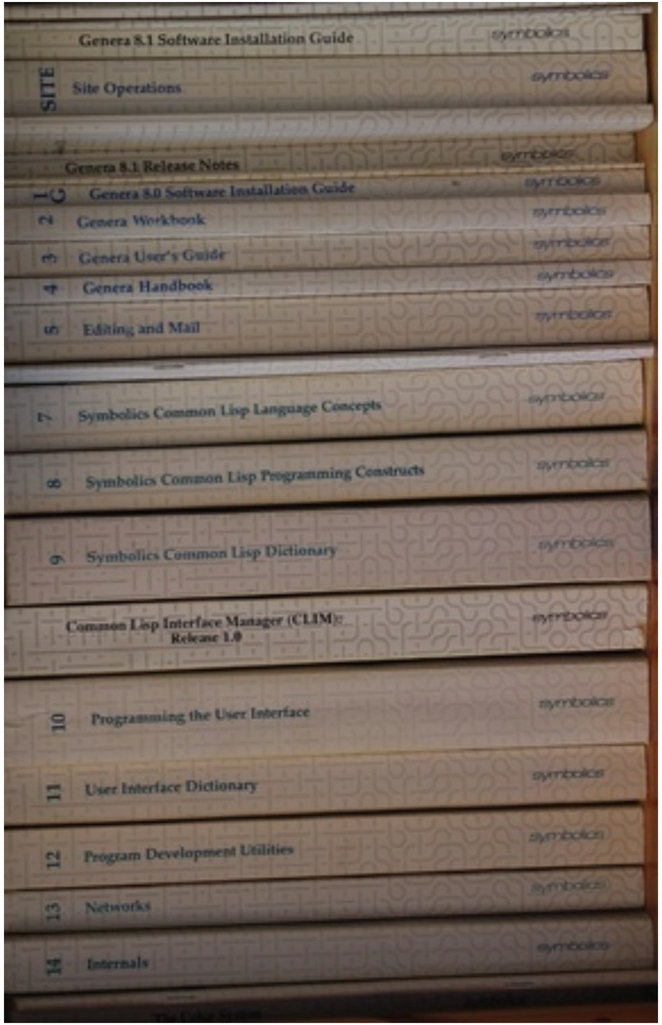
Regex Considered Harmful: Use Rosie Pattern Language Instead

Lua Workshop 2016

Jamie A. Jennings, Ph.D.
IBM Cloud CTO Office
October, 2016

Disclaimer:

These slides describe work I have done for my employer, IBM, but I am speaking here only for myself, not for IBM.


Genera 8.1 Software Installation Guide
SITE
Site Operations
Genera 8.1 Release Notes
Genera 8.0 Software Installation Guide
2 Genera Workbook
3 Genera User's Guide
4 Genera Handbook
5 Editing and Mail
7 Symbolics Common Lisp Language Concepts
8 Symbolics Common Lisp Programming Constructs
9 Symbolics Common Lisp Dictionary
Common Lisp Interface Manager (CLIM): Release 1.0
10 Programming the User Interface
11 User Interface Dictionary
12 Program Development Utilities
13 Networks
14 Internals
symbolics

Problem space

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

“But most of it is like cat videos on YouTube” Nate Silver

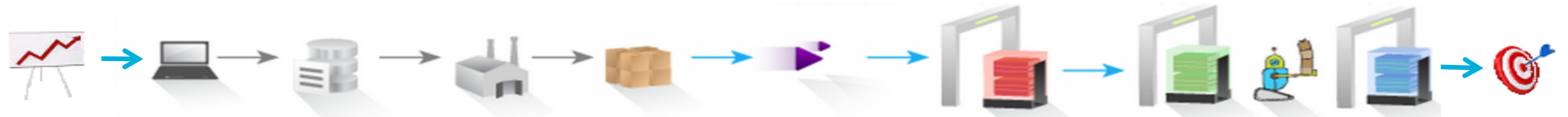
Estimates are that less than 0.5% of data is ever analyzed!

(Antonio Regalado, MIT Technology Review, https://www.technologyreview.com/s/514346/the-data-made-me-do-it/)

Disclaimer: This is a *manager slide*

DevOps Analytics Team:

applying machine learning and other analytics to DevOps data to improve quality and efficiency of software development



In:

- Business Objectives: Business data
- Development: • Logs • Events • Metrics
- SCM: • Logs • Events • Metrics
- Build: • Logs • Events • Metrics
- Package Repo: • Logs • Events • Metrics
- Deploy: • Logs • Events • Metrics
- Test: • Logs • Events • Metrics
- Stage: • Logs • Events • Metrics
- Prod: • Logs • Events • Metrics
- Business Results: Business data

DevOps Analytics Platform (includes Rosie)

Out:

- Root cause Analysis
- Impact Analysis
- Anomaly Detection
- Web Experience Analysis
- A/B Testing Analysis
- Usage Patterns Recognition
- Correlation between Usage Patterns and Business Risk / Growth Opportunities
- Business Analytics

Log files: many formats, often mixed in the same file

E.g. Apache Spark logs contain “standard” entries mixed with Java exceptions and Python tracebacks

