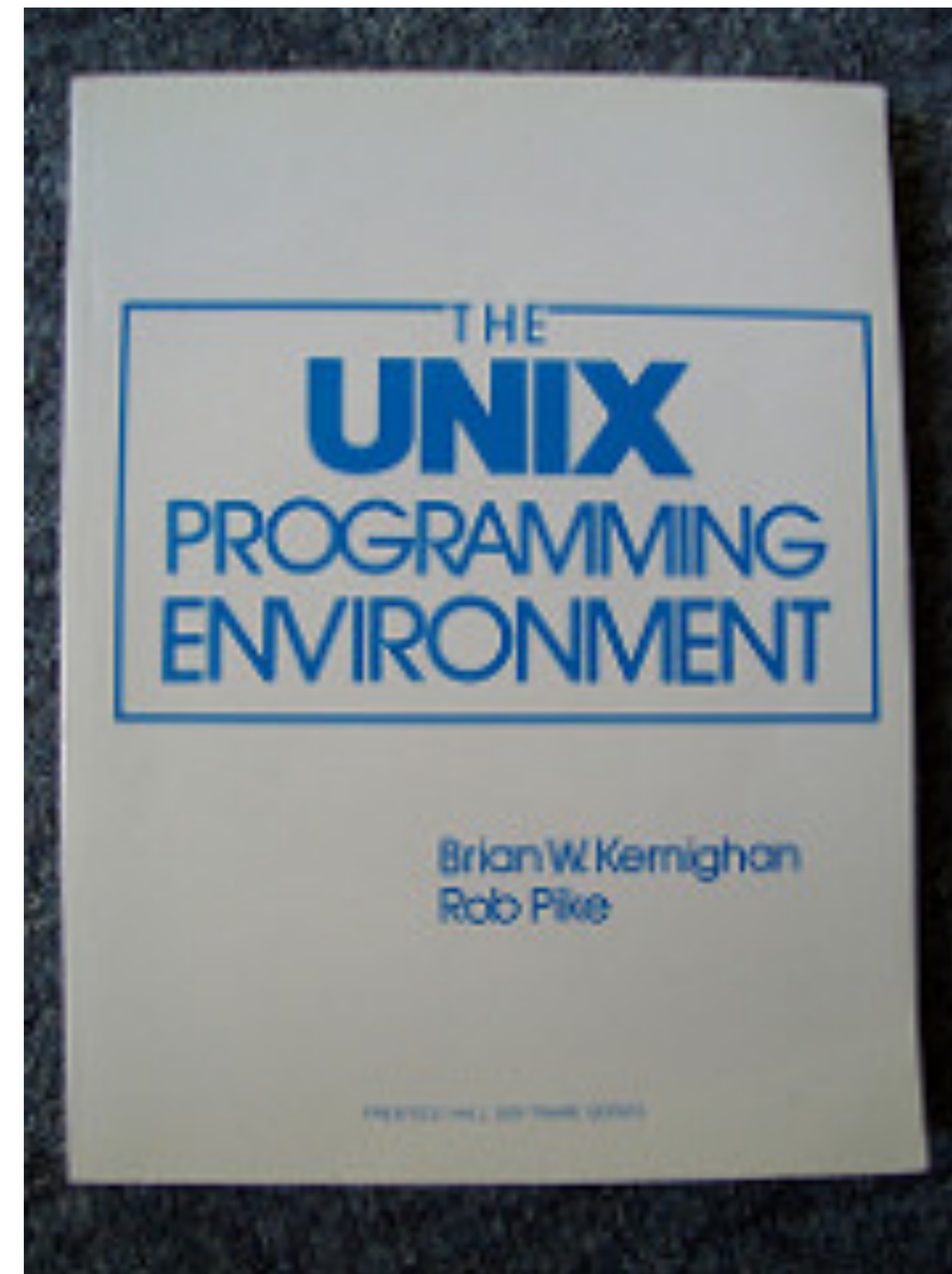
Current approaches

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

Abraham Maslow

Regular expressions as tools:

70s

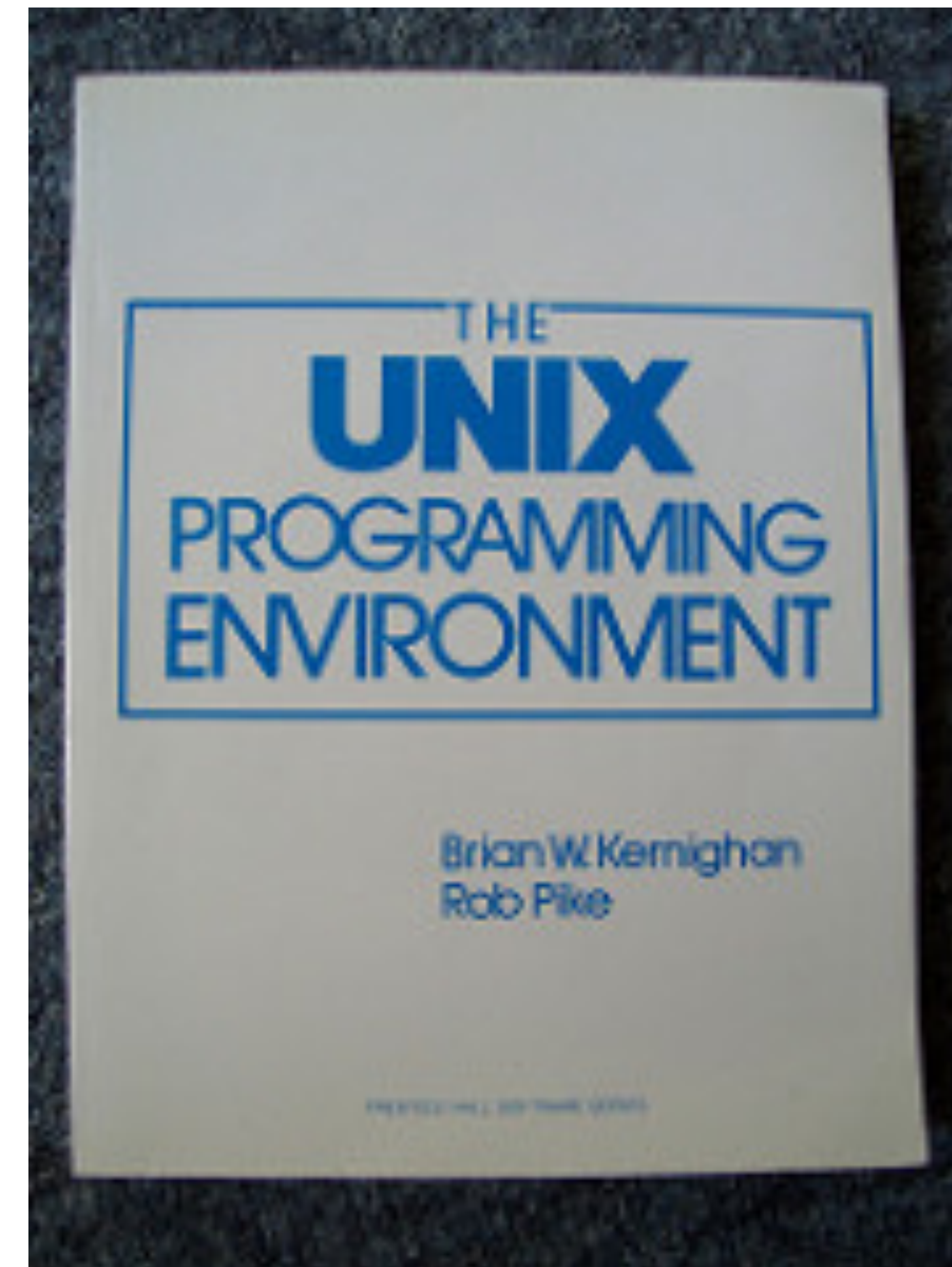


On the command line:

```
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:

70s



2017

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'
```

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
```

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`

namespace!

Regex issue #1: 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

Regex issue #1: Notoriously hard to read & maintain

- Dense, **cryptic** syntax
- Semant
- Flags th
- Regex

RPL syntax looks like a programming language.

→ Patterns can be named
→ Whitespace, comments, simplified operators

RPL expressions compose.

→ Enables encapsulation and **packages** of patterns

"…Now they have two problems."

Jamie Zawinski
http://regex.info/blog/2006-09-15/247

Regex issue #2: Performance is highly 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 issue #2: Performance is highly variable

Matching this $re against a 94-character input takes around 65 seconds in Perl*

```
$re = “^(.*?,){29}Gold”;
$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”;
```

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

Regex issue #2: Performance is highly variable

Mat ... seconds
in P

In RPL, expressions are greedy and possessive.

→ Backtracking is explicit
→ To get exponential backtracking, you write it that way

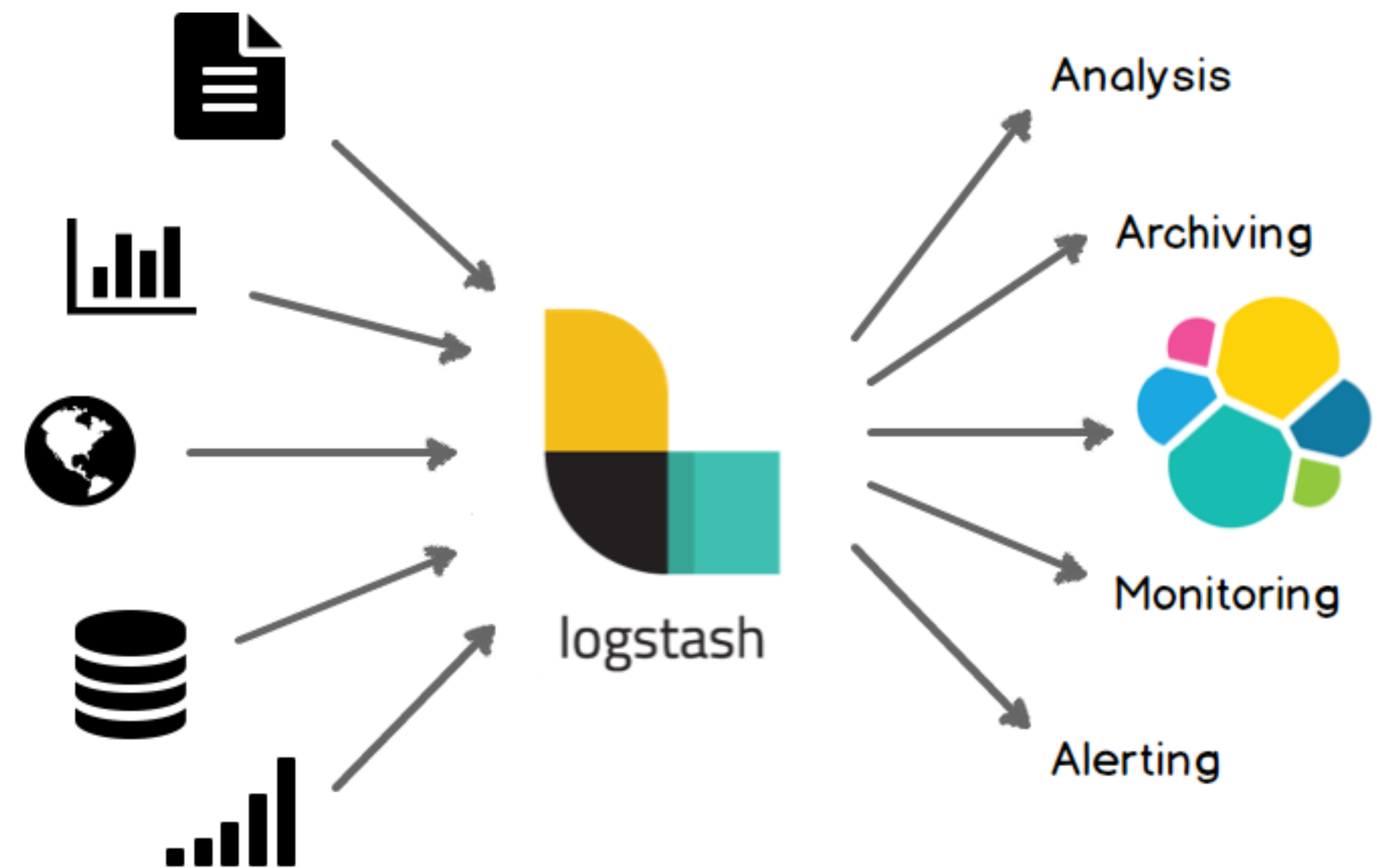
RPL makes it difficult to be accidentally inefficient.

$r
$i ... 24,25,26,
Br

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

Issue #3: Regex collections? Grok does this. Others?

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.



Issue #3: Regex collections? Grok does this. Others?

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.

Logstash ships with about 120 patterns by default. You can find them here: https://github.com/logstash-plugins/logstash-patterns-core/tree/master/patterns. You can add your own trivially. (See the `patterns_dir` setting)

Issue #3: Regex collections? Grok does this. Others?

> 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.

> Logstash ships with about 120 patterns by default. You can find them here: https://github.com/logstash-plugins/logstash-patterns-core/tree/master/patterns. You can add your own trivially. (See the `patterns_dir` setting)

Caveats

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

Still cryptic, 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$
```

Still cryptic, and they don't play well with dev tools

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

RPL is designed like a programming language.

