LPEG: a new approach to pattern matching in Lua

Roberto Ierusalimschy

Pontifícia Universidade Católica
do Rio de Janeiro

(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 | ab) (cd | bcde)) e? ⊗ "abcde"
→"a" - "bcde" - ""

(a | ab) ((cd | bcde) e?) ⊗ "abcde"
→"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)
  - Aho & Ullman. The Theory of Parsing, Translation and Compiling. Prentice Hall, 1972.

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 <- A_1 / A_2 / ...

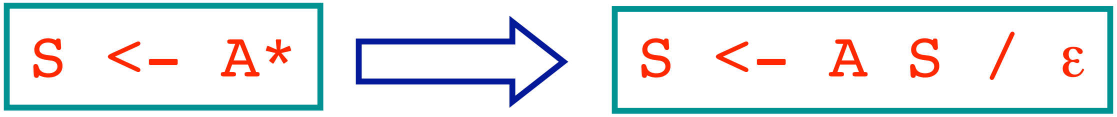
- to match A, try first A_1
- if it fails, backtrack and try A_2
- repeat until a match

Restricted Backtracking

```
S <- A B
A <- A1 / A2 / ...
```

- once an alternative A_1 matches for `A`, no more backtrack for this rule
- even if `B` fails!

Example: greedy repetition



`S <- A*`

`S <- A S / ε`

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

Non-blind greedy repetition

```
S <- 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 <- . S / ','
```

Non-blind non-greedy repetition

```
S <- B / A S
```

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

```
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

```
EOS <- !.
```

```
comment <- '/*' (!'*/' .)* '*/'
```

- next grammar matches $a^n b^n c^n$
  - a non context-free language

```
S  <- &P1 P2
P1 <- AB 'c'
AB <- 'a' AB 'b' / ε
P2 <- 'a'* BC !.
BC <- 'b' BC 'c' / ε
```

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

```
lower = lpeg.R("az")
upper = lpeg.R("AZ")
letter = lower + upper
digit = lpeg.R("09")
alphanum = letter + digit + "_"
```

LPEG basic constructs

```
lpeg.R("xy")     -- range
lpeg.S("xyz")    -- set
lpeg.P("name")   -- literal
lpeg.P(number)   -- that many characters
P1 + P2          -- ordered choice
P1 * P2          -- concatenation
-P               -- not P
P1 - P2          -- P1 if not P2
P^n              -- at least n repetitions
P^-n             -- at most n repetitions
```

LPEG basic constructs: Examples

```
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:

```
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

```
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'

```
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 * P
```

```
(!P .)* P
```

```
{ P + 1 * lpeg.V(1) }
```

```
S <- P / . <S>
```

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:

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

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

Captures: examples

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

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

Captures: examples

```
list = re.compile"({}%w* (',' {}%w*)*) -> {}"

t = list:match"a,b,c,d")

-- t is {1,3,5,7}
```

Captures: examples

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

t = exp:match'(a b (c d) ())'

-- t is {'a', 'b', {'c', 'd'}, {}}
```

Captures: examples

```
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", "", ""
```

Captures: examples

```
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

Substitutions: example

```
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 `{~ ... ~}`

```
P = "{~ ('0' -> '1' / '1' -> '0' / .)* ~}"

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
```

Parsing Machine: Optimizations

```
'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
```

Parsing Machine: Optimizations

```
'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>
```

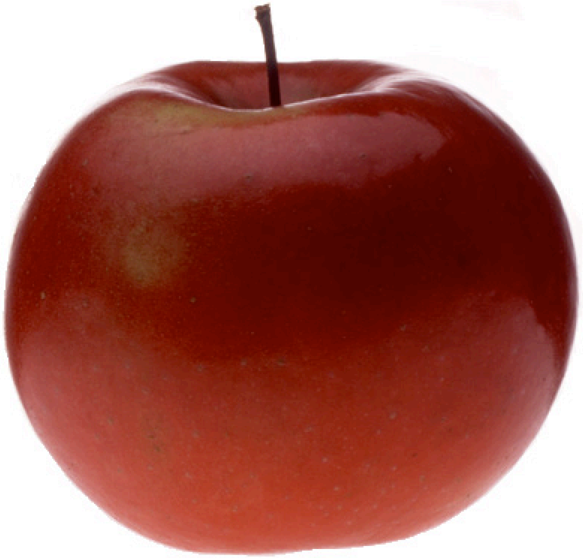
```
00: call -> 2
01: jmp -> 10
02: testchar 'a' (61)-> 7
03: choice -> 7 (1)
04: char 'n' (6e)
05: char 'a' (61)
06: commit -> 9
07: any * 1
08: jmp -> 2
09: ret
10: end
```

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>
```

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

Benchmarks



X

Benchmarks: Search

- programmed in LPEG:
  - `S <- '@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

pattern	PCRE	POSIX regex	Lua	LPEG
`'@the'`	5.3	14	3.6	40
`'Omega'`	6.0	14	3.5	40
`'Alpha'`	6.7	15	4.2	40
`'amethysta'`	27	38	24	47
`'heith'`	32	44	26	50
`'eartt'`	40	53	36	52

time (milisecond) for searching a string in the Bible

pattern	PCRE	POSIX regex	Lua	LPEG	false starts
`'@the'`	5.3	14	3.6	40	0
`'Omega'`	6.0	14	3.5	40	8853
`'Alpha'`	6.7	15	4.2	40	17,851
`'amethysta'`	27	38	24	47	256,897
`'heith'`	32	44	26	50	278,986
`'eartt'`	40	53	36	52	407,883

Search...

- because they are programmed in LPEG, we can optimize them:
  - `S <- '@the' / . <S>`
  - `S <- '@the' / . [^@]* <S>`

time (milisecond) for searching a string in the Bible

pattern	PCRE	POSIX regex	Lua	LPEG	LPEG (2)	false starts
`'@the'`	5.3	14	3.6	40	9.9	0
`'Omega'`	6.0	14	3.5	40	10	8853
`'Alpha'`	6.7	15	4.2	40	11	17,851
`'amethysta'`	27	38	24	47	21	256,897
`'heith'`	32	44	26	50	23	278,986
`'eartt'`	40	53	36	52	26	407,883

time (milisecond) for searching a pattern in the Bible

pattern	PCRE	POSIX regex	Lua	LPEG
`[a-zA-Z]{14,}`	10	15	16	4.0
`([a-zA-Z]+) *'Abram'`	16	12	12	1.9
`([a-zA-Z]+) *'Joseph'`	51	30	36	5.6

time (milisecond) for parsing some languages

language	Lex/ Yacc	leg	LPEG
`lists`	113	150	93
`arithmetic expressions`	100	147	107
`"simple language"`	110	147	130

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