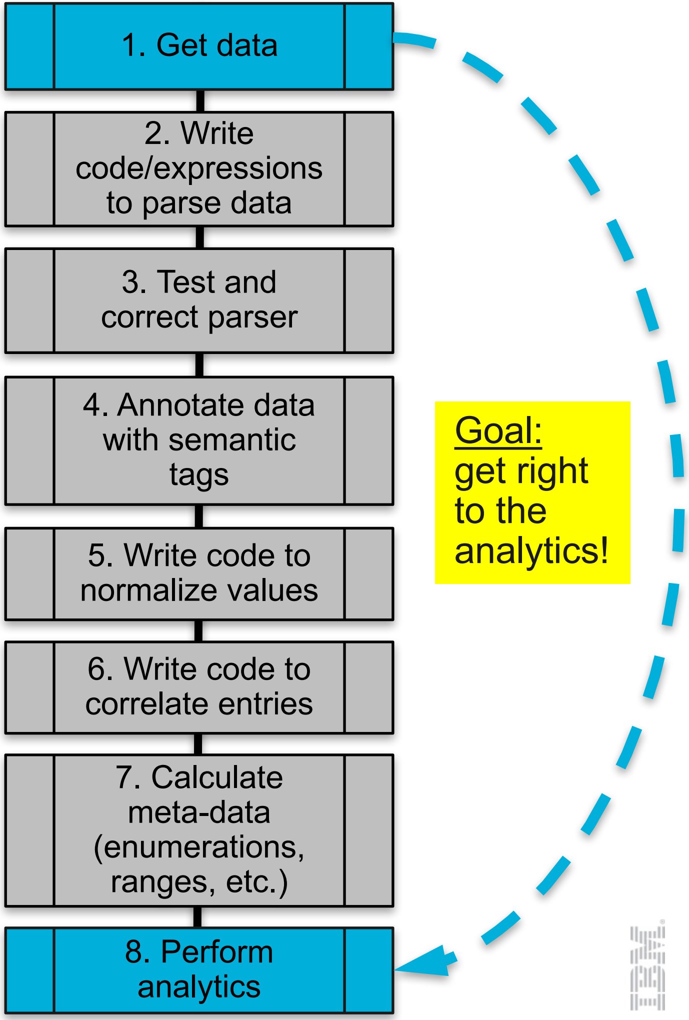
```
16/02/08 10:14:33 INFO SparkContext: Running Spark version 1.6.0
16/02/08 10:14:33 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java
[...]
16/02/08 10:14:38 ERROR Executor: Exception in task 1.0 in stage 5.0 (TID 10)
java.lang.NullPointerException
   at org.apache.spark.sql.types.Metadata$.org$apache$spark$sql$types$Metadata$$toJsonValue(Metadata.scala:185)
   at org.apache.spark.sql.types.Metadata$$anonfun$2.apply(Metadata.scala:172)
   at org.apache.spark.sql.types.Metadata$$anonfun$2.apply(Metadata.scala:172)
   at
   scala.collection.TraversableLike$$anonfun$map$1.apply(TraversableLike.scala:244)
[...]
16/02/08 10:14:38 INFO DAGScheduler: Job 4 failed: collect at
/home/al/dev/git/devopsrca/pydevops/devops/test/rca_test.py:23, took 0.138982 s
Traceback (most recent call last):
 File "/home/al/dev/git/devopsrca/pydevops/devops/test/rca_test.py", line 23, in <module>
   print ind.collect()
 File "/opt/spark-1.6.0-bin-hadoop2.6/python/pyspark/sql/dataframe.py", line 280, in collect
   port = self._jdf.collectToPython()
 File "/opt/spark-1.6.0-bin-hadoop2.6/python/lib/py4j-0.9-src.zip/py4j/java_gateway.py", line 813, in __call__
 File "/opt/spark-1.6.0-bin-hadoop2.6/python/pyspark/sql/utils.py", line 45, in deco
   return f(*a, **kw)
```

How to spend data science effort?

- Recurring estimate: **80%** of analysis effort is preparing the data
- Much of the world's data is unstructured or semi-structured
- Therefore, much of the world's data needs to be:
  - **Parsed** to extract the useful bits
  - **Annotated** and labeled
  - **Normalized** to standard formats
  - **Sanitized** to hide sensitive bits
  - And **correlated** with related bits of information

<u>The key issue is scale:</u>

✓ Lots of data formats ("variety")
✓ Lots of data ("volume")
✓ Near-real-time requirements ("velocity")



Current approaches

"If the only tool you have is a hammer…"

Abraham Maslow

Grok’s networking patterns

Networking

MAC (?:%{CISCOMAC}|%{WINDOWSMAC}|%{COMMONMAC})

CISCOMAC (?:(?:[A-Fa-f0-9]{4}\.){2}[A-Fa-f0-9]{4})

WINDOWSMAC (?:(?:[A-Fa-f0-9]{2}-){5}[A-Fa-f0-9]{2})

COMMONMAC (?:(?:[A-Fa-f0-9]{2}:){5}[A-Fa-f0-9]{2})

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}))|:))|(([0-9A-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}))|:)))(%.+)?

IPV4 (?<![0-9])(?:(?:25[0-5]|2[0-4][0-9]|[0-1]?[0-9]{1,2})[.](?:25[0-5]|2[0-4][0-9]|[0-1]?[0-9]{1,2})[.](?:25[0-5]|2[0-4][0-9]|[0-1]?[0-9]{1,2})[.](?:25[0-5]|2[0-4][0-9]|[0-1]?[0-9]{1,2}))(?![0-9])

IP (?:%{IPV6}|%{IPV4})

HOSTNAME \b(?:[0-9A-Za-z][0-9A-Za-z-]{0,62})(?:\.(?:[0-9A-Za-z][0-9A-Za-z-]{0,62}))*(\.?|\b)HOST %{HOSTNAME}

IPORHOST (?:%{HOSTNAME}|%{IP})

HOSTPORT %{IPORHOST}:%{POSINT}

Regex issue #1: Notoriously hard to read & maintain

- **Unmaintainable** dense, cryptic syntax
- **Un-composable** expressions
- **Not portable** across implementations

"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 #2: Performance is highly variable

"The worst-case exponential-time backtracking strategy [is] used almost everywhere else, including ed, sed, Perl, PCRE, and Python." (Russ Cox https://swtch.com/~rsc/regexp/regexp2.html)

E.g. matching this 29-character string takes around 36 seconds in Perl*

```
$input = "aaaaaaaaaaaaaaaaaaaaaaaaaaaa";
$re ="a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?aaaaaaaaaaaaaaaaaaaaaaaaaaaa";
```

And this real-world example takes around 65 seconds 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";
```

Regex considered harmful (at scale)

Lessons

(1) **Do not use in a big data pipeline**
 – Not implemented efficiently; performance highly variable
 – Limited portability; tied to the necessary scaffolding (Perl, Python, Ruby, Java, js, …)

(2) **Avoid long expressions**
 – Dense syntax; hard to read; nearly impossible to maintain
 – But composition is fraught!

(3) **Avoid large collections of expressions**
 – Dense syntax; hard to read; nearly impossible to maintain
 – Semantics and capabilities vary across implementations

Rosie Pattern Language

"All progress depends on the unreasonable [woman]"

George Bernard Shaw, paraphrased

Rosie Pattern Engine

INPUT