→ It reads like code
→ It diffs like code
→ It debugs like code
→ It tests like **modular** code

Rosie Pattern Language

“All progress depends on the unreasonable [woman]”

George Bernard Shaw, paraphrased

RPL

```
----
---- json.rpl    rpl patterns for processing json input
----
---- © Copyright IBM Corporation 2016, 2017, 2018.
---- 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
   member = key ":" value
   object = "{" ( member ("," member)* )? "}"
   array = "[" ( value ("," value)* )? "]"
in
   value = ~ string / number / object / array / true / false / null
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 "123e65", "0e+1", "0e1", "20e1", "1E22", "1E-2", "1E+2", "123e45", "1e-2", "1e+2"
-- 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

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

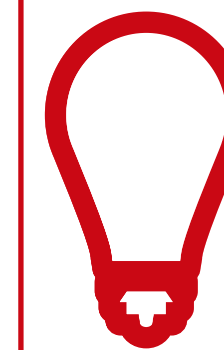
```
----
---- json.rpl    rpl patterns for processing json input
----
---- © Copyright IBM Corporation 2016, 2017, 2018.
---- LICENSE: MIT License (https://opensource.org/licenses/mit-license.html)
---- AUTHOR: Jamie A. Jennings

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
   member = key ":" value
   object = "{" ( member ("," member)* )? "}"
   array = "[" ( value ("," value)* )? "]"
in
   value = ~ string / number / object / array / true / false / null
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 "123e65", "0e+1", "0e1", "20e1", "1E22", "1E-2", "1E+2", "123e45", "1e-2", "1e+2"
-- 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
```

NAMED PATTERNS

Can your ‘grep’ do this?

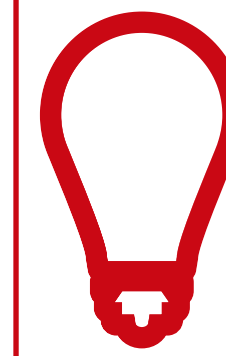


NAMED PATTERNS

```
$ curl -s www.google.com
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en"><head><meta content="Search the world's i
nformation, including webpages, images, videos and more. Google has many special features to help you find exactly what yo
u're looking for." name="description"><meta content="noodp" name="robots"><meta content="text/html; charset=UTF-8" http-eq
uiv="Content-Type"><meta content="/images/branding/googleg/1x/googleg_standard_color_128dp.png" itemprop="image"><title>Go
ogle</title><script nonce="JfwlaIwglq/p59AusKSAHQ==">(function(){window.google={kEI:'8g-sW7rQJezTjwTMp7qYCw',kEXPI:'0,1353
747,57,1654,304,583,433,281,838,287,1071,154,731,141,193,55,802,209,97,42,258,31,168,356,2337766,231,32,329294,1294,12383,
4855,32692,15247,867,41,275,10445,1402,6381,3335,2,2,4604,2197,367,1214,326,1776,2314,3191,224,2218,260,5107,575,1119,2,57
8,728,606,1826,58,2,1,3,1297,1712,2158,453,2096,658,636,8,302,1267,222,552,1231,884,133,283,2,841,283,3337,525,22,599,5,2,
2,743,574,426,748,3,774,1472,283,556,1266,464,1450,69,1050,334,10,120,328,782,234,386,8,1003,81,7,1,2,26,462,93,527,29,983
,6,406,444,7,7,62,569,1216,99,429,241,536,412,499,119,668,393,1068,45,79,374,1085,243,2,8,304,318,59,88,411,412,2,198,355,
454,54,1142,144,280,76,16,21,1,54,18,40,63,2,288,255,108,263,4,135,130,3,460,2,35,202,58,43,73,12,28,1,1005,6,32,385,67,15
9,92,556,135,38,61,180,332,287,218,116,38,45,58,24,219,466,15,377,159,28,68,183,68,56,94,2,332,680,276,331,384,127,672,599
2947,2554,5997691,20,2800075,4,1572,549,332,445,1,2,1,1,78,1,512,388,583,9,304,1,8,1,2,1,1,2130,1,1,1,1,1,414,1,263,49,39,
22,5,1,5,5,6,121,67,2,2,4,2,38,6,1,33,8,22308707',authuser:0,kscs:'c9c918f0_8g-sW7rQJezTjwTMp7qYCw',kGL:'US'};google.kHL='
en';})();google.time=function(){return(new Date).getTime()};(function(){google.lc=[];google.li=0;google.getEI=function(a){
for(var b;a&&(!a.getAttribute||!(b=a.getAttribute("eid")));)a=a.parentNode;return b||google.kEI};google.getLEI=function(a)
{for(var b=null;a&&(!a.getAttribute||!(b=a.getAttribute("leid")));)a=a.parentNode;return b};google.https=function(){return
"https:"==window.location.protocol};google.ml=function(){return null};google.log=function(a,b,e,c,g){if(a=google.logUrl(a,
b,e,c,g)){b=new Image;var d=google.lc,f=google.li;d[f]=b;b.onerror=b.onload=b.onabort=function(){delete d[f]};google.vel&&
google.vel.lu&&google.vel.lu(a);b.src=a;google.li=f+1}};google.logUrl=function(a,b,e,c,g){var d="",f=google.ls||"";e||-1!=
b.search("&ei=")||(d="&ei="+google.getEI(c),-1==b.search("&lei=")&&(c=google.getLEI(c))&&(d+="&lei="+c));c="";!e&&google.c
shid&&-1==b.search("&cshid=")&&"slh"!=a&&(c="&cshid="+google.cshid);a=e||"/"+(g||"gen_204")+"?atyp=i&ct="+a+"&cad="+b+d+f+
"&zx="+google.time()+c;/^http:/i.test(a)&&google.https()&&(google.ml(Error("a"),!1,{src:a,glmm:1}),a="");return a};}).call
(this);(function(){google.y={};google.x=function(a,b){if(a)var c=a.id;else{do c=Math.random();while(google.y[c])}google.y[
c]=[a,b];return!1};google.lm=[];google.plm=function(a){google.lm.push.apply(google.lm,a)};google.lq=[];google.load=functio
n(a,b,c){google.lq.push([[a],b,c])};google.loadAll=function(a,b){google.lq.push([a,b])};}).call(this);google.f={};</script
><script nonce="JfwlaIwglq/p59AusKSAHQ==">var a=window.location,b=a.href.indexOf("#");if(0<=b){var c=a.href.substring(b+1)
;/(^|&)q=/.test(c)&&-1==c.indexOf("#")&&a.replace("/search?"+c.replace(/(^|&)fp=[^&]*/g,"")+"&cad=h")};</script><style>#gb
ar,#guser{font-size:13px;padding-top:1px !important;}#gbar{height:22px}#guser{padding-bottom:7px !important;text-align:rig
```

Can your ‘grep’ do this?

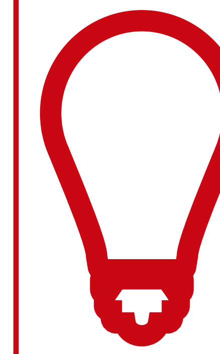
```
$ curl -s www.google.com
```



Named patterns

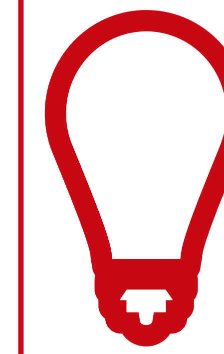
Can your 'grep' do this?

```
$ curl -s www.google.com | rosie grep -o subs net.url_common
```



Named patterns

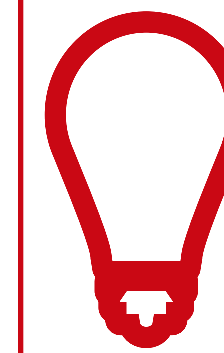
Can your ‘grep’ do this?



Named patterns

```
$ curl -s www.google.com | rosie grep -o subs net.url_common
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
$
```

Can your 'grep' do this?



NAMED PATTERNS

```
$ curl -s www.google.com | rosie grep -o subs net.url_common
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

pattern net.url_common
==> package net, pattern url_common

Can your 'grep' do this?

CUSTOMIZABLE SYNTAX 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?

