LPeg: an Alternative to regexs based on PEGs

Roberto Ierusalimschy

PEG: Parsing Expression Grammars

- Not totally unlike context-free grammars
- Incorporate useful constructs from pattern-matching systems
  - `a?`, `a*`, `a+`, `[a-z]`
- Key differences from CFGs: restricted backtracking and predicates

PEG: Short History

- Restricted backtracking and the not predicated first proposed by Alexander Birman, ~1970
- Later described by Aho & Ullman as TDPL (Top Down Parsing Languages) and GTDPL (general TDPL)
  - Aho & Ullman. The Theory of Parsing, Translation and Compiling. Prentice Hall, 1972
- Revamped by Bryan Ford, MIT, 2002
  - pattern-matching sugar and Packrat implementation

PEG x CFG

- There is an ongoing discussion about how PEG compares with CFG
  - I do not care : -)
- LPEG uses PEG as an alternative for pattern matching, not for parsing
  - although it can be used for parsing, too
  - particularly good for "little languages", like regexs, email addresses, CSV, etc.

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, backtrack and try E followed by F
- If all options fail, A fails

Restricted Backtracking

- `S <- A B;  A <- A1 / A2 / ...`
- To match A, first try A1
- If it fails, backtrack and try A2
- Repeat until a match
- Once an alternative matches, no more backtrack for this rule
  - if if B fails!
- Resulting semantics equivalent to parser combinators using *Maybe* instead of *List*

Predicates

- `!exp` only matches if `exp` fails
    - either `exp` or `!exp` must fail, so predicate never consumes any input
- `&exp` is sugar for `!!exp`
- Predicates allow arbitrary look ahead
- Example: `!.` matches end of input

PEG x regexs

- PEG has a clear and formally-defined semantics
  - instead of an ad-hoc set of operators and rules
- PEG allows whole grammars
  - properly contains all LR(k) languages
- PEG can express most regex extensions
  - possessive and lazy repetitions, independent sub-patterns, look ahead, etc.
- PEG allows abstraction (names)

rexex for Mail::RFC822::Address Validation

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0
31]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+
(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\
r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
 \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)
?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
   ...  (60 lines deleted)
\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:
\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])
*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\
.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(
?:\r\n)?[ \t])*))*)?;\s*)
```

http://www.ex-parrot.com/pdw/Mail-RFC822-Address.html

Example: Repetition

- `S <- A*        --->   S <- A S / ε`
- Ordered choice makes repetition greedy
- Restricted backtracking makes it *blind*
- Matches maximum span of As
  - *possessive* repetition

Example: Non-Blind Greedy Repetition

- `S <- A* B      --->  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 also fails
  - engine backtracks until a match
- Result is a conventional greedy repetition

Example: Lazy Repetition

- `A*?B      --->   S <- B / A S`
- Ordered choice makes repetition lazy
- Matches miminum number of As until a B
  - also called *reluctant* repetition
- Another translation: `(!B A)* B`

PEG x LPeg

- LPeg is mainly intended for pattern matching
  - small change in the grammar of grammars
- LPeg can be used Snobol-style, where patterns are first-class objects
- LPeg is implemented through a "parsing machine", instead of memoization
- LPeg offers a diverse set of *captures*
  - includes conventional regex captures, substitution, and up to semantic actions

Small Change in Patterns

PEG

```
Grammar    <- S Definition+ !.
Definition <- Identifier '<-' S Expression
Expression <- ...
Primary    <- '(' S Expression ')' S
```

LPEG

```
Pattern    <- Grammar / Expression
Grammar    <- S Definition+ !.
Definition <- Identifier '<-' S Expression
Expression <- ...
Primary    <- '(' S Pattern ')' S
```

Snobol Style

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

regex style

```
letter = re.compile("[a-zA-Z]")
```

```
print(re.find("hello World", "[A-Z]"))
  --> 7

print(re.match("hello world", "{ [a-zA-Z]+ }"))
  --> hello

print(re.match("hello world", "({[a-zA-Z]+} %s*)*"))
  --> hello    world
```

Captures

- `{patt}` - captures the substring that matches `patt`
- `patt -> {}` - creates a list with all captures from `patt`
- `patt -> string` - captures string
  - with placeholders changed to captures from `patt`
- `{~ patt ~}` - captures substring, changing all substrings captured inside it by thei values of the captures

Examples

```
p = "{~ ([aeiou] -> '(%0)' / .)* ~}"
print(re.match("a lovely day", p))
  --> (a) l(o)v(e)ly d(a)y
```

