LPEG: a new approach to pattern matching in Lua

Roberto Ierusalimschy
(real) regular expressions

- inspiration for most pattern-matching tools
  - Ken Thompson, 1968
- very efficient implementation
- too limited
  - weak in what can be expressed
  - weak in how to express them
(real) regular expressions

- "problems" with non-regular languages
- problems with complement
  - C comments
  - C identifiers
- problems with captures
  - intrinsic non determinism
  - "longest-matching" rule makes concatenation non associative
Longest-Matching Rule

- breaks $O(n)$ time when searching
- breaks associativity of concatenation

\[
((a \mid ab) (cd \mid bcde)) \ e? \ \odot \ "abcde"
\rightarrow "a" - "bcde" - ""
\]

\[
(a \mid ab) ((cd \mid bcde) \ e?) \ \odot \ "abcde"
\rightarrow "ab" - "cd" - "e"
\]
"regular expressions"

- set of ad-hoc operators
  - possessive repetitions, lazy repetitions, look ahead, look behind, back references, etc.
- no clear and formally-defined semantics
- no clear and formally-defined performance model
  - ad-hoc optimizations
- still limited for several useful tasks
  - parenthesized expressions
"regular expressions"

- unpredictable performance
- hidden backtracking

```
(.*),(.*),(.*),(.*),(.*)[.;] ✗
"a,word,and,other,word;"

(.*),(.*),(.*),(.*),(.*)[.;] ✗
",,,,,,,,,,,,,,,,,,,,,,,,,,;"
```
PEG: Parsing Expression Grammars

- not totally unlike context-free grammars
- emphasis on string recognition
  - not on string generation
- incorporate useful constructs from pattern-matching systems
  - a*, a?, a+
- key concepts: ordered choice, restricted backtracking, and predicates
Short history

• restricted backtracking and the not predicate first proposed by Alexander Birman, ~1970

• later described by Aho & Ullman as TDPL (Top Down Parsing Languages) and GTDPL (general TDLP)
Short history

• revamped by Bryan Ford, MIT, in 2002
  • pattern-matching sugar
  • Packrat implementation
• main goal: unification of scanning and parsing
  • emphasis on parsing
PEG in PEG

grammar <- (nonterminal '<-' sp pattern)+
pattern <- alternative ('/' sp alternative)*
alternative <- ([!&]? sp suffix)+
suffix <- primary ([*+?] sp)*
primary <- (' sp pattern ') sp
     / .' sp / literal / charclass
     / nonterminal !'<'-
literal <- ['] (!['] .)* ['] sp
charclass <- '[' (!']' (. '-' . / .)))* ']' sp
nonterminal <- [a-zA-Z]+ sp
sp <- [ \t\n]*
PEGs basics

A <- B C D / E F / ...

- to match A, match B followed by C followed by D
- if any of these matches fails, try E followed by F
- if all options fail, A fails
Ordered Choice

A <- A₁ / A₂ / ...

• to match A, try first A₁
• if it fails, backtrack and try A₂
• repeat until a match
Restricted Backtracking

\[
\begin{align*}
S & \leftarrow A \ B \\
A & \leftarrow A_1 / A_2 / \ldots
\end{align*}
\]

- once an alternative $A_1$ matches for $A$, no more backtrack for this rule
- even if $B$ fails!
Example: greedy repetition

- ordered choice makes repetition greedy
- restricted backtracking makes it blind
- matches maximum span of $A$s
- possessive repetition

\[
S \leftarrow A^* \quad \Rightarrow \quad S \leftarrow A \ S \ / \ \epsilon
\]
Non-blind greedy repetition

\[ S \leftarrow A S / B \]

• ordered choice makes repetition greedy
• whole pattern only succeeds with \( B \) at the end
• if ending \( B \) fails, previous \( A S \) fails too
  • engine backtracks until a match
  • conventional greedy repetition
Non-blind greedy repetition: Example