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

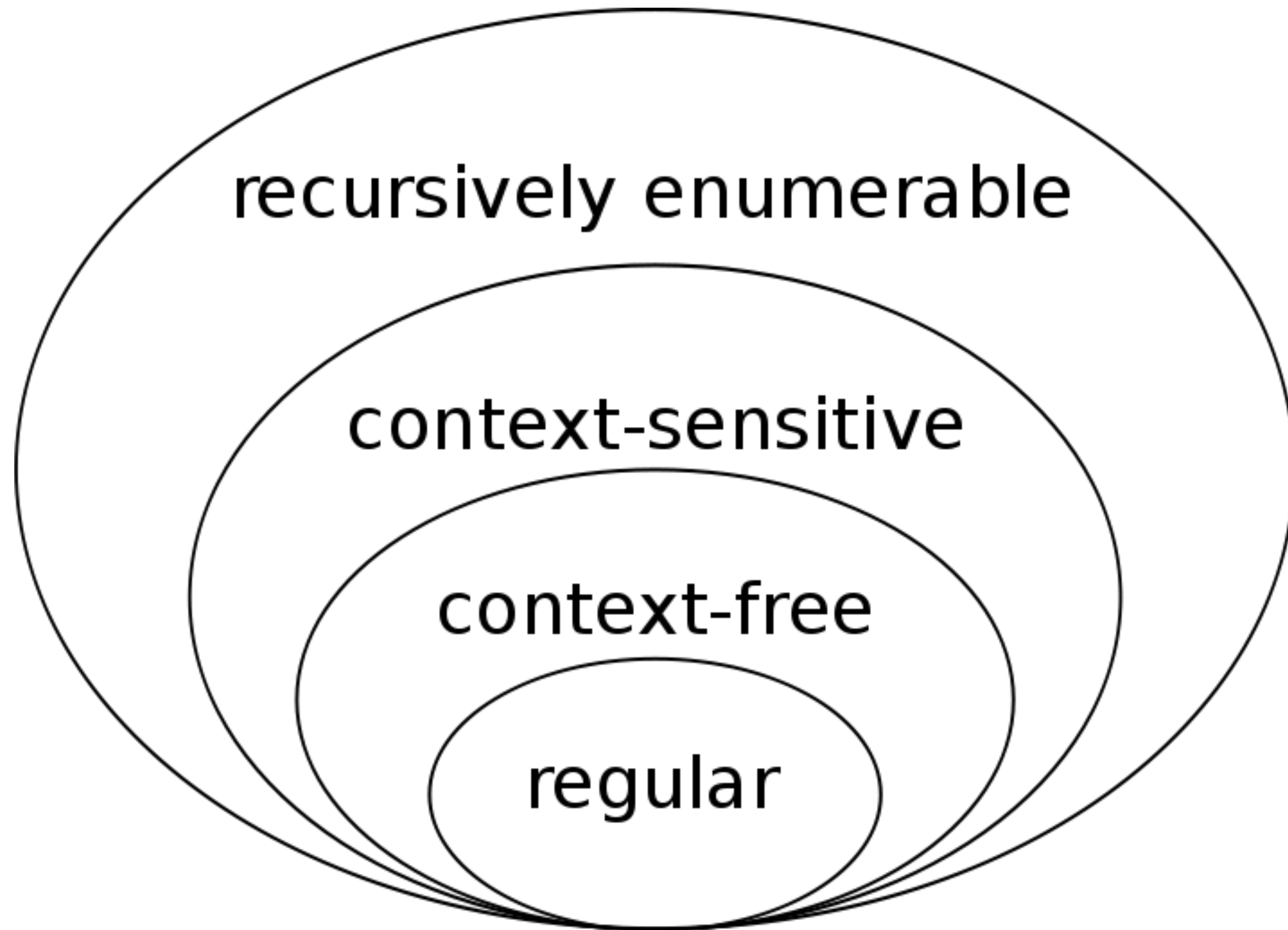
Matching line

num.denoted_hex

num.hex (a sub-match)

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

recursively enumerable

context-sensitive

context-free

regular

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

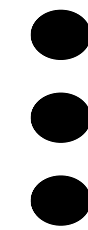
For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), to express the syntax of programming languages and protocols. The power of generative grammars to express ambiguity is crucial to their original purpose of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

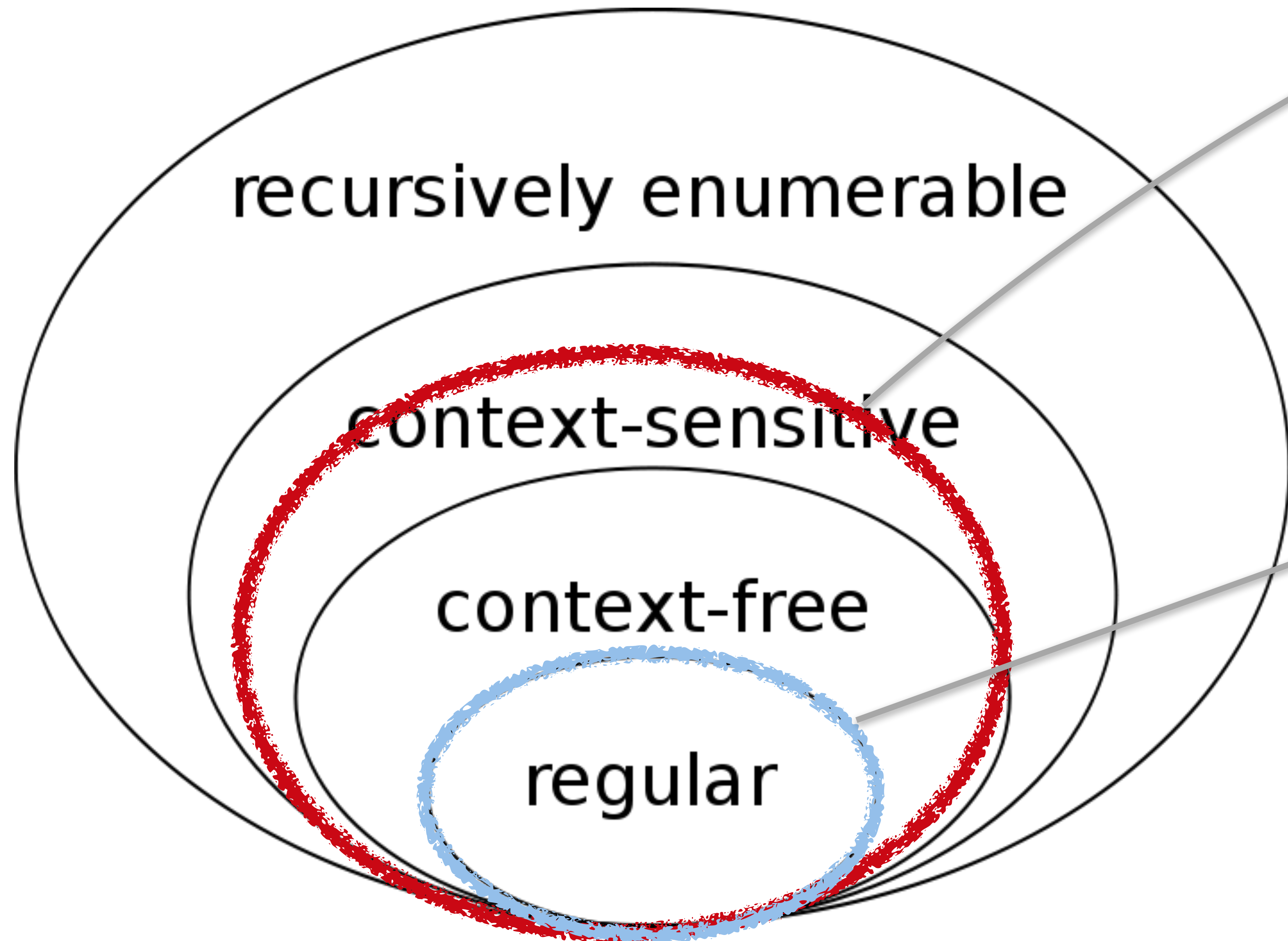
Current text pattern-matching tools are based on regular expressions. However, pure regular expressions have proven too weak a formalism for the task: many interesting patterns either are difficult to describe or cannot be described by regular expressions. Moreover, the inherent non-determinism of regular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



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

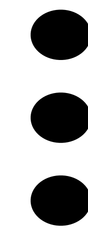
For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), to express the syntax of programming languages and protocols. The power of generative grammars to express ambiguity is crucial to their original purpose of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

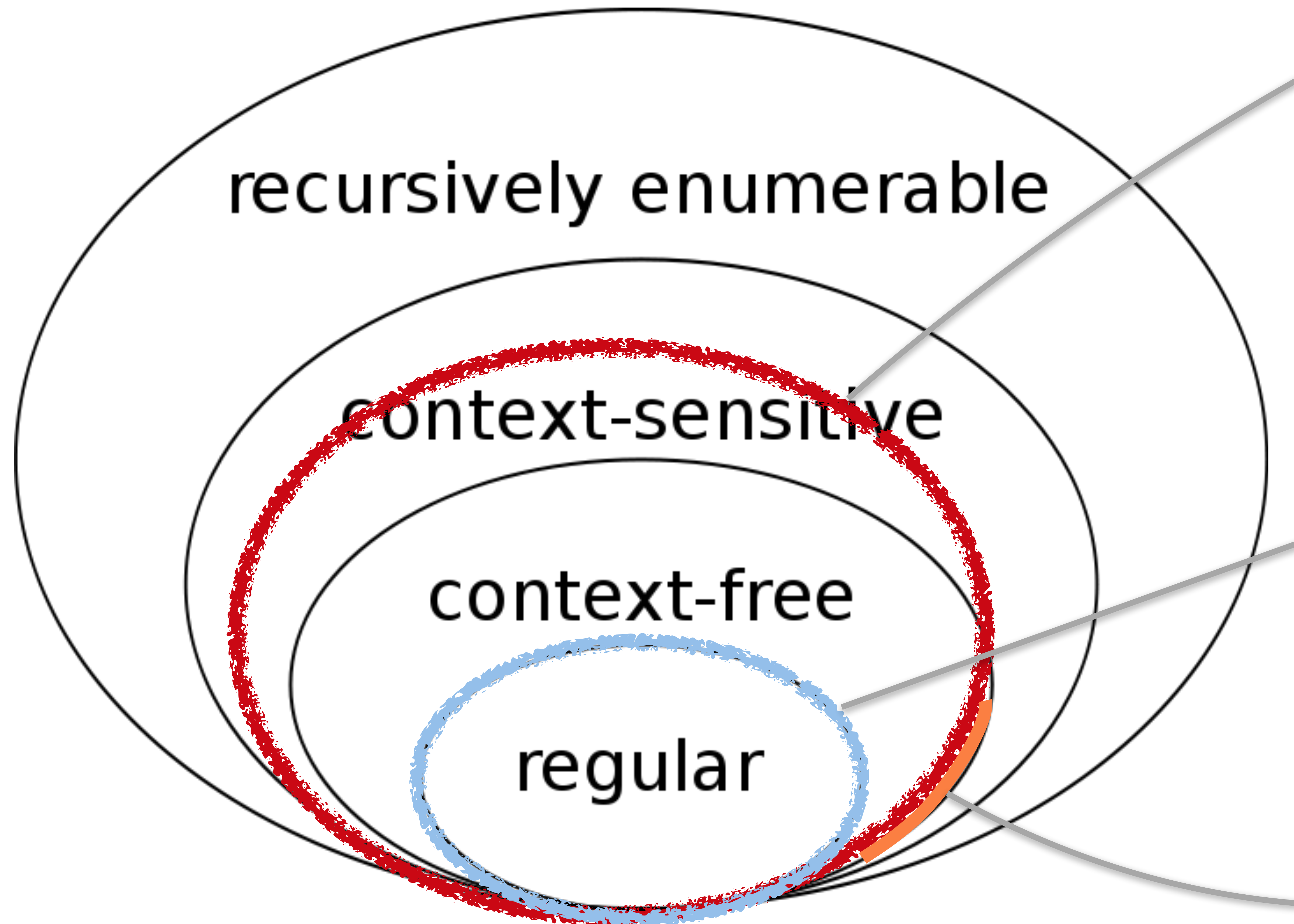
Current text pattern-matching tools are based on regular expressions. However, pure regular expressions have proven too weak a formalism for the task: many interesting patterns either are difficult to describe or cannot be described by regular expressions. Moreover, the inherent non-determinism of regular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



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

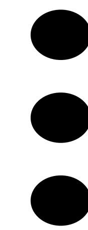
For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), to express the syntax of programming languages and protocols. The power of generative grammars to express ambiguity is crucial to their original purpose of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

RPL and regular expressions: similarities

RPL and regular expressions: similarities

```
pat?
pat+
pat*
pat{n}
pat{n,m}
```

Same syntax as regex

- RPL is greedy
- RPL is possessive

RPL and regular expressions: similarities

```
pat?
pat+
pat*
pat{n}
pat{n,m}
```

Same syntax as regex

- RPL is greedy
- RPL is possessive

```
[:name:]
[list]
[a-z]
[^…]
[cs1 cs2 …]
```

Simplified syntax from regex

- RPL requires escaping of
  `[ ] - ^`
- RPL allows one name or list or range at a time:
  `[a-z123]` not allowed

RPL and regular expressions: similarities

```
pat?
pat+
pat*
pat{n}
pat{n,m}
```

Same syntax as regex

- RPL is greedy
- RPL is possessive

```
[:name:]
[list]
[a-z]
[^…]
[cs1 cs2 …]
```

Simplified syntax from regex

- RPL requires escaping of
  `[ ] - ^`
- RPL allows one name or list or range at a time:
  `[a-z123]` not allowed

```
> pat
< pat
! pat
```

Simplified syntax from regex

- Compare to:
  `(?=(pat))`
  `(?<=(pat))`

RPL and regular expressions: a key difference

`p / q`

"Choice" is different:

RPL uses *ordered choice,* like other PEG grammars.