```
p = "R <- ({.} R) -> '%2%1' / {''}"
print(re.match("a lovely day", p))
  --> yad ylevol a
```

Example: SExp

```
SExp = [[
SExp <- ('(' Sp SExp* ')' Sp) -> {} / Atom
Atom <- {[^()%s]+} Sp
Sp <- %s*
]]
```

```
t = re.match("(a (b c) () d)", p)
-- t == {'a', {'b', 'c'}, {}, 'd'}
```

Example: CSV

```
record = re.compile[[
  record <- (field (',' field)* end) -> {}
  field <- escaped / nonescaped
  nonescaped <- { [^,"%nl]* }
  escaped <- '"' {~ ([^"] / '""' -> '"')* ~} '"'
  end <- (%nl / !.)
]]
```

```
s = [["a ""name""", another name]]
t = record:match(s)
-- t == {'a "name"', ' another name'}
```

More Captures

- {:name: exp :} - named capture
- =name - back reference to capture with that name

```
{ {:q: ["'] :} ('\' . / !(=q) .)* =q }
```

Indented Text

```
first line
  subline 1
  subline 2
second line
third line
  subline 3.1
    subline 3.1.1
  subline 3.2
```

```
{'first line'; {'subline 1'; 'subline 2'};
 'second line';
 'third line'; { 'subline 3.1'; {'subline 3.1.1'};
                 'subline 3.2'};
}
```

Indented Text

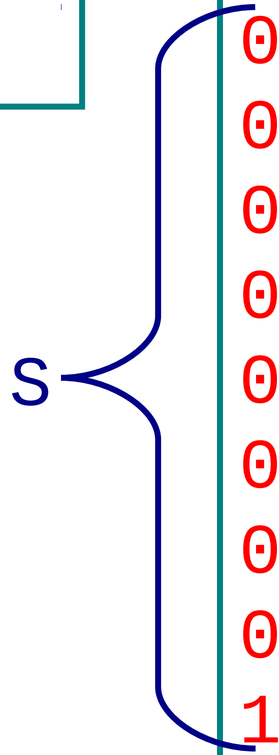
```
p = re.compile[[
  block <- (firstline
             (otherline / nestedblock)*) -> {}
  firstline <- {:ident: ' '* :} line
  otherline <-  =ident !' ' line
  nestedblock <- &(=ident ' ') block
  line <- {[^%nl]*} %nl
]]
```

LPeg for Mail Address Validation

```
address <- mailbox / group
group <- phrase ":" mailboxes? ";"
phrase <- word ("," word?)*
mailboxes <- mailbox ("," mailbox?)*
mailbox <- addr_spec / phrase route_addr
route_addr <- "<" route? addr_spec ">"
route <- ("@" domain) ("," ("@" domain)?)* ":"
addr_spec <- local_part "@" domain
local_part <- word ("." word)*
domain <- sub_domain ("." sub_domain)*
sub_domain <- domain_ref / domain_literal
domain_ref <- atom
domain_literal <- "[" ([^][] / "\" .)* "]"
word <- atom / quoted_string
atom <- [^] %c()<>@,;:\".[]+
quoted_string <- '"' ([^"\%nl] / "\" .)* '"'
```

Implementation: Example

```
S = re.compile[[
  S <- 'xuxu' / . S
]]
```

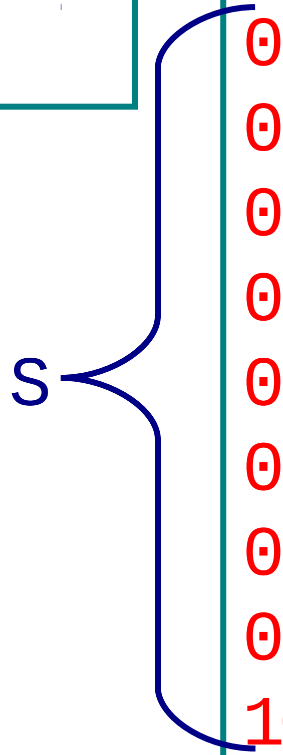


```
00: call -> 2
01: jmp -> 11
02: choice -> 8
03: char 'x'
04: char 'u'
05: char 'x'
06: char 'u'
07: commit -> 10
08: any
09: call -> 2
10: ret
11: end
```

Implementation: Optimizations

```
S = re.compile[[
  S <- 'xuxu' / . S
]]
```

```
00: call -> 2
01: jmp -> 11
02: testchar 'x'-> 8
03: choice -> 8 (1)
04: char 'u'
05: char 'x'
06: char 'u'
07: commit -> 10
08: any
09: jmp -> 2
10: ret
11: end
```



That is it. Thank you.