```
2015-08-23T03:36:25-05:00 10.108.69.93 sshd[16537]: Did not receive identification string from 208.43.117.11
2015-08-23T03:36:30-05:00 10.91.62.208 emerald[10991]: "ui1_db1" #80338: max number of retransmissions (20) reached STATE_
2015-08-23T03:36:31-05:00 10.91.62.206 emerald[1084]: "ui1_db2" #85168: discarding duplicate packet; already STATE_I1
2015-08-23T03:36:31-05:00 10.91.62.206 emerald[1084]: "ui1_db1" #84039: next payload type of ISAKMP Hash Payload ...ue: 211
2015-08-23T03:36:31-05:00 10.91.62.206 emerald[1084]: "ui1_db1" #84039: malformed payload in packet
2015-08-23T03:36:31-05:00 10.91.62.206 emerald[1084]: "ui1_db1" #84039: next payload type of ISAKMP Hash Payload ...known value: 196
2015-08-23T03:36:31-05:00 10.91.62.206 emerald[1084]: "ui1_db1" #84039: malformed payload in packet
```

OUTPUT


```
{"syslog" : {"text" : "2015-08-23T03:36:30-05:00 10.91.62.208 pluto[10991]: \"ui1_db1\" #80338: max number of [...]",
             "pos" : 1,
             "subs" : [ {"datetime.datetime_RFC3339" : {"text" : "2015-08-23T03:36:30-05:00",
                                                        "pos" : 1,
                                                        "subs" : [ {"datetime.full_date_RFC3339" : {"text" : "2015-08-23",
                                                                                                     "pos" : 1 } },
                                                                   {"datetime.full_time_RFC3339" : {"text" : "03:36:30-05:00",
                                                                                                     "pos" : 12 } } ] } },
                        {"network.ip_address" : {"text" : "10.91.62.208",
                                                 "pos" : 27 } },
                        {"process" : {"text" : "pluto[10991]",
                                      "pos" : 40,
                                      "subs" : [ {"common.word" : {"text" : "pluto",
                                                                   "pos" : 40 } },
                                                 {"common.int" : {"text" : "10991",
                                                                  "pos" : 46 } } ] } },
                        {"MAX" : {"text" : "\"ui1_db1\" #80338: max number of retransmissions (20) reached STATE_R1",
                                  "pos" : 54,
                                  "subs" : [ {"common.identifier_plus" : {"text" : "ui1_db1",
                                                                          "pos" : 55 } },
                                             {"common.int" : {"text" : "80338",
                                                              "pos" : 73 } } ] } } ] } }
```

RPL is designed like a programming language

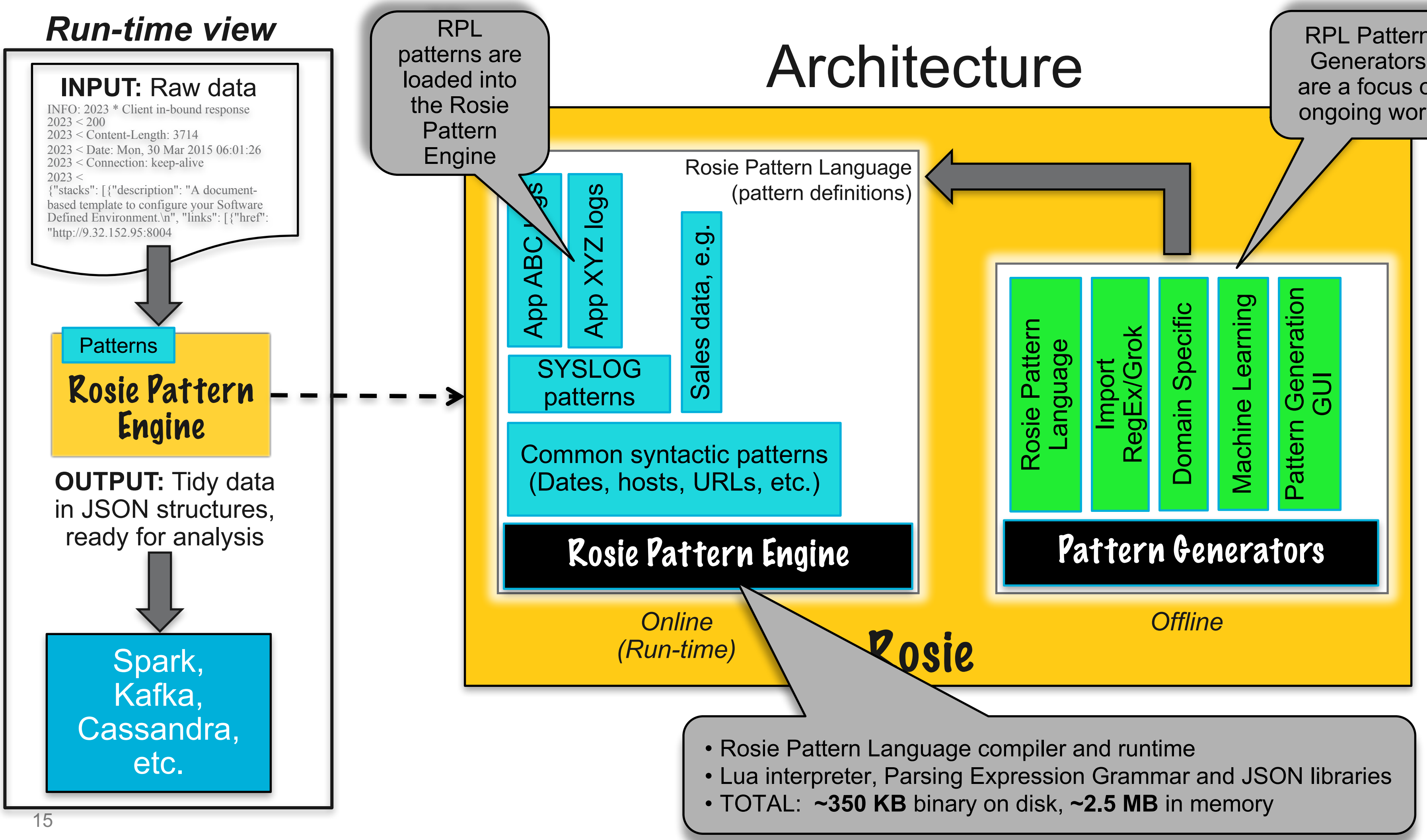
With closures!

Comments
Identifiers
Whitespace
Quoted literals
Macros
Modules (not shown)

[RPL Language Reference](#)
(Github)