Meaning: First try **`p`**. If **`p`** fails, backtrack and try **`q`**.

Choices are possessive.

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 {} []

Community

- 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

Debugging

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

Paul R. Ehrlich

Trace a (mis-)match

```
$ echo '17:30:4' | rosie match time.rfc3339
$
```

Trace a (mis-)match

```
$ echo '17:30:4' | rosie match time.rfc3339
$
$ echo '17:30:4' | rosie trace time.rfc3339
Expression: {rfc3339_time {[:space:]}* {offset}?}
Looking at: 《17:30:4》 (input pos = 1)
No match
├── Expression: rfc3339_time
│   Looking at: 《17:30:4》 (input pos = 1)
│   No match
│   └── Expression: {hour ":" minute ":" second {secfrac}?}
│       Looking at: 《17:30:4》 (input pos = 1)
│       No match
│       ├── Expression: hour
│       │   Looking at: 《17:30:4》 (input pos = 1)
│       │   Matched 2 chars
│       ├── Expression: ":"
│       │   Looking at: 《:30:4》 (input pos = 3)
│       │   Matched 1 chars
│       ├── Expression: minute
│       │   Looking at: 《30:4》 (input pos = 4)
│       │   Matched 2 chars
│       ├── Expression: ":"
│       │   Looking at: 《:4》 (input pos = 6)
│       │   Matched 1 chars
│       ├── Expression: second
│       │   Looking at: 《4》 (input pos = 7)
│       │   No match
│       │   └── Expression: {{[0-5] [0-9]} / "60"}
│       │       Looking at: 《4》 (input pos = 7)
│       │       No match
│       │       ├── Expression: {[0-5] [0-9]}
```

Trace a (mis-)match

```
$ echo '17:30:4' | rosie match time.rfc3339
$
$ echo '17:30:4' | rosie trace time.rfc3339
Expression: {rfc3339_time {[:space:]}* {offset}?}
Looking at: 《17:30:4》 (input pos = 1)
No match
├── Expression: rfc3339_time
│   Looking at: 《17:30:4》 (input pos = 1)
│   No match
│   └── Expression: {hour ":" minute ":" second {secfrac}?}
│       Looking at: 《17:30:4》 (input pos = 1)
│       No match
│       ├── Expression: hour
│       │   Looking at: 《17:30:4》 (input pos = 1)
│       │   Matched 2 chars
│       ├── Expression: ":"
│       │   Looking at: 《:30:4》 (input pos = 3)
│       │   Matched 1 chars
│       ├── Expression: minute
│       │   Looking at: 《30:4》 (input pos = 4)
│       │   Matched 2 chars
│       ├── Expression: ":"
│       │   Looking at: 《:4》 (input pos = 6)
│       │   Matched 1 chars
│       ├── Expression: second
│       │   Looking at: 《4》 (input pos = 7)
│       │   No match
│       │   └── Expression: {{[0-5] [0-9]} / "60"}
│       │       Looking at: 《4》 (input pos = 7)
│       │       No match
│       │       ├── Expression: {[0-5] [0-9]}
```

Pattern definition

Input text

Failure point

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; 456; 7)"
{"data": "(1.2; 3; 456; 7)",
 "e": 17,
 "s": 1,
 "subs":
   [{"data": "(1.2; 3; 456; 7)",
     "e": 17,
     "s": 1,
     "subs":
       [{"data": "1.2; 3; 456; 7",
         "e": 16,
         "s": 2,
         "subs":
           [{"data": "1.2",
            {"data": " 3",
            {"data": " 456",
            {"data": " 7",
             "type": "des.find.*"}],
         "type": "des.semicolons"}],
     "type": "des.parentheses"}],
 "type": "*"}
Rosie>
```

Read-eval-print loop

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

```
Rosie> .match des.tryall "(1.2; 3; 456; 7)"
{"data": "(1.2; 3; 456; 7)",
 "e": 17,
 "s": 1,
 "subs":
   [{"data": "(1.2; 3; 456; 7)",
     "e": 17,
     "s": 1,
     "subs":
       [{"data": "1.2; 3; 456; 7",
         "e": 16,
         "s": 2,
         "subs":
           [{"data": "1.2",
            {"data": " 3",
            {"data": " 456",
            {"data": " 7",
             "type": "des.find.*"}],
         "type": "des.semicolons"}],
     "type": "des.parentheses"}],
 "type": "*"}
Rosie>
```

Read-eval-print loop

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

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
$
```

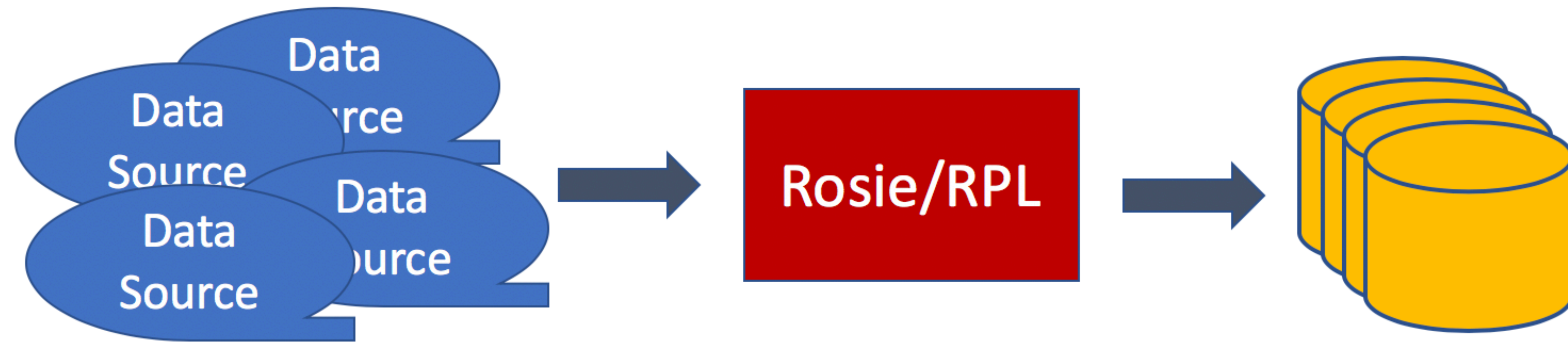
- ☑ Part of the documentation
- ☑ Regression when making changes
- ☑ Use them in app build/compile stage

Some non-CLI use cases

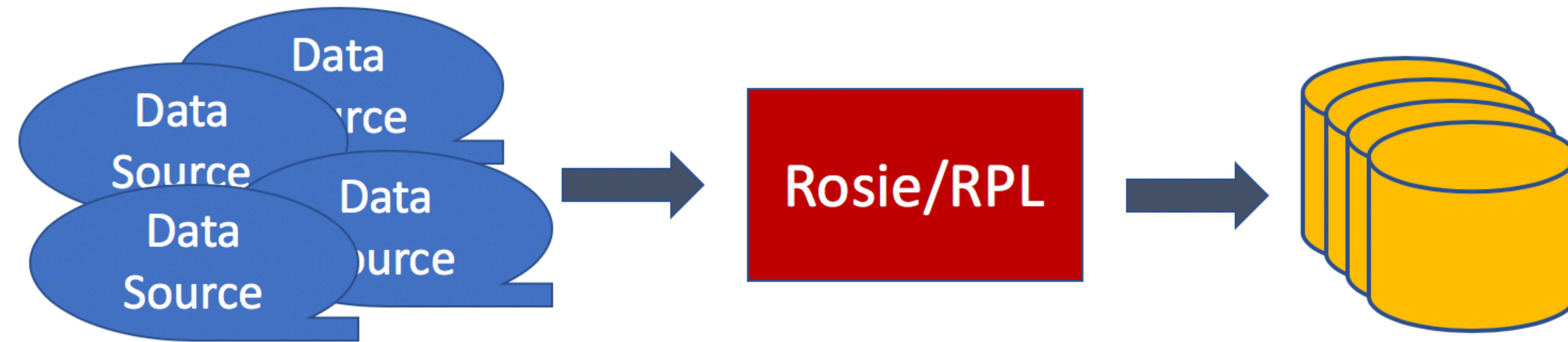
“I want to believe”

Fox Mulder, FBI

1. “Big data” parsing (streaming and batch)



1. “Big data” parsing (streaming and batch)



2. Mining source code repositories

- “Micro-grammar” approach:

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

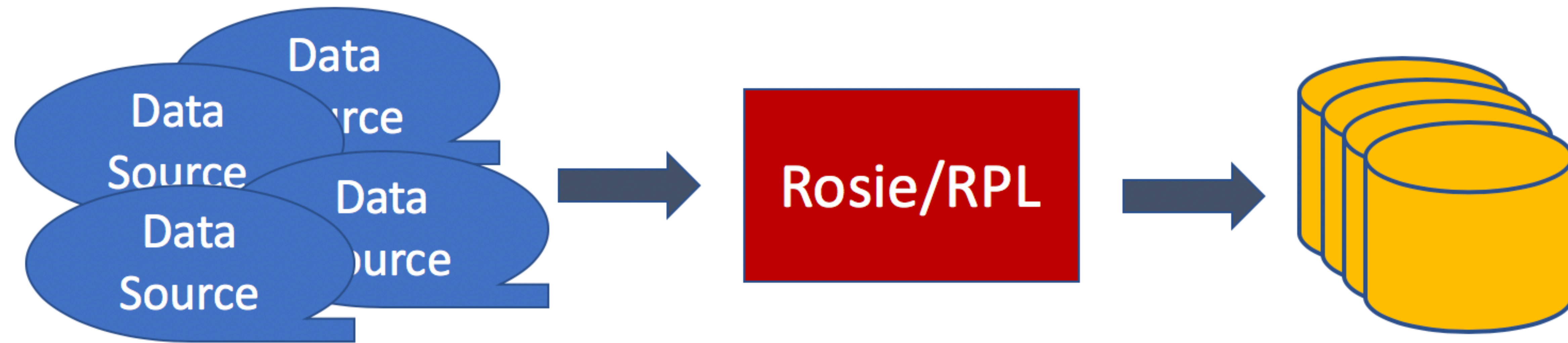
- NCSU students:

  Wrote RPL patterns to extract 6 kinds of language features from 10 different 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	✓	✓	✓	✓	✓	✓		

Work in progress

1. “Big data” parsing (streaming and batch)



2. 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:

  Wrote RPL patterns to extract 6 kinds of language features from 10 different 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	✓	✓	✓	✓	✓	✓		

Work in progress

3. Secure engineering principle: <u>Parse everything!</u>

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

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

Using Rosie in programs: Python example

Task

Given a string indicating start of a line comment, count the non-blank non-comment lines (i.e. lines of code).

```
import rosie
engine = rosie.engine()
source_line, errs = engine.compile(bytes('!{[:space:]* "' + comment_start + '"/$}'))
# [snipped: error check]

def is_source(line):
    if not line: return False
    match, leftover, abend, t0, t1 = engine.match(source_line, bytes(line), 1, b"bool")
    return match and True or False

def count(f):
    count = 0
    for line in f:
        if is_source(line): count += 1
    return count
```