- find the last comma in a subject

\[ s \gets .s / ',,' \]
Non-blind non-greedy repetition

- ordered choice makes repetition lazy
- matches minimum number of $A$s until a $B$
  - lazy (or reluctant) repetition

\[
S \leftarrow B / A S
\]

```
comment <- '/*' end_comment
end_comment <- '*/' / . end_comment
```
Predicates

- check for a match without consuming input
  - allows arbitrary look ahead
- \(!A\) (not predicate) only succeeds if \(A\) fails
  - either \(A\) or \(!A\) fails, so no input is consumed
- \&A\) (and predicate) is sugar for \(!!A\)
Predicates: Examples

\[
\text{EOS} \leftarrow !. \\
\text{comment} \leftarrow '/*' (!'*/' .)* '*/'
\]

- next grammar matches \(a^n b^n c^n\)
- a non context-free language

\[
\begin{align*}
\text{S} & \leftarrow \& \text{P1} \text{ P2} \\
\text{P1} & \leftarrow \text{AB} 'c' \\
\text{AB} & \leftarrow 'a' \text{ AB} 'b' / \varepsilon \\
\text{P2} & \leftarrow 'a' * \text{ BC} !. \\
\text{BC} & \leftarrow 'b' \text{ BC} 'c' / \varepsilon
\end{align*}
\]
Right-linear grammars

- for right-linear grammars, PEGs behave exactly like CFGs
- it is easy to translate a finite automata into a PEG

```
EE  <-  '0' OE / '1' EO / !.
OE  <-  '0' EE / '1' OO
EO  <-  '0' OO / '1' EE
OO  <-  '0' EO / '1' OE
```
LPEG: PEG for Lua

- a small library for pattern matching based on PEGs
- emphasis on pattern matching
  - but with full PEG power
LPEG: PEG for Lua

- SNOBOL tradition: language constructors to build patterns
  - verbose, but clear

```lua
lower = lpeg.R("az")
upper = lpeg.R("AZ")
letter = lower + upper
digit = lpeg.R("09")
alphanum = letter + digit + "_"
```
## LPEG basic constructs

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lpeg.R(&quot;xy&quot;)</code></td>
<td>-- range</td>
</tr>
<tr>
<td><code>lpeg.S(&quot;xyz&quot;)</code></td>
<td>-- set</td>
</tr>
<tr>
<td><code>lpeg.P(&quot;name&quot;)</code></td>
<td>-- literal</td>
</tr>
<tr>
<td><code>lpeg.P(number)</code></td>
<td>-- that many characters</td>
</tr>
<tr>
<td><code>P1 + P2</code></td>
<td>-- ordered choice</td>
</tr>
<tr>
<td><code>P1 * P2</code></td>
<td>-- concatenation</td>
</tr>
<tr>
<td><code>-P</code></td>
<td>-- not P</td>
</tr>
<tr>
<td><code>P1 - P2</code></td>
<td>-- P1 if not P2</td>
</tr>
<tr>
<td><code>P^n</code></td>
<td>-- at least n repetitions</td>
</tr>
<tr>
<td><code>P^-n</code></td>
<td>-- at most n repetitions</td>
</tr>
</tbody>
</table>
LPEG basic constructs: Examples

```lua
reserved = (lpeg.P"int" + "for" + "double" + "while" + "if" + ...) * -alphanum

identifier = ((letter + "_") * alphanum^0) - reserved

print(identifier:match("foreach"))  --> 8
print(identifier:match("for"))      --> nil
```
"regular expressions" for LPEG

• module `re` offers a more conventional syntax for patterns
• similar to "conventional" regexs, but literals must be quoted
  • avoid problems with magic characters

```
print(re.match("for", "[a-z]*"))  -->  4

s = "/** a comment**/ plus something"
print(re.match(s, "'/*/ {(!'*/' .)*} '*/'"))
  -->  * a comment*
```
"regular expressions" for LPEG

• patterns may be precompiled:

```python
s = "/** a comment**/ plus something"
comment = re.compile("'/** {(!'*/' .)*} */'"
print(comment.match(s))  -->  * a comment*
```
LPEG grammars