```
---- -*- Mode: rpl; -*-
----
---- json.rpl    some rpl patterns for processing json input
----
---- © Copyright IBM Corporation 2016.

--
-- Match against 'json' to capture a json value
--

json.string = "\""  {"\\\"" / {! [\"] .}}* "\""

alias json.int = { [-]? {[1-9][0-9]+} / [0-9] }
alias json.frac = { [.] [0-9]+ }
alias json.exp = { [eE] [+-]? [0-9]+ }
json.number = { json.int json.frac? json.exp? }

json.true = "true"
json.false = "false"
json.null = "null"

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

json = json.value
```

Architecture



Demo

“I want to believe”

Fox Mulder, FBI

Highlights of Rosie Pattern Language (1)

Pattern to match

Input to match against

Read/eval/print loop prompt (for interactive pattern development)

Shorthand version of JSON output contains: Pattern name, position in input, matching text, and sub-matches

```
Rosie> .match "Hello", "Hello world"
[*:
 [pos: 1,
  text: "Hello",
  subs:
   []]]
Warning: unmatched characters at end of input
Rosie> h = "Hello"
Rosie> .match h "world", "Hello world"
[*:
 [pos: 1,
  text: "Hello world",
  subs:
   [1: [h:
         [pos: 1,
          text: "Hello",
          subs:
           []]]]]]
Rosie> _
```

Patterns are written like programs

- To match a literal string, put it in quotes
- Otherwise, it's an identifier
- Identifiers are defined using assignment statements
- When an identifier used within a pattern is matched, it appears as a sub-match in the output

<u>Notes</u>

1. Patterns entered at the command line do not have names, so they are represented by a "*" in the output in place of a name
2. A named pattern, such as "h" in this example, becomes a sub-match
3. A pattern is allowed to match a prefix of the input text

Highlights of Rosie Pattern Language (2)

```
Rosie> d
alias d = [:digit:]
Rosie> .match d+, "996"
[*:
 [pos: 1,
  text: "996",
  subs:
   []]]
Rosie> .match d{2,3}, "996"
[*:
 [pos: 1,
  text: "996",
  subs:
   []]]
Rosie> .match (d{2,3})+, "996 901 54"
[*:
 [pos: 1,
  text: "996 901 54",
  subs:
   []]]
Rosie> .match {"9" d d} / .*, "996 901 54"
[*:
 [pos: 1,
  text: "996",
  subs:
   []]]
Warning: unmatched characters at end of input
Rosie>
```

Ask Rosie what is the definition of the identifier “d”

This pattern matches a sequence of exactly 2 or 3 digits

By adding a “+”, we can match one or more of the 2-or-3-digit sequences

This pattern says: Match a “9” followed by 2 more digits, OR (if that fails) match everything

RPL Patterns share a lot with regular expressions

- $. * ? + and bounded repetition, for example
- Character sets such as [:alpha:] and [A-F]

Some differences are:

- The choice operator is “/” and is *ordered choice*
- Parentheses are for grouping only
- Tokenization is automatic, but is disabled for expressions inside curly braces {…}

(And in other places where tokenizing would be the wrong thing to do, e.g. quantified expressions like **d+**. Generally, Rosie tries to “do the right thing”.)

Notes

1. There are hundreds of patterns in the RPL library
2. The RPL tokenizer behaves much like the word boundary operator in regex, where it must be explicitly written as \b
3. The parentheses in **(d{2,3})+** are needed for proper tokenization
4. Without curly braces, the pattern **“9” d d** will match a 9 followed by two more digits as separate tokens

Highlights of Rosie Pattern Language (3)

RPL Patterns are typically saved in files

- The Rosie Pattern Engine reads (and compiles) RPL files
- There are hundreds of useful patterns available in RPL packages, including:
    - Timestamps in various formats
    - Network addresses, paths
    - Various log file formats
    - Numbers, identifiers, etc.

An excerpt from `rpl/network.rpl` in the Rosie distribution:

```
-- Some basic HTTP patterns

http_command_name = "GET" / "HEAD" / "PUT" / "POST" / "DELETE" /
                    "TRACE" / "OPTIONS" / "CONNECT" / "PATCH"

network.http_command = http_command_name (network.url / common.path)

network.http_version = {"HTTP" "/" digit+ "." digit+}     -- e.g. "HTTP/1.1"
```

Comments start with a double dash. Note also the ability to use whitespace for readability

```
Rosie> .match basic.datetime_patterns, "Feb 22, 2016"
[basic.datetime_patterns:
 [pos: 1,
  text: "Feb 22, 2016",
  subs:
   [1: [datetime.simple_date:
         [pos: 1,
          text: "Feb 22, 2016",
          subs:
           []]]]]]
Rosie> .match basic.datetime_patterns, "2016-02-25T03:40:11Z"
[basic.datetime_patterns:
 [pos: 1,
  text: "2016-02-25T03:40:11Z",
  subs:
   [1: [datetime.datetime_RFC3339:
         [pos: 1,
          text: "2016-02-25T03:40:11Z",
          subs:
           [1: [datetime.full_date_RFC3339:
                 [pos: 1,
                  text: "2016-02-25",
                  subs:
                   []]],
            2: [datetime.full_time_RFC3339:
                 [pos: 12,
                  text: "03:40:11Z",
                  subs:
                   []]]]]]]]]
```

Note: Rosie does parse out the month, day, year, etc. separately. Those sub-matches are not shown here for clarity.

Highlights (4): CLI

The command-line interface to the Rosie Pattern Engine reads pattern definitions from RPL files, and matches against input from files