[snip]

Using Rosie in programs: Python example

<u>Task</u>

Given a string indicating start of a line comment, count the non-blank non-comment lines (i.e. lines of code).

```
import rosie
engine = rosie.engine()
source_line, errs = engine.compile(bytes('!{[:space:]* "' + comment_start + '"/$}'))
# [snipped: error check]

def is_source(line):
    if not line: return False
    match, leftover, abend, t0, t1 = engine.match(source_line, bytes(line), 1, b"bool")
    return match and True or False

def count(f):
    count = 0
    for line in f:
        if is_source(line): count += 1
    return count
```

1. Get engine

[snip]

Using Rosie in programs: Python example

Task

Given a string indicating start of a line comment, count the non-blank non-comment lines (i.e. lines of code).

```
import rosie
engine = rosie.engine()
source_line, errs = engine.compile(bytes('!{[:space:]* "' + comment_start + '"/$}'))
# [snipped: error check]

def is_source(line):
    if not line: return False
    match, leftover, abend, t0, t1 = engine.match(source_line, bytes(line), 1, b"bool")
    return match and True or False

def count(f):
    count = 0
    for line in f:
        if is_source(line): count += 1
    return count
```

[snip]

1. Get engine

2. Compile RPL

Using Rosie in programs: Python example

Task

Given a string indicating start of a line comment, count the non-blank non-comment lines (i.e. lines of code).

```
import rosie
engine = rosie.engine()
source_line, errs = engine.compile(bytes('!{[:space:]* "' + comment_start + '"/$}'))
# [snipped: error check]

def is_source(line):
    if not line: return False
    match, leftover, abend, t0, t1 = engine.match(source_line, bytes(line), 1, b"bool")
    return match and True or False

def count(f):
    count = 0
    for line in f:
        if is_source(line): count += 1
    return count
[snip]
```

1. Get engine

2. Compile RPL

3. Match

Using Rosie in programs: Improvements coming, and <u>help wanted</u>

Today:

Go

Once and future:

Performance

Total time (seconds)

250.00
225.00
200.00
175.00
150.00
125.00
100.00
75.00
50.00
25.00
0.00

0
1,000,000
2,000,000
3,000,000
4,000,000

Number of input lines (syslog)

- rosie -o json
- rosie_dev -o json
- grok/ruby
- grok/jruby

Grok/ruby

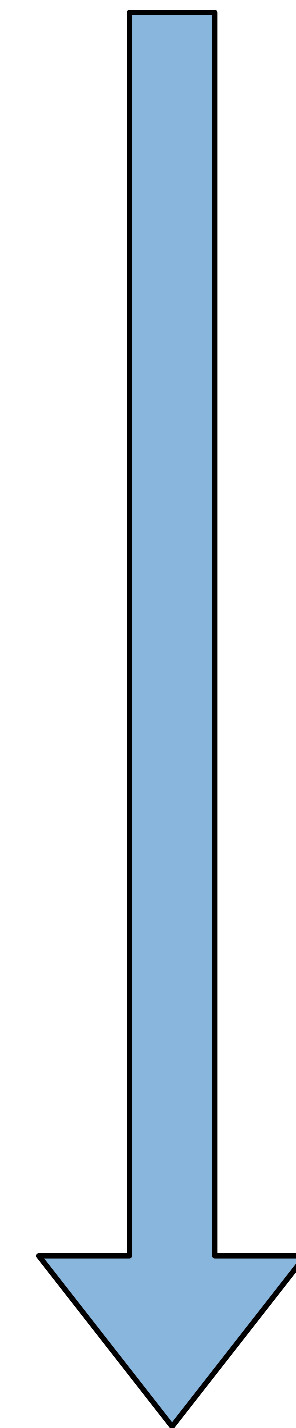
Failed with utf8 error
before finishing

Grok/jruby

Failed with utf8 error
before finishing

Rosie 1.0.0

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.

Roadmap & Community

“If you want to go fast, go alone.
If you want to go far, go together.”

“Proverb”

Roadmap

Roadmap

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

Extensibility
User-written macros
User-written output encoders

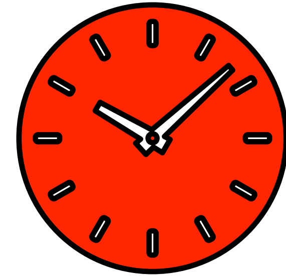
Command line/scripting convenience
Traverse directories
Follow links or not, etc.

Compiler Optimizations
Common subexpression elimination
New vm instructions
Flow analysis

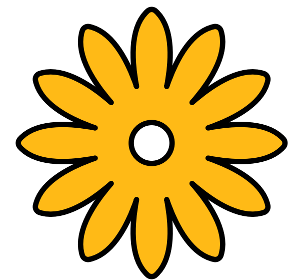
Regex-to-rosie converter
Re-use existing regex
Give them unit tests
Debug them

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

Join the Rosie user community!

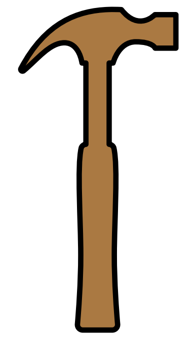


```
git clone …
make;
make install (optional)
```



Contribute Patterns

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



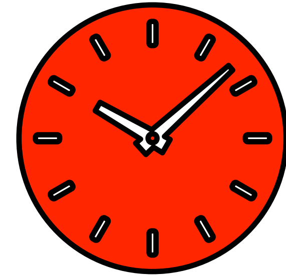
Write Tools

- 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

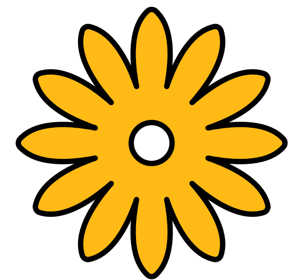
Join the Rosie user community!



```
git clone …
make;
make install (optional)
```

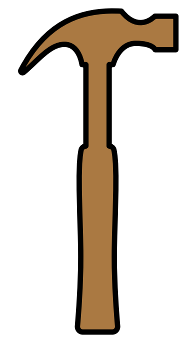
Or: brew install rosie

Or: pip install rosie



Contribute Patterns

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

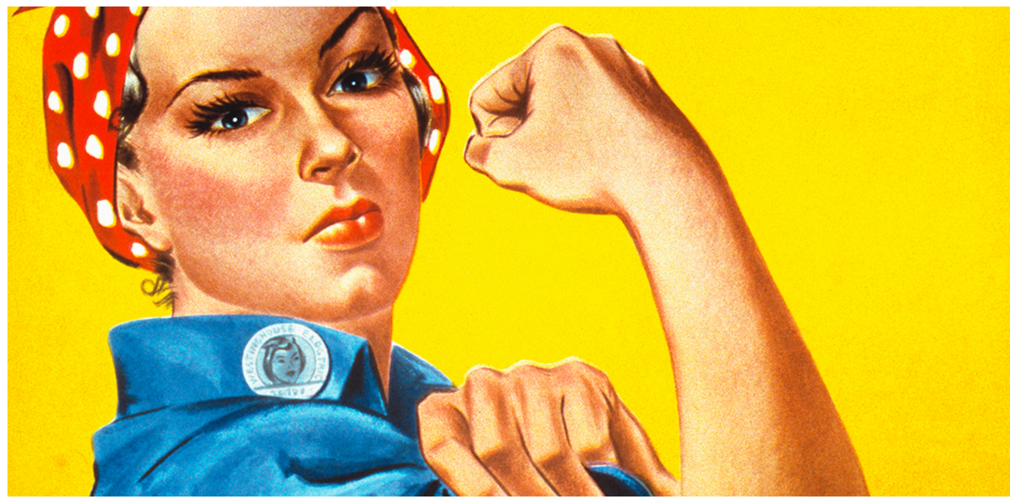


Write Tools

- 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



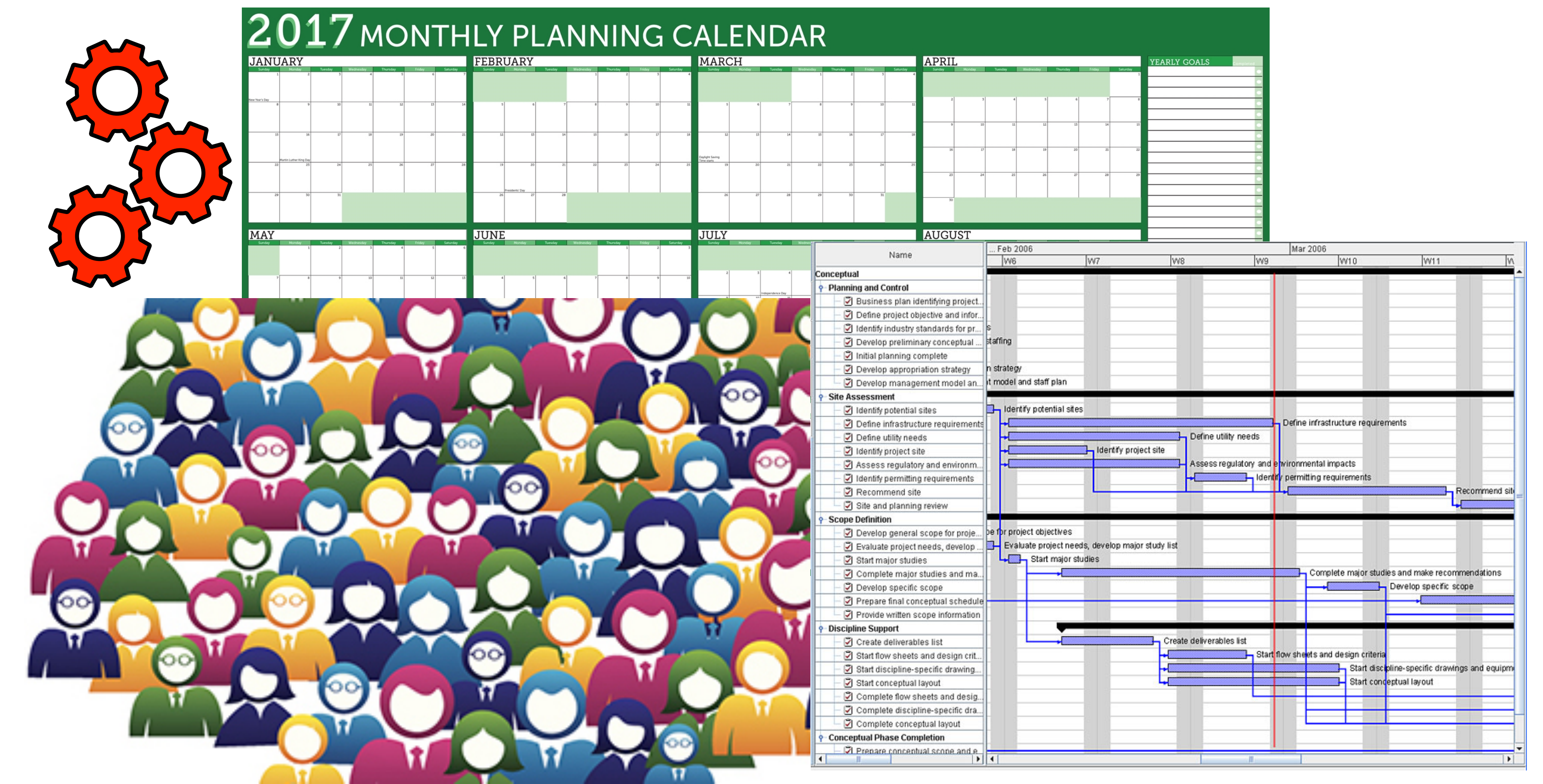
Conclusion

Conclusion

Faster

- Dev time:
  - ✓ library of patterns you don’t have to write
  - ✓ new patterns composed of existing patterns
- Run time: matching performance very good

