Lpeg Tutorial

Simple Matching

LPeg is a powerful notation for matching text data, which is more capable than Lua string patterns and standard regular expressions. However, like any language you need to know the basic words and how to combine them.

The best way to learn is to play with patterns in an interactive session, first by defining some shortcuts:

```
$ lua -llpeg
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio
> match = lpeg.match -- match a pattern against a string
> P = lpeg.P -- match a string literally
> S = lpeg.S  -- match anything in a set
> R = lpeg.R  -- match anything in a range
```

If you dont' want to create shorcuts manually, you can do this:

```
> setmetatable(_ENV or _G, { __index = lpeg or require"lpeg" })
```

I don't recommend doing this in serious code, but to explore LPeg, it is very convenient.

Matches occur against the *start* of the string, and successful matches return the position immediately after the successful match, or `nil` if unsuccesful. (Here I'm using the fact that `f'x'` is equivalent to `f('x')` in Lua; using single quotes has the same meaning as double quotes.)

```
> = match(P'a','aaa')
2
> = match(P'a','123')
nil
```

It works like `string.find`, except it only returns one index.

You can match against *ranges* or *sets* of characters:

```
> = match(R'09','123')
2
> =  match(S'123','123')
2
```

Matching more than one item is done with the ^ operator. In this case, the match is equivalent to the Lua pattern '^a+' - one or more occurrances of 'a':

```
> = match(P'a'^1,'aaa')
4
```

Combining patterns in order is done with the `*` operator. This is equivalent to '^ab*' - one 'a' followed by zero or more 'b's:

```
> = match(P'a'*P'b'^0,'abbc')
4
```

So far, lpeg is giving us a more verbose way of expressing regular expressions, but these patterns are *composible* - they can be easily built up from simpler patterns, without awkward string operations. In this way, lpeg patterns can be made to be easier to read than their equivalent regular expressions. Note that you can often leave out an explicit `P` call when constructing patterns, if one of the arguments is already a pattern:

```
> maybe_a = P'a'^-1  -- one or zero matches of 'a'
> match_ab = maybe_a * 'b'
> = match(match_ab, 'ab')
3
> =  match(match_ab, 'b')
2
> =  match(match_ab, 'aaab')
nil
```

The `+` operator means *either* one or the other pattern:

```
> either_ab = (P'a' + P'b')^1 -- sequence of either 'a' or 'b'
> = either_ab:match 'aaa'
4
> =  either_ab:match 'bbaa'
5
```

Note that the pattern object has a `match` method!

Of course, `S'ab'^1` would be a shorter way to say this, but the arguments here can be arbitary patterns.

Basic Captures

Getting the index after a match is all very well, and you can then use `string.sub` to extract the strings. But there are ways of explicitly asking for *captures*:

```
> C = lpeg.C  -- captures a match
> Ct = lpeg.Ct -- a table with all captures from the pattern
```

The first is equivalent to how '(...)' is used in Lua patterns (or '\(...\)' in regular expressions)

```
> digit = R'09' -- anything from '0' to '9'
> digits = digit^1 -- a sequence of at least one digit
> cdigits= C(digits)  -- capture digits
> = cdigits:match '123'
123
```

So to get the string value, enclose the pattern in `C`.

This pattern doesn't cover a general integer, which may have a '+' or '-' up front:

```
> int = S'+-'^-1 * digits
> = match(C(int),'+23')
+23
```

Unlike with Lua patterns or regular expressions, you don't have to worry about escaping 'magic' characters - every character in a string stands for itself: '(','+','*', etc are just their ASCII equilvalents.

A special kind of capture is provided by the `/` operator - it passes the captured string through a function or a table. Here I'm adding one to the result, just to show that the result has been converted into a number with `tonumber`:

```
> =  match(int/tonumber,'+123') + 1
124
```

Note that multiple captures can be returned by a match, just like `string.match`. This is equivalent to '^(a+)(b+)':

```
> = match(C(P'a'^1) * C(P'b'^1), 'aabbbb')
aa      bbbb
```

Building more complicated Patterns

Consider general floating-point numbers:

```
> function maybe(p) return p^-1 end
> digits = R'09'^1
> mpm = maybe(S'+-')
> dot = '.'
> exp = S'eE'
> float = mpm * digits * maybe(dot*digits) * maybe(exp*mpm*digits)
> = match(C(float),'2.3')
2.3
> = match(C(float),'-2')
-2
> = match(C(float),'2e-02')
2e-02
```

This lpeg pattern is *easier* to read than the regular expression equivalent '[-+]?[0-9]+\.?[0-9]+([eE][+-]?[0-9]+)?'; shorter is always better! One reason is that we can work with patterns as *expressions*: factor out common patterns, write functions for convenience and clarity, etc. Note that there is no penalty for writing things out in this fashion; lpeg remains a very fast way to parse text!