```
jamiejennings: ./run -patterns | grep network
This is Rosie v0.88
basic.network_patterns      definition      red
network.email_address       definition      red
network.fqdn                definition      red
network.host                definition
network.http_command        definition      red
network.http_version        definition      red
network.ip_address          definition      red
network.protocol            definition
network.url                 definition      red
jamiejennings:
```

The "patterns" option lists the patterns loaded, and the color in which matches will appear

```
jamiejennings: cat /etc/resolv.conf
#
# Mac OS X Notice
#
# This file is not used by the host name and address resolution
# or the DNS query routing mechanisms used by most processes on
# this Mac OS X system.
#
# This file is automatically generated.
#
domain raleigh.ibm.com
nameserver 9.0.128.50
nameserver 9.0.130.50
```

Any text can be used as input

```
jamiejennings: ./run 'common.word basic.network_patterns' /etc/resolv.conf
domain raleigh.ibm.com
nameserver 9.0.128.50
nameserver 9.0.130.50
jamiejennings:
```

This pattern finds lines that start with a word followed by a network address

```
jamiejennings: ./run basic.matchall /etc/resolv.conf
#
# Mac OS X Notice
#
# This file is not used by the host name and address resolution
# or the DNS query routing mechanisms used by most processes on
# this Mac OS X system .
#
# This file is automatically generated .
#
domain raleigh.ibm.com
nameserver 9.0.128.50
nameserver 9.0.130.50
jamiejennings:
```

`Basic.matchall` is pattern that looks for a few dozen common patterns, anywhere in the input

Notes

1. There are hundreds of patterns in the RPL library
2. The single quotes on the command line prevent the shell from interpreting characters (such as dot) in the RPL pattern
3. Rosie Pattern Engine generates JSON. The JSON is converted to just the matching text and printed in color because this is easier to read in a terminal window
4. In this example:
   - Punctuation prints in black
   - Words print in yellow, and likely identifiers in cyan
   - Network addresses print in red
   - Numbers, including hex, print as underlined

Highlights (5): Interactive Pattern Development

```
jamiejennings: ./run -repl
Rosie> common.number
common.number = (common.denoted_hex / (common.float / (common.hex / common.int)))
Rosie> common.denoted_hex
common.denoted_hex = {"0x" common.hex}
Rosie> .match common.number, "0x3C"
[common.number:
 [text: "0x3C",
  subs:
   [1: [common.denoted_hex:
         [text: "0x3C",
          subs:
           [1: [common.hex:
                 [text: "3C",
                  subs:
                   [],
                  pos: 3]]],
          pos: 1]]],
  pos: 1]]
Rosie> .eval common.number, "0x3C"
    CHOICE: (common.denoted_hex / (common.float / (common.hex / common.int)))
    Matched "0x3C" (against input "0x3C")
    Explanation:
       IDENTIFIER: common.denoted_hex
       Matched "0x3C" (against input "0x3C")
       Explanation (identifier's definition): {"0x" common.hex}
          GROUP: {"0x" common.hex}
          Matched "0x3C" (against input "0x3C")
          Explanation:
             SEQUENCE: "0x" common.hex
             Matched "0x3C" (against input "0x3C")
             Explanation:
 1.............LITERAL STRING: "0x"
                Matched "0x" (against input "0x3C")
                IDENTIFIER: common.hex
                Matched "3C" (against input "3C")
                Explanation (identifier's definition): hex_digits+
 2...............QUANTIFIED EXP (raw): hex_digits+
                   Matched "3C" (against input "3C")
 3.....BOUNDARY
       Matched "" (against input "")
[common.number:
 [text: "0x3C",
  subs:
   [1: [common.denoted_hex:
         [text: "0x3C",
          subs:
           [1: [common.hex:
                 [text: "3C",
                  subs:
                   [],
                  pos: 3]]],
          pos: 1]]],
  pos: 1]]
Rosie> _
```

A pattern name evaluates to its definition, which Rosie then displays

The input "0x3C" matches `common.number`, generating a match structure

The ".eval" command takes the same arguments as ".match" and prints a trace (highlighted at left) of the matching process

The Rosie Pattern Engine has a read/eval/print loop that can be used to develop and test patterns. Existing patterns are available, and new patterns can be defined. A detailed trace explains how a pattern matches (or fails) against sample input.

Notes

1. The ".eval" command always produces a trace, whether the match succeeds or fails.
2. The ".match" command by default prints a trace when a match fails.
3. The effect of automatic tokenization is shown explicitly in the trace output, where Rosie shows the step of matching BOUNDARY (the inter-token boundary).
4. In this example, Rosie looks for BOUNDARY only after the `common.number` is matched, and the end of the input successfully matches BOUNDARY.

RPL for root cause analysis

RPL code written to parse Apache Spark logs

```
-----
---- spark.rpl       patterns for Apache Spark logs
----
---- (c) 2016, Jamie A. Jennings
-----

spark.filename = {[:alnum:]/[_%!$@.,~-]}+

spark.command = "Spark Command:" .*
spark.using_message = "Using" .*

spark.ignore = "="* $

spark.message = .*
spark.typical = datetime.simple_slash_date datetime.simple_time common.word common.identifier_plus_plus ":" spark.message

spark.py_identifier = {![:space:] .}*

spark.driver_stacktrace = "Driver stacktrace:"
spark.caused_by = "Caused by:" common.dotted_identifier
spark.and_more = [:space:]* "..." common.int "more"

spark.py_traceback_start = "Traceback" .*
spark.py_traceback_file = [:space:]* "File" {"\"" common.path "\", line"} common.int ", in" spark.py_identifier
spark.py_line = " "{4,} spark.message
spark.py_java_exception_start = ":" common.dotted_identifier ":" {!{common.dotted_identifier $} .}* common.dotted_identifier $