- described by tables
  - `lpeg.V` creates a non terminal

```plaintext
S, V = lpeg.S, lpeg.V
number = lpeg.R"09"^1

exp = lpeg.P{"Exp",
  Exp = V"Factor" * (S"+-" * V"Factor")^0,
  Factor = V"Term" * (S"*/" * V"Term")^0,
  Term = number + "(" * V"Exp" * ")"
}
```
LPEG grammars with 're'

```python
exp = re.compile[['
    Exp  <-  <Factor>  ([+-]  <Factor>)*
    Factor  <-  <Term>  ([*/]  <Term>)*
    Term  <-  [0-9]+  /  '(%  <Exp>  )'
']]
```
**Search**

- unlike most pattern-matching tools, LPEG has no implicit search
  - works only in *anchored mode*
- search is easily expressed within the pattern:

\[
(1 - P)^0 \ast P
\]

\[
\{ P + 1 \ast \text{lpeg.V}(1) \}
\]

\[
(!P \ast .)\ast P
\]

\[
S \leftarrow P / . \langle S \rangle
\]
Captures

- patterns that create values based on matches
  - `lpeg.C(patt)` - captures the match
  - `lpeg.P(patt)` - captures the current position
  - `lpeg.Cc(values)` - captures 'value'
  - `lpeg.Ct(patt)` - creates a list with the nested captures
  - `lpeg.Ca(patt)` - "accumulates" the nested captures
Captures in 're'

- reserves parentheses for grouping
  - `{patt}` - captures the match
  - `{}` - captures the current position
  - `patt -> {}` - creates a list with the nested captures
Captures: examples

- Each capture match produces a new value:

```python
list = re.compile(r'\{%w*\} (',', ' {\%w*\})*)

print(list.match("a,b,c,d"))  -->  a  b  c  d
```
Captures: examples

```python
list = re.compile"\{\}\w* (',', \{\}\w*)*"

print(list.match("a,b,c,d"))  -->  1 3 5 7
```
Captures: examples

```python
list = re.compile("({}%w* (',') {}%w*)* \rightarrow {}"")
t = list:match"a,b,c,d")
-- t is {1,3,5,7}
```
Captures: examples

```python
exp = re.compile(['
    S <- <atom> / '( %s* <S>* -> {} ' )' %s* 
    atom <- { [a-zA-Z0-9]+ } %s* 
  ])

for t in exp.match('(a b (c d) ())'):
    print(t)
```

```python
-- t is {'a', 'b', {'c', 'd'}, {}}
```
function split (s, sep)
    sep = lpeg.P(sep)
    local elem = lpeg.C((1 - sep)^0)
    local p = elem * (sep * elem)^0
    return lpeg.match(p, s)
end

split("a,b,,,,", ",," ) --> "a", "b", ",", ","
function split (s, sep)
  sep = lpeg.P(sep)
  local elem = lpeg.C((1 - sep)^0)
  local p = lpeg.Ct(elem * (sep * elem)^0)
  return lpeg.match(p, s)
end

split("a,b,,", ",")  --> {"a", "b", ",", ","}
Substitutions

- No special function; done with captures
  - `lpeg.Cs(patt)` - captures the match, with nested captures replaced by their values
  - `patt / string` - captures 'string', with marks replaced by nested captures
  - `patt / table` - captures 'table[match]'
  - `patt / function` - applies 'function' to match
digits = lpeg.C(lpeg.R"09"^1)
letter = lpeg.C(lpeg.R"az")
Esc = lpeg.P"\\"

Char = (1 - Esc)
  + Esc * digits / string.char
  + Esc * letter / { n = "\n", t = "\t",
                      ...
                    }

p = lpeg.Cs(Char^0)
p:match([[\n\97b]])  --> "\nab"
Substitutions in 're'

- Denoted by \{~ \ldots ~\}

\[
P = \{\sim (\'0\' \rightarrow \'1\' / \'1\' \rightarrow \'0\' / . )\,* \sim\}\n\]

print(re.match("1101 0110", P))  -->  0010 1001
Substitutions in 're'

