LPEG: a new approach to pattern matching

Roberto Ierusalimschy

Pontifícia Universidade Católica do Rio de Janeiro

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: *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.
- revamped by Bryan Ford, MIT, in 2002
  - pattern-matching sugar
  - *Packrat* implementation

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

Restricted Backtracking

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

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

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
- `!.` matches end of input
  - any character fails
- `&A` (and predicate) is sugar for `!!A`

Predicates: example

- predicates allow PEGs for non context-free languages
- next grammar matches $a^n b^n c^n$

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

PEG x (real) regular expressions

- regular expressions are too limited
  - problems with captures and non-greedy repetitions
  - problems with complement
- PEG allows whole grammars
  - nesting, etc.

PEG x "regular expressions"

- PEG has a clear and formally-defined semantics
  - instead of a set of ad-hoc operators
- PEG has a clear and formally-defined performance model
  - no need for ad-hoc optimizations
- PEG allows a simple and efficient implementation
  - parsing machines

LPEG: PEG for Lua

- a small library for pattern matching based on PEGs
- SNOBOL tradition: language constructors to build patterns
  - verbose, but clear

```
letter = lpeg.R("az")
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 grammars

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

```
V = lpeg.V
addop = lpeg.S"+-"
mulop = lpeg.S"*/"
number = lpeg.R"09"^1

exp = lpeg.P{"Exp",
  Exp = V"Factor" * (addop * V"Factor")^0,
  Factor = V"Term" * (mulop * V"Term")^0,
  Term = number + "(" * V"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 + 1 * lpeg.V(1) }
```

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: 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: example

```
local Q = lpeg.P'"'
local R = (1 - lpeg.S',"\n')
local IQ = (1 - Q) + (Q * Q / '"')
local field = Q * lpeg.Cs(IQ^0) * Q
            + lpeg.C(R^0)

local End = lpeg.P'\n' + -1

local record = field * (',' * field)^0 * End

function csv (s)
  return lpeg.match(record, s)
end
```

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

Conclusions

- PEG offers a nice conceptual base for pattern matching
- LPEG implements PEG with a performance competitive with other pattern-matching tools
- for those that do not like its verboseness, there is a module that accepts regexp-like notation
  - some limitations when using other Lua values