spark.java_exception_start = common.dotted_identifier
alias spark.fn_or_native = (spark.filename ":" common.int) / "Native Method"
spark.exception_start = common.dotted_identifier ":" spark.message
spark.exception_at = [:space:]* "at" { common.dotted_identifier "(" spark.fn_or_native ")" }

spark.patterns = spark.typical /
                 spark.py_traceback_file /
                 spark.exception_at /
                 spark.exception_start /
                 spark.java_exception_start /
                 spark.py_java_exception_start /
                 spark.driver_stacktrace /
                 spark.caused_by /
                 spark.py_traceback_start /
                 spark.py_line /
                 spark.py_traceback_file /
                 spark.py_line /
                 spark.and_more /
                 spark.command /
                 spark.using_message /
                 spark.ignore
```

Notes
1. The basic.matchall pattern can be used to quickly see what Rosie can already recognize in an input file
2. Then, more complex patterns can be assembled interactively using existing patterns
3. Here, the input files are Apache Spark logs
4. The logs contain a mix of Python and Java information

Output generated using this RPL code

```
jamiejennings: ./run spark.patterns ~/Data/spark-log3.log | head -5
16/02/08 10:14:33  INFO SparkContext Running Spark version 1.6.0
16/02/08 10:14:33  WARN NativeCodeLoader Unable to load native-hadoop library for your platform... using builtin-java classes where applicabl
e
16/02/08 10:14:33  INFO SecurityManager Changing view acls to: al
16/02/08 10:14:33  INFO SecurityManager Changing modify acls to: al
16/02/08 10:14:33  INFO SecurityManager SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(al); use
rs with modify permissions: Set(al)
jamiejennings: ./run -json spark.patterns ~/Data/spark-log3.log | head -1
{"spark.patterns":{"pos":1,"text":"16\/02\/08 10:14:33 INFO SparkContext: Running Spark version 1.6.0","subs":[{"spark.typical":{"pos":1,"tex
t":"16\/02\/08 10:14:33 INFO SparkContext: Running Spark version 1.6.0","subs":[{"datetime.simple_slash_date":{"pos":1,"text":"16\/02\/08","s
ubs":{}}},{"datetime.simple_time":{"pos":10,"text":"10:14:33 ","subs":{}}},{"common.word":{"pos":19,"text":"INFO","subs":{}}},{"common.identi
fier_plus_plus":{"pos":24,"text":"SparkContext","subs":{}}},{"spark.message":{"pos":38,"text":"Running Spark version 1.6.0","subs":{}}}]}}]}}
jamiejennings:
```

Implementation

“Simplicity does not precede complexity, but follows it.”

Alan Perlis

RPL is a language of parser combinators