More complicated structures can be composed from these building blocks. Consider the task of parsing a list of floating point numbers. A list is a number followed by zero or more groups consisting of a comma and a number:

```
> listf = C(float) * (',' * C(float))^0
> = listf:match '2,3,4'
2       3       4
```

That's cool, but it would be even cooler to have this as an actual list. This is where `lpeg.Ct` comes in; it collects all the captures within a pattern into a table.

```
= match(Ct(listf),'1,2,3')
table: 0x84fe628
```

Stock Lua does not pretty-print tables, but you can use [? Microlight] for this job:

```
> tostring = require 'ml'.tstring
> = match(Ct(listf),'1,2,3')
{"1","2","3"}
```

The values are still strings. It's better to write `listf` so that it converts its captures:

```
> floatc = float/tonumber
> listf = floatc * (',' * floatc)^0
```

This way of capturing lists is very general, since you can put *any* expression that captures in the place of `floatc`. But this list pattern is still too restrictive, because generally we want to ignore whitespace

```
> sp = P' '^0  -- zero or more spaces (like '%s*')
> function space(pat) return sp * pat * sp end -- surrond a pattern with optional space
> floatc = space(float/tonumber)
> listc = floatc * (',' * floatc)^0
> =  match(Ct(listc),' 1,2, 3')
{1,2,3}
```

It's a matter of taste, but here I prefer to allow optional space around the *items*, rather than allowing space specifically around the *delimiter* ','.

With lpeg, we can be programmers again with pattern matching, and reuse patterns:

```
function list(pat)
    pat = space(pat)
    return pat * (',' * pat)^0
end
```

So, a list of identifiers (according to the usual rules):

```
> idenchar = R('AZ','az')+P'_'
> iden = idenchar * (idenchar+R'09')^0
> =  list(C(iden)):match 'hello, dolly, _x, s23'
"hello" "dolly" "_x"    "s23"
```

Using explicit ranges seems old-fashioned and error-prone. A more portable solution is to use the lpeg equivalent of *character classes*, which are by definition locale-independent:

```
> l = {}
> lpeg.locale(l)
> for k in pairs(l) do print(k) end
"punct"
"alpha"
"alnum"
"digit"
"graph"
"xdigit"
"upper"
"space"
"print"
"cntrl"
"lower"
> iden =  (l.alpha+P'_') * (l.alnum+P'_')^0
```

Given this definition of `list`, it's easy to define a simple subset of the common CSV format, where each record is a list separated by a linefeed:

```
> listf =  list(float/tonumber)
> csv = Ct( (Ct(listf)+'\n')^1 )
> =  csv:match '1,2.3,3\n10,20, 30\n'
{{1,2.3,3},{10,20,30}}
```

One good reason to learn lpeg is that it performs very satisfactorily. This pattern is a *lot* faster than parsing the data with Lua string matching.

String Substitution

I will show that lpeg can do all that `string.gsub` can do, and more generally and flexibly.

One operator that we have not used yet is `-`, which means 'either/or'. Consider the problem of matching double-quoted strings. In the simplest case, they are a double-quote followed by any characters which are not a double-quote, followed by a closing double-quote. `P(1)` matches *any* single character, i.e. it is the equivalent of '.' in string patterns. A string may be empty, so we match zero or more non-quote characters:

```
> Q = P'"'
> str = Q * (P(1) - Q)^0 * Q
> = C(str):match '"hello"'
"\"hello\""
```

Or you may want to extract the contents of the string, without quotes. In this context, just using `1` instead of `P(1)` is not ambiguous, and in fact this is how you will usually see this 'any x which is not a P' pattern:

```
> str2 = Q * C((1 - Q)^0) * Q
> = str2:match '"hello"'
"hello"
```

This pattern is obviously generalizable; often the terminating pattern is not