Conclusion

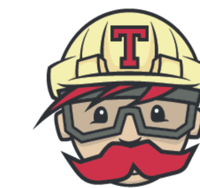


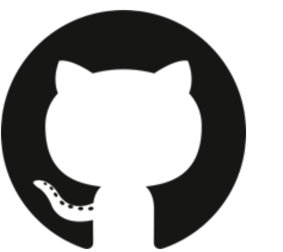
Faster

- ✦ Dev time:
  - ✓ library of patterns you don't have to write
  - ✓ new patterns composed of existing patterns
- ✦ Run time: matching performance very good

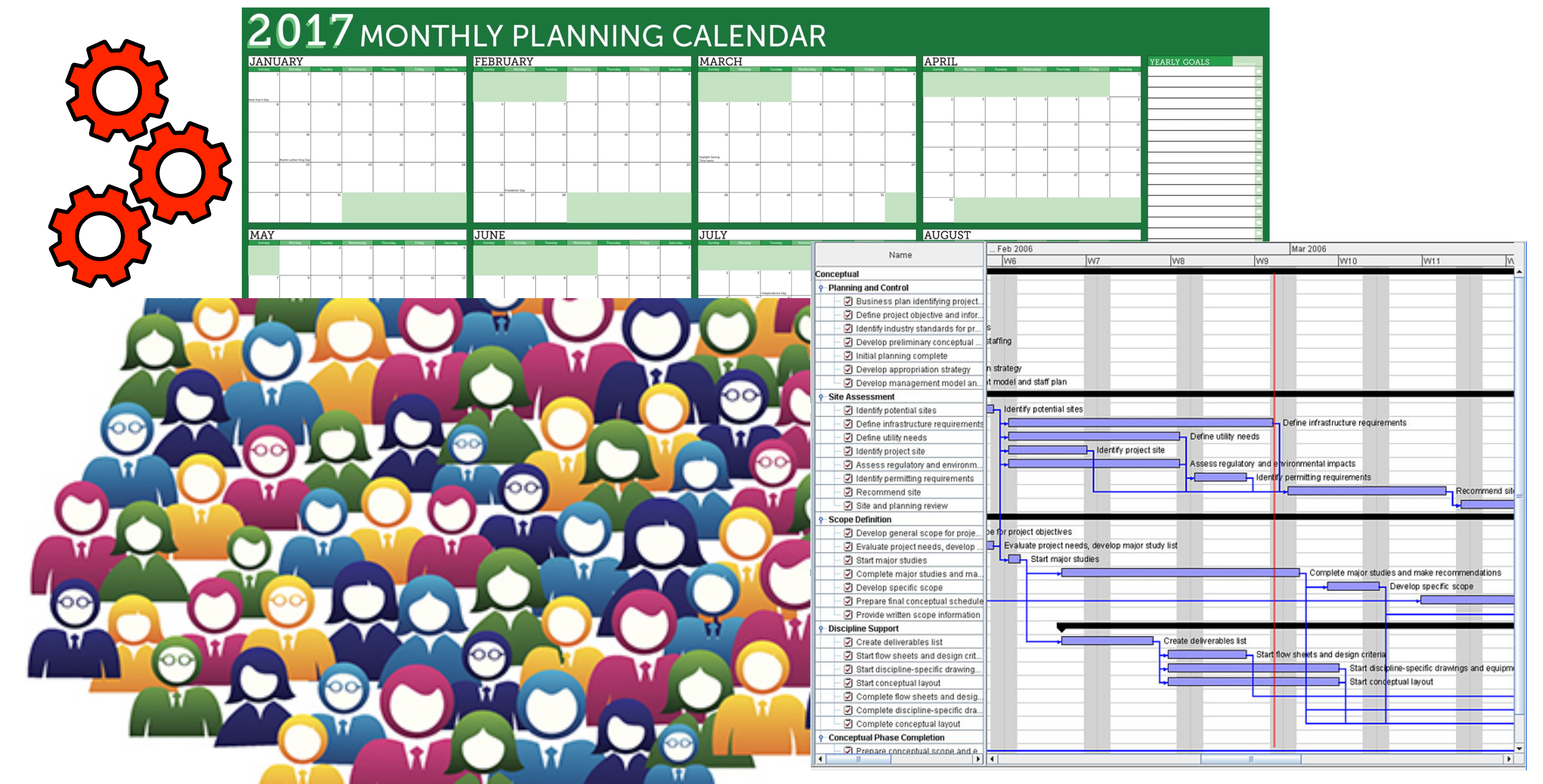
Better

- ✦ shareable libraries
- ✦ conformance to RFCs
- ✦ readable syntax, and strict semantics (and no flags)
- ✦ plays well with DevOps tools (git/diff, package management, unit tests)





Conclusion

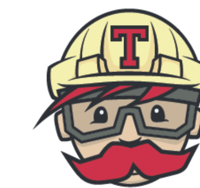


Faster

- Dev time:
  - ✓ library of patterns you don't have to write
  - ✓ new patterns composed of existing patterns
- Run time: matching performance very good

Better

- shareable libraries
- conformance to RFCs
- readable syntax, and strict semantics (and no flags)
- plays well with DevOps tools (git/diff, package management, unit tests)



Cheaper

- ROI in reduced development and maintenance costs
- And, it's free open source software (MIT license)

Thank you!

Additional slides for
reference

Rosie Pattern Language features

- **Pattern libraries**
  - Standard library
  - 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 can be used like grep)
  - Readable, maintainable
  - Works well with git/diff, pipelines (unit tests), dependency mgmt

Formal basis:

✦ Parser combinators
✦ Based on Parsing Exp. Grammars
✦ Linear-time in input size: $O(n)$
✦ Not a "packrat" implementation

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: accept recursive structures
- PEGs [Ford, 2004]
  - "Scanner-less parsing"
  - Linear time matching
  - 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

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

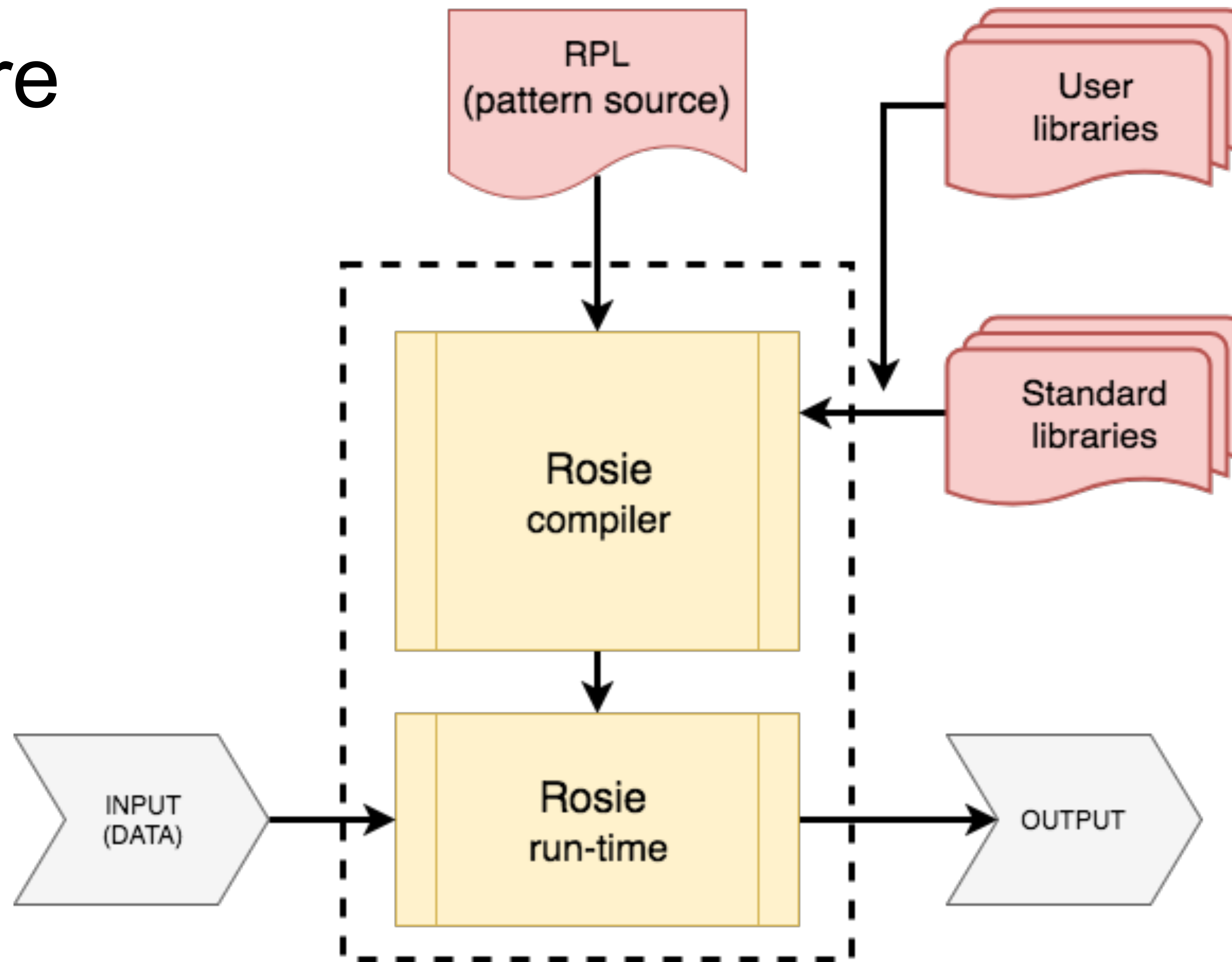
Current text pattern-matching tools are based on regular expressions. However, pure regular expressions have proven too weak a formalism for the task: many interesting patterns either are difficult to describe or cannot be described by regular expressions. Moreover, the inherent non-determinism of regular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