```
Rosie> network.http_command
network.http_command = http_command_name (network.url / common.path)
Rosie> .match network.http_command, "GET http://www.ibm.com/index.html"
{"network.http_command":
   {"text": "GET http://www.ibm.com/index.h...",
    "pos": 1.0,
    "subs":
      [{"http_command_name":
         {"text": "GET",
          "pos": 1.0,
          "subs": []}},
       {"network.url":
         {"text": "http://www.ibm.com/index.html",
          "pos": 5.0,
          "subs":
            [{"common.path":
               {"text": "/index.html",
                "pos": 23.0,
                "subs": []}}]}}]}}
Rosie>
```

Parser combinators are

- Recursive descent parsers
- Based on higher order functions
- Considered easy to read
- Often used to parse CFLs

Rosie Pattern Language

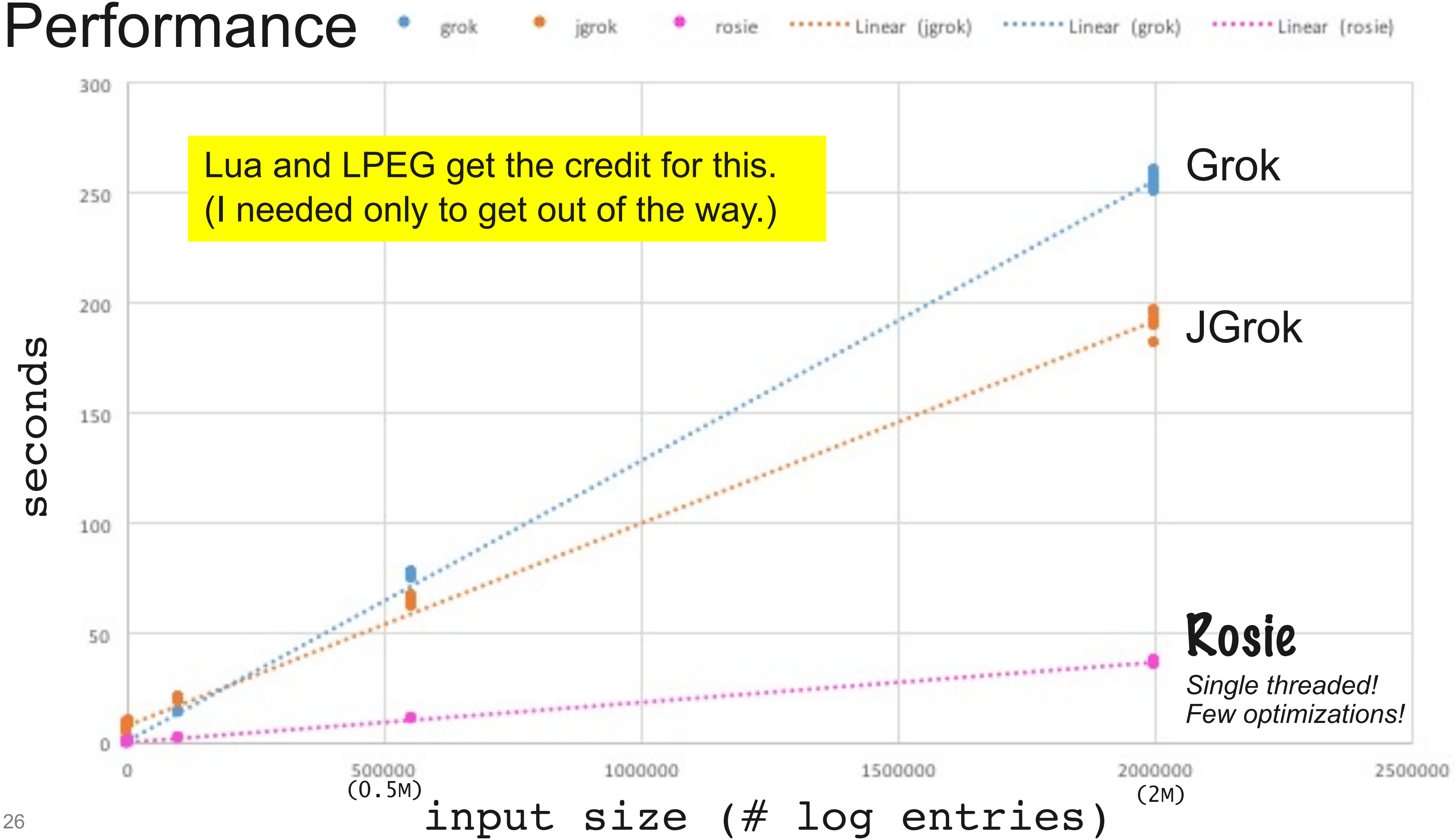
- Recognizes deterministic CFLs
- Combinators are:
  - Sequence
  - Ordered choice
  - Quantified expressions
  - Predicates: look ahead, look behind, negation
- Tokenized (“cooked”) and untokenized (“raw”) expressions

Patterns in the RPL library (at present)

- Basic
  - number, identifier, word, and more
  - and quoted/bracketed versions
- Commonly used and specific
  - int, float, hex, and other numbers
  - several kinds of identifiers
  - path names for Unix and Windows
  - GUIDs
- Network patterns
  - ip address, domain name, email address, http url and commands
- 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, or
  - match nested and balanced {} []
- Log files
  - syslog constituents (covers most log files)
  - Java exceptions, Python tracebacks
- Source code (micro-grammar approach)
  - Extract line and block comments
  - Extract code (no comments)
  - Python, Ruby, Perl, js, Java, Perl, C, C++, …

Performance



Other capabilities, current and forthcoming

Language

- Lexical scope (nested environments)
- Modules have their own environments with import/export controls (forthcoming)
- "Macros" (i.e. pattern generating functions)
  - Have Lua functions for AST → AST
  - Need more experimentation
- Post-processing instructions (forthcoming)
  - Match → Match
  - Lua as extension language
  - Uses include
    - Format conversion
    - Sanitizing and anonymizing
    - Meta-data collection

Implementation

- Self-hosting
  - Allows easy language modifications
  - A compiler extension interface would allow language extensions
- Interfaces: API, CLI, REPL
  - Native APIs in C and Lua
  - C API is auto-generated from Lua API
- Foreign function interface: librosie
  - Sample clients in
    Python, Perl, Ruby, js, Go, …
    Lua???
  - Grok replacement (for ELK stack)
- Output generator is a Lua function
- Persist compiled patterns to disk (forthcoming)
- More debugging capabilities (forthcoming)

Conclusion

- Rosie Pattern Language
  - Designed for parsing "in the large"
  - More expressive than regex
  - With in-line automated tokenization
  - And many features commonly found in programming languages
- Rosie Pattern Engine
  - Small (~ 350 KB on disk, ~ 2.5 MB memory) and relatively fast (around 4x competition)
  - With pattern development tools
    - REPL
    - Debugger

      "Eval" (interpreter) shows full match trace

      Future: breakpoints, single step, single identifier trace
  - Implemented in Lua, using LPEG
  - Released as open source in February, 2016

Exploring lpeg enhancements to support RPL pattern debugging

The End

"Turn out the lights, the party's over"

Willie Nelson, "The Party's Over"

Open Source Software, MIT License

Github (public) https://github.com/jamiejennings/rosie-pattern-language/

IBM developerWorks Open (tutorials, blog) https://developer.ibm.com/open/rosie-pattern-language/

Implementation details (v0.92b)

Component	Implementation language	Description	Location
"Sample" RPL patterns	Rosie Pattern Language (RPL)	100's of patterns: • Numbers, identifiers • Network, email addrs • Many dates & times • Syslog elements • Etc.	Public github MIT License https://github.com/jamiejennings/rosie-pattern-language/tree/master/rpl
Rosie REPL Rosie CLI Rosie Debugger	Lua	~ 600 lines of Lua code ~ 25 lines of RPL These leverage the API	Public github MIT License https://github.com/jamiejennings/rosie-pattern-language
Rosie API	Native: Lua, C Others: via libffi	~ 20 functions	
Rosie Compiler	Lua (parser in RPL, bootstrap in Lua/LPEG)	~ 1300 lines of Lua code ~ 60 lines of RPL	
LPEG CJSON	ANSI C	Lua PEG library ~ 46 Kb Lua JSON library ~ 54 Kb	Public web, MIT License http://www.inf.puc-rio.br/~roberto/lpeg/
Lua	ANSI C	Lua interpreter ~ 224 Kb	Public web, MIT License http://lua.org

rprint (awk-like processing of Rosie json output)