CVS ← (<record> (%nl <record>)* ) → {}
record ← (<field> (',,' <field>)* ) → {}
field ← "" <escaped> '"' / <simple>
simple ← { [^,"%nl]* }
escaped ← {~ ([^"] / "'"->"'" )* ~}
Implementation

• Any PEG can be recognized in linear time
  • but constant is too high
  • space is also linear!
• LPEG uses a *parsing machine* for matching
  • each pattern represented as code for the PM
  • backtracking may be exponential for some patterns
  • but has a clear performance model
• quite efficient for "usual" patterns
Parsing Machine code

'ana'

00: char 'a' (61)
01: char 'n' (6e)
02: char 'a' (61)
03: end
Parsing Machine code

'ana' / .

00: choice -> 5
01: char 'a' (61)
02: char 'n' (6e)
03: char 'a' (61)
04: commit -> 6
05: any * 1
06: end
'ana' / .

00: testchar 'a' (61) -> 5
01: choice -> 5 (1)
02: char 'n' (6e)
03: char 'a' (61)
04: commit -> 6
05: any * 1
06: end
'hi' / 'foo'

00: testchar 'h' (68) -> 3
01: char 'i' (69)
02: jmp -> 6
03: char 'f' (66)
04: char 'o' (6f)
05: char 'o' (6f)
06: end
Parsing Machine: Grammars