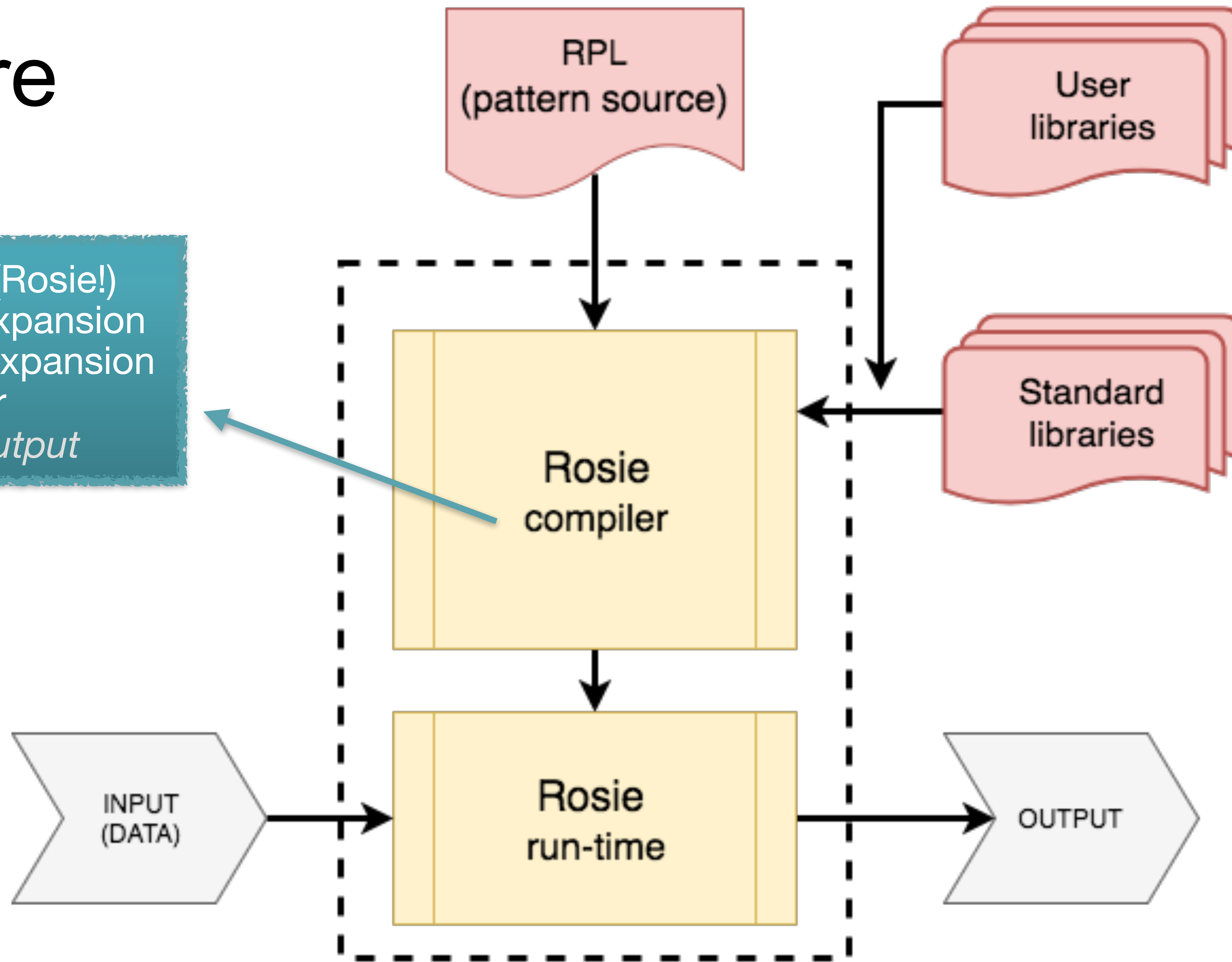
Architecture



Compiler, run-time, std lib total ~500 KB on disk

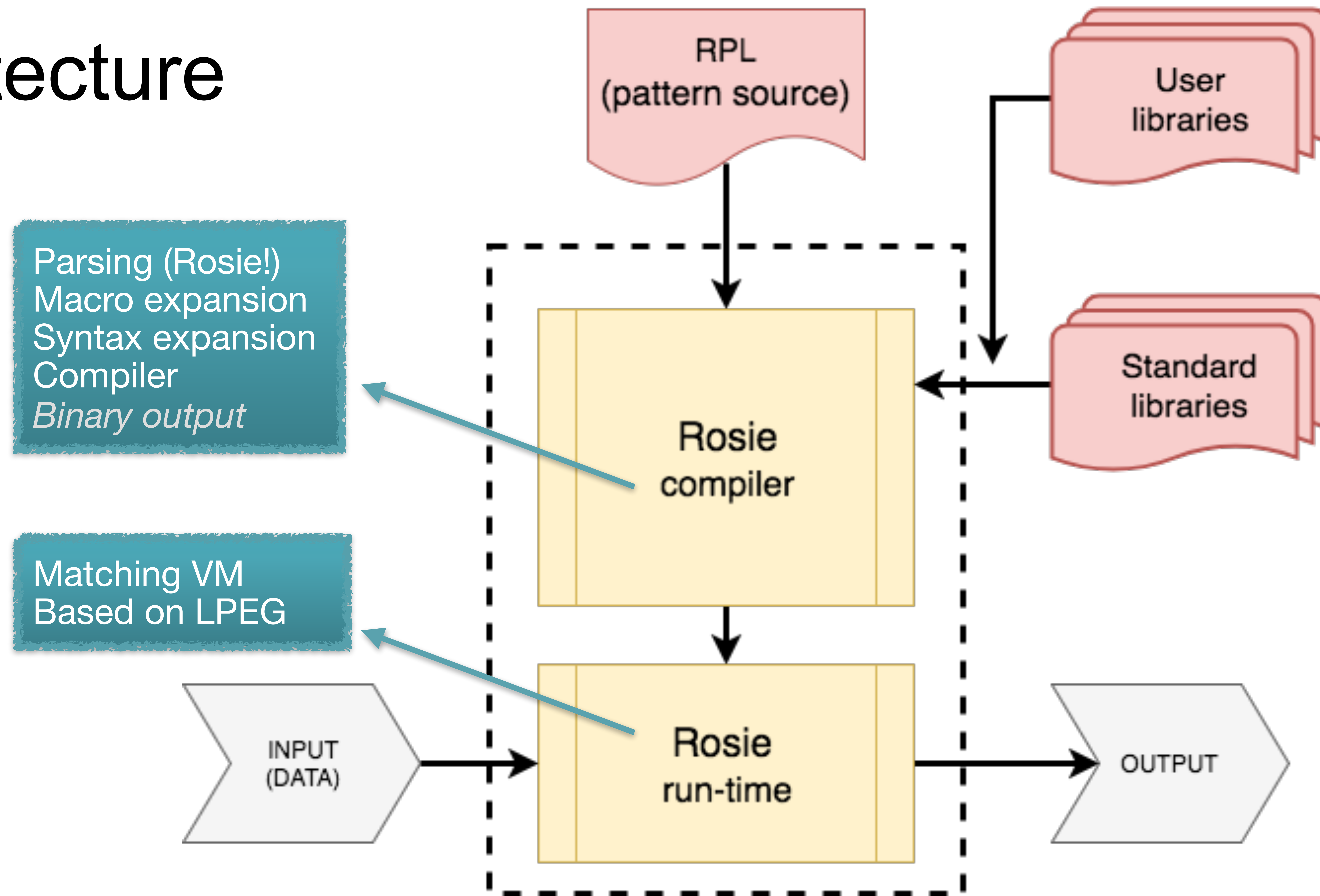
Architecture

Parsing (Rosie!)
Macro expansion
Syntax expansion
Compiler
Binary output



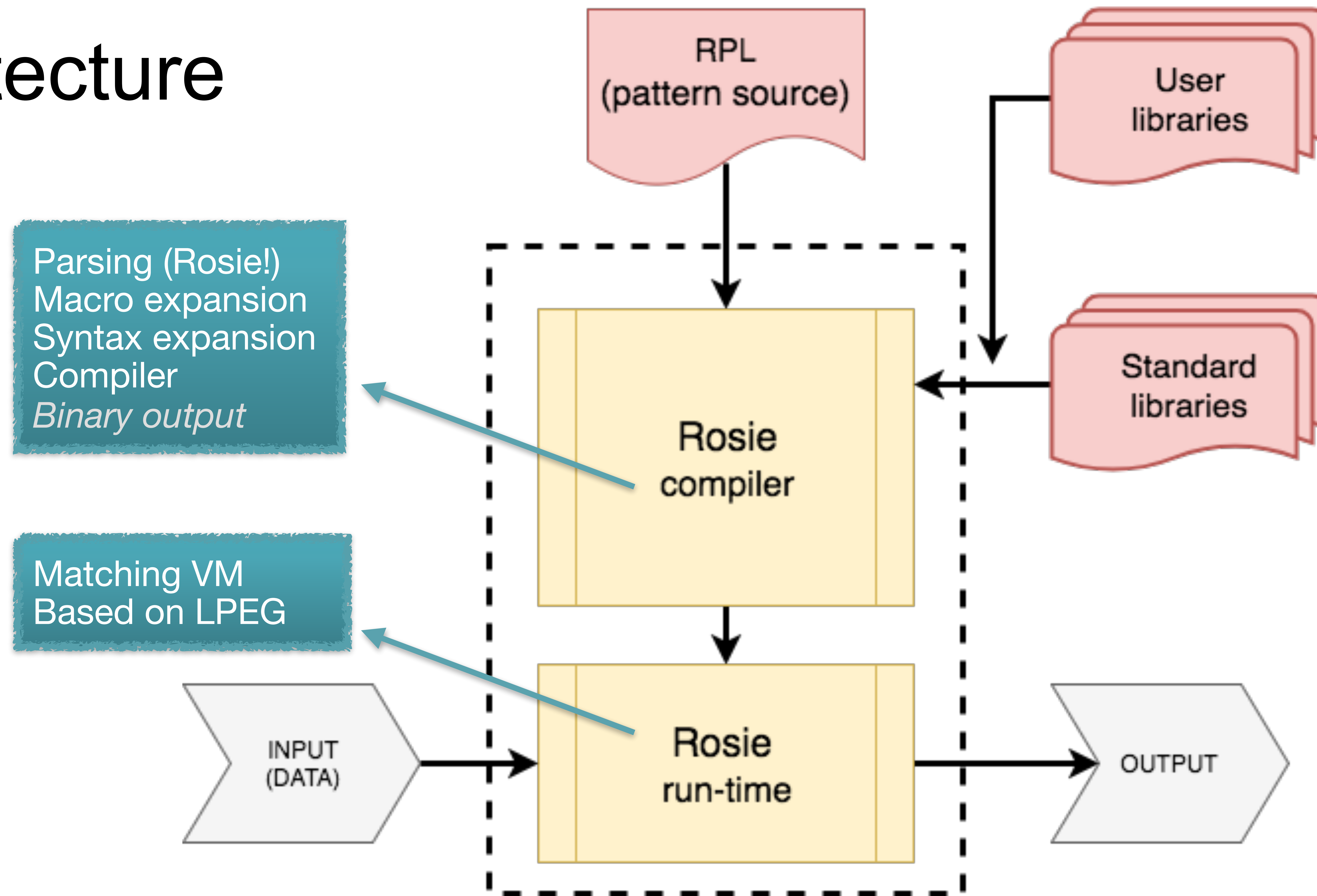
Compiler, run-time, std lib total ~500 KB on disk