```
bash-3.2$ rosie -encode json -wholefile py.line_comments_only sklearn/utils/validation.py |
rprint 'for i=1,NF do print($i); end'
# Authors: Olivier Grisel
#          Gael Varoquaux
#          Andreas Mueller
#          Lars Buitinck
#          Alexandre Gramfort
#          Nicolas Tresegnie
# License: BSD 3 clause
# Silenced by default to reduce verbosity. Turn on at runtime for
# performance profiling.
# First try an O(n) time, O(1) space solution for the common case that
# everything is finite; fall back to O(n) space np.isfinite to prevent
# false positives from overflow in sum method.
# is numpy array
# Don't get num_samples from an ensembles length!
# force an upcast to `long` under Python 2
# special notation for singleton tuples
# create new with correct sparse # convert dtype # force copy
# store whether originally we wanted numeric dtype
# not a data type (e.g. a column named dtype in a pandas DataFrame)
# if input is object, convert to float.
# no dtype conversion required # dtype conversion required. Let's select the first element of the
# list of accepted types.
# To ensure that array flags are maintained
# make sure we actually converted to numeric:
# only csr, csc, and coo have `data` attribute # FIXME NotFittedError_ --> NotFittedError in 0.19
bash-3.2$
```

Rosie Pattern Engine API

- Engine management
  - New engine
  - Configure engine
  - Delete engine
  - Query engine configuration
  - Query engine environment
  - Future: Set logging level
- Environment (per engine)
  - Load string (RPL definitions)
  - Load file (RPL definitions)
  - Load manifest (files of RPL definitions)
  - Erase environment
- Matching (per engine)
  - Match against string
  - Match against file
- Debugging (per engine)
  - Eval against input string (full trace)
  - Eval against input file (full trace)
  - Future:
    - Trace single identifier (combinator)
    - Breakpoint

Rosie is self-hosting

- Rosie is a parser, and Rosie is used to parse Rosie Pattern Language
- About 60 lines of RPL (core) to define the current RPL (v0.99)
- Capabilities (e.g. syntax error reporting) made for RPL itself can be applied to user patterns, and vice-versa (e.g. macros)
- Ability to support multiple versions of RPL, even different dialects
- Non-trivial user extensions to RPL can be had 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

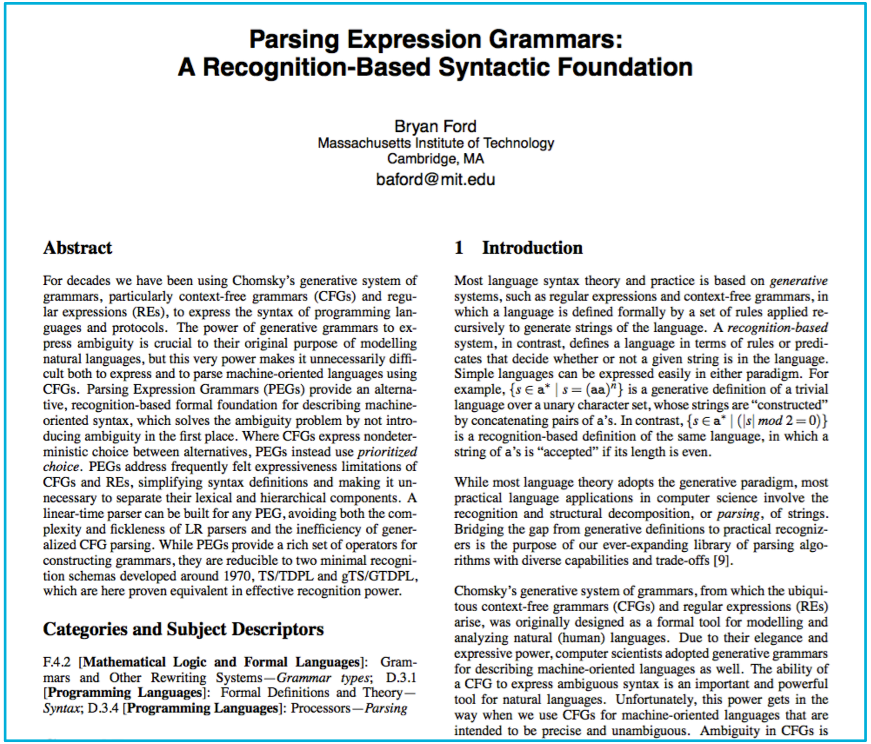
Tokenization is non-trivial

RPL	Meaning
`a a`	`a~a`
`(a a)`	`a~a`
`{a a}`	`aa`
`a+`	`aaaa...a`
`a+ b`	`aaaa...a~b`
`(a)+`	`a~a~a~a~...~a`
`(a)+ b`	`a~a~a~a~...~a~b`
`(a / b)`	`a` `b`
`(a / b) c`	`a~c` `b~c`
`{{a / b} c}`	`ac` `bc`
`{(a / b) c}`	*???* → `Same as {{a / b} c}`
`(a b)+`	`a~b~a~b~...a~b`
`{a b}+`	`ababab...ab`
`(a b)+ c`	`a~b~a~b~...~c`
`{a b}+ c`	`abab...ab~c`

- Token boundary
  - Token boundary is denoted "~"
  - Has a default value (approx. \b)
  - Default is idempotent
  - Is redefinable!
  - User's definition may not be idempotent
- Requires careful implementation
- E.g. implementation of (p)* in Lua/lpeg:
  peg = (p * (~ * p)^0)^-1

Parsing Expression Grammars

- Rosie's operators
  - Parsing Expression Grammars
  - Instead of CFG or regex
  - Express all deterministic CFLs
  - And some non-CFLs, e.g. $a^n b^n c^n$
- PEGs [Ford, 2004]
  - Scanner-less parsing
  - Compare to regular expressions
    - Greedy quantifiers: *, +, ?
    - Ordered choice operator: /
    - Predicates: "looking at", "not looking at"
  - Linear time algorithms
  - Languages recognized by PEGs are
    - A superset of regular languages
    - All languages recognized by LL(k) and LR(k) parsers



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.

Categories and Subject Descriptors

F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—*Grammar types*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax*; D.3.4 [**Programming Languages**]: Processors—*Parsing*

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 ubiquitous context-free grammars (CFGs) and regular expressions (REs) arise, was originally designed as a formal tool for modelling and analyzing natural (human) languages. Due to their elegance and expressive power, computer scientists adopted generative grammars for describing machine-oriented languages as well. The ability of a CFG to express ambiguous syntax is an important and powerful tool for natural languages. Unfortunately, this power gets in the way when we use CFGs for machine-oriented languages that are intended to be precise and unambiguous. Ambiguity in CFGs is

Infinite loop in Perl RE?

- Claimed on stack exchange that this regex never terminates?
  - See ‘man perlre’
  - 'foo' =~ m{ (o?)* }x;
  - “Perl has special code to detect infinite recursion in this case and break out.”
  - Alex Brown Dec 7 '10 at 16:09
- http://stackoverflow.com/questions/4378455/what-is-the-complexity-of-regular-expression