S <- 'ana' / . <S>

00: call -> 2
01: jmp -> 10
02: testchar 'a' (61) -> 7
03: choice -> 7 (1)
04: char 'n' (6e)
05: char 'a' (61)
06: commit -> 9
07: any * 1
08: jmp -> 2
09: ret
10: end
Parsing Machine: Right-linear Grammars

```
EE  <- '0' <OE> / '1' <EO> / !.
OE  <- '0' <EE> / '1' <OO>
EO  <- '0' <OO> / '1' <EE>
OO  <- '0' <EO> / '1' <OE>
```

```plaintext
... 19: testchar '0'-> 22
20: jmp -> 2
21: jmp -> 24
22: char '1'
23: jmp -> 8
24: ret
...```
Benchmarks

Apple × Orange
Benchmarks: Search

- programmed in LPEG:
  - $S \leftarrow '@the' / . <S>$
  - $(!'@the' .)* '@the'
- these searches are not expressible in Posix and Lua; crashes PCRE
- built-in in PCRE, Posix, and Lua
time (milisecond) for searching a string in the Bible

<table>
<thead>
<tr>
<th>pattern</th>
<th>PCRE</th>
<th>POSIX regex</th>
<th>Lua</th>
<th>LPEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>'@the'</td>
<td>5.3</td>
<td>14</td>
<td>3.6</td>
<td>40</td>
</tr>
<tr>
<td>'Omega'</td>
<td>6.0</td>
<td>14</td>
<td>3.5</td>
<td>40</td>
</tr>
<tr>
<td>'Alpha'</td>
<td>6.7</td>
<td>15</td>
<td>4.2</td>
<td>40</td>
</tr>
<tr>
<td>'amethysta'</td>
<td>27</td>
<td>38</td>
<td>24</td>
<td>47</td>
</tr>
<tr>
<td>'heith'</td>
<td>32</td>
<td>44</td>
<td>26</td>
<td>50</td>
</tr>
<tr>
<td>'eartt'</td>
<td>40</td>
<td>53</td>
<td>36</td>
<td>52</td>
</tr>
</tbody>
</table>
time (milisecond) for searching a string in the Bible

<table>
<thead>
<tr>
<th>pattern</th>
<th>PCRE</th>
<th>POSIX regex</th>
<th>Lua</th>
<th>LPEG</th>
<th>false starts</th>
</tr>
</thead>
<tbody>
<tr>
<td>'@the'</td>
<td>5.3</td>
<td>14</td>
<td>3.6</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>'Omega'</td>
<td>6.0</td>
<td>14</td>
<td>3.5</td>
<td>40</td>
<td>8853</td>
</tr>
<tr>
<td>'Alpha'</td>
<td>6.7</td>
<td>15</td>
<td>4.2</td>
<td>40</td>
<td>17,851</td>
</tr>
<tr>
<td>'amethysta'</td>
<td>27</td>
<td>38</td>
<td>24</td>
<td>47</td>
<td>256,897</td>
</tr>
<tr>
<td>'heith'</td>
<td>32</td>
<td>44</td>
<td>26</td>
<td>50</td>
<td>278,986</td>
</tr>
<tr>
<td>'eartt'</td>
<td>40</td>
<td>53</td>
<td>36</td>
<td>52</td>
<td>407,883</td>
</tr>
</tbody>
</table>
• because they are programmed in LPEG, we can optimize them:
  
  \[
  \text{S} \leftarrow ' @\text{the}' / . \ <S>
  \]
  
  \[
  \text{S} \leftarrow ' @\text{the}' / . \ [^@]* \ <S>
  \]
time (milisecond) for searching a string in the Bible

<table>
<thead>
<tr>
<th>pattern</th>
<th>PCRE</th>
<th>POSIX regex</th>
<th>Lua</th>
<th>LPEG</th>
<th>LPEG (2)</th>
<th>false starts</th>
</tr>
</thead>
<tbody>
<tr>
<td>'@the'</td>
<td>5.3</td>
<td>14</td>
<td>3.6</td>
<td>40</td>
<td>9.9</td>
<td>0</td>
</tr>
<tr>
<td>'Omega'</td>
<td>6.0</td>
<td>14</td>
<td>3.5</td>
<td>40</td>
<td>10</td>
<td>8853</td>
</tr>
<tr>
<td>'Alpha'</td>
<td>6.7</td>
<td>15</td>
<td>4.2</td>
<td>40</td>
<td>11</td>
<td>17,851</td>
</tr>
<tr>
<td>'amethysta'</td>
<td>27</td>
<td>38</td>
<td>24</td>
<td>47</td>
<td>21</td>
<td>256,897</td>
</tr>
<tr>
<td>'heith'</td>
<td>32</td>
<td>44</td>
<td>26</td>
<td>50</td>
<td>23</td>
<td>278,986</td>
</tr>
<tr>
<td>'eartt'</td>
<td>40</td>
<td>53</td>
<td>36</td>
<td>52</td>
<td>26</td>
<td>407,883</td>
</tr>
</tbody>
</table>
time (milisecond) for searching a pattern in the Bible

<table>
<thead>
<tr>
<th>pattern</th>
<th>PCRE</th>
<th>POSIX regex</th>
<th>Lua</th>
<th>LPEG</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[a-zA-Z]{14,}</code></td>
<td>10</td>
<td>15</td>
<td>16</td>
<td>4.0</td>
</tr>
<tr>
<td>([a-zA-Z]+) 'Abram'</td>
<td>16</td>
<td>12</td>
<td>12</td>
<td>1.9</td>
</tr>
<tr>
<td>([a-zA-Z]+) 'Joseph'</td>
<td>51</td>
<td>30</td>
<td>36</td>
<td>5.6</td>
</tr>
</tbody>
</table>
time (milisecond) for parsing some languages

<table>
<thead>
<tr>
<th>language</th>
<th>Lex/ Yacc</th>
<th>leg</th>
<th>LPEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>lists</td>
<td>113</td>
<td>150</td>
<td>93</td>
</tr>
<tr>
<td>arithmetic expressions</td>
<td>100</td>
<td>147</td>
<td>107</td>
</tr>
<tr>
<td>&quot;simple language&quot;</td>
<td>110</td>
<td>147</td>
<td>130</td>
</tr>
</tbody>
</table>
Conclusions

• PEG offers a nice conceptual base for pattern matching
• LPEG unifies matching, searching, and substitutions; it also unifies captures and semantic actions
• LPEG implements PEG with a performance competitive with other pattern-matching tools and with other parsing tools
Conclusions

• implementation with 2200 lines of C + 200 lines of Lua
• prototype implementation of a JIT: 3x faster
• LPEG seems particularly suited for languages that are too complex for regex but too simple for lex/yacc
  • DSL, XML, regexs(!)
• still missing Unicode support