Architecture



Compiler, run-time, std lib total ~500 KB on disk

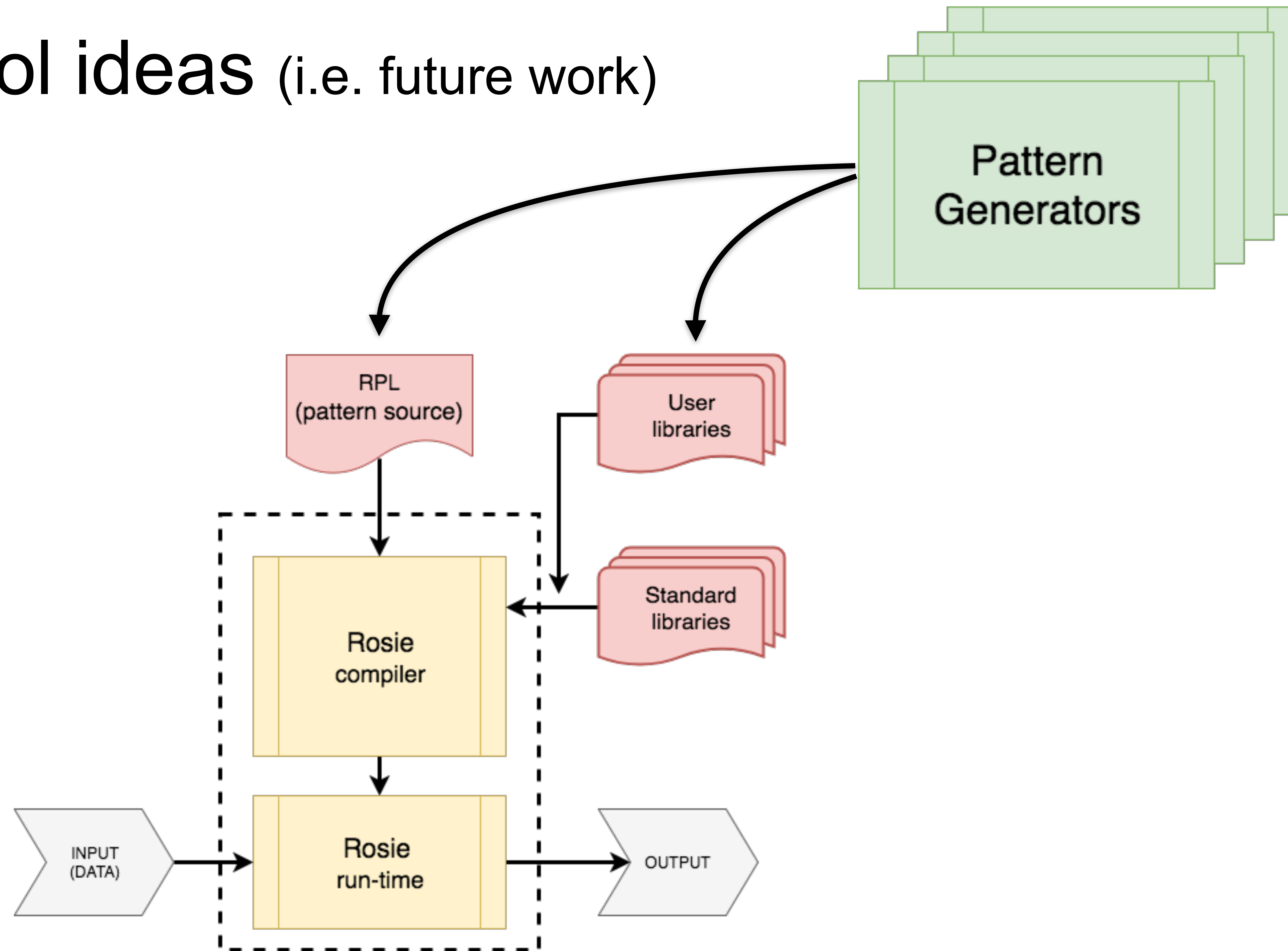
Architecture



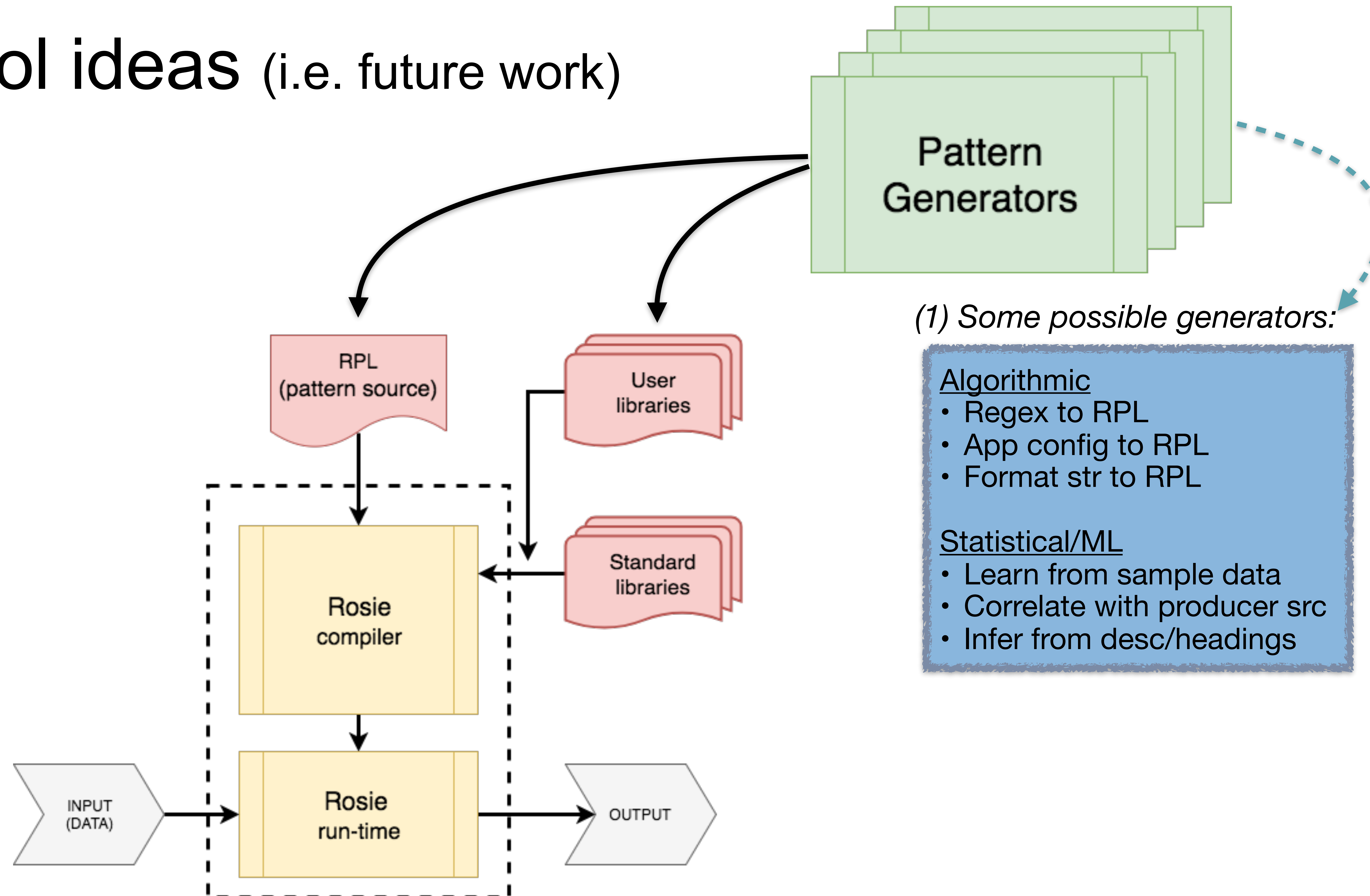
Compiler, run-time, std lib total ~500 KB on disk

RPL Compiler	~ 5k sloc, Lua	+ Requires `liblua.a` (330 KB)
Rosie Engine	<< 1k sloc, Lua ~ 3k sloc, C	+ Requires `liblua.a` (330 KB) + Requires `cjson.so` (50KB)
Rosie CLI, REPL	< 1k sloc, Lua	+ Requires `readline.so` (from user)

Cool ideas (i.e. future work)



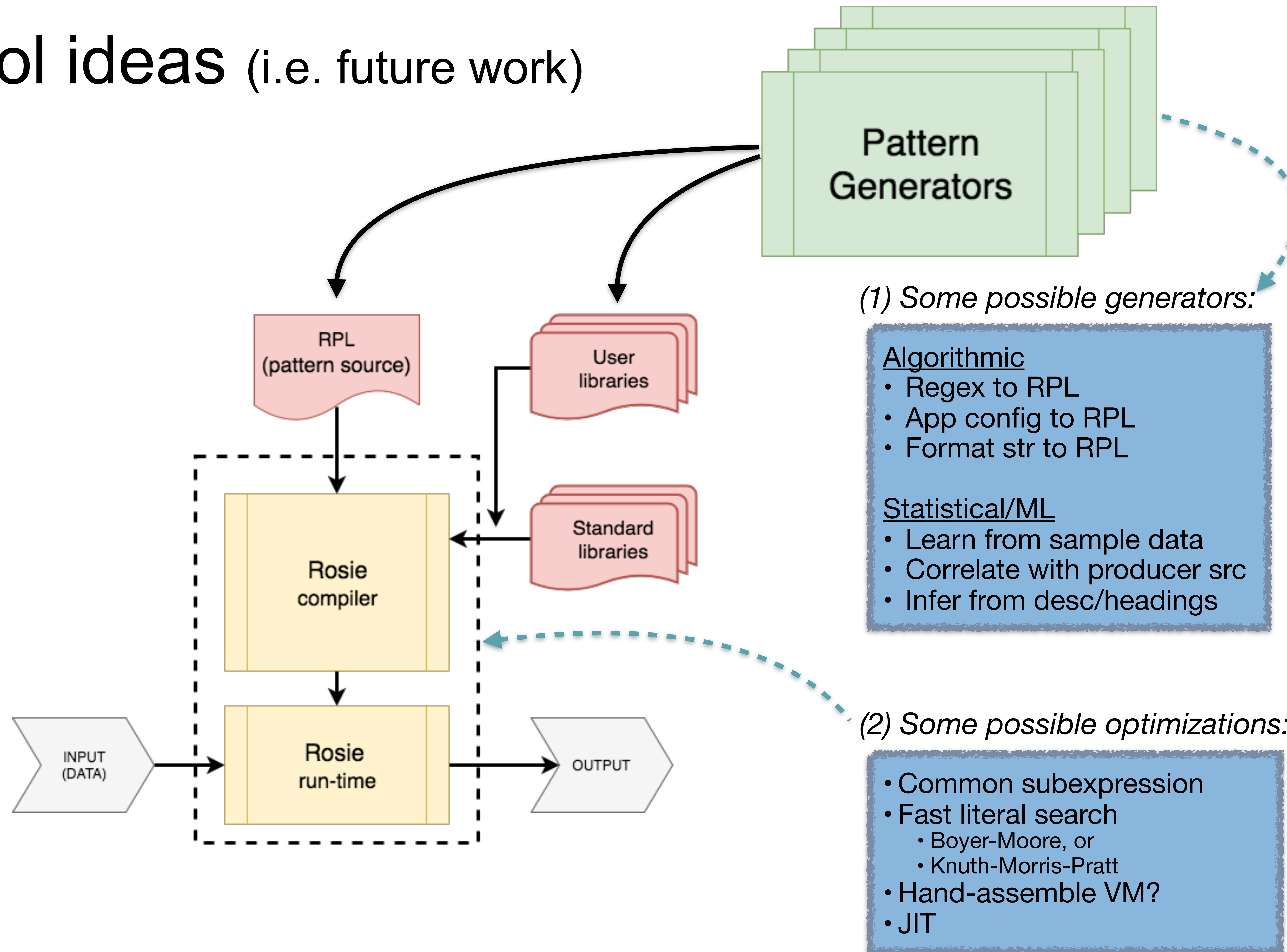
Cool ideas (i.e. future work)



(1) Some possible generators:

Algorithmic
- Regex to RPL
- App config to RPL
- Format str to RPL

Statistical/ML
- Learn from sample data
- Correlate with producer src
- Infer from desc/headings


Cool ideas (i.e. future work)
Pattern Generators
RPL (pattern source)
User libraries
Standard libraries
Rosie compiler
Rosie run-time
INPUT (DATA)
OUTPUT
(1) Some possible generators:
Algorithmic
• Regex to RPL
• App config to RPL
• Format str to RPL
Statistical/ML
• Learn from sample data
• Correlate with producer src
• Infer from desc/headings
(2) Some possible optimizations:
• Common subexpression
• Fast literal search
• Boyer-Moore, or
• Knuth-Morris-Pratt
• Hand-assemble VM?
• JIT

Rosie is self-hosting

- Rosie is a parser, and Rosie is used to parse Rosie Pattern Language
- About 110 lines of RPL (core) to define the RPL
- 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!*

```
$ rosie match -o line '!{[:space:]*$} !{[:space:]* "--"}' rpl_1_1.rpl | wc
     111     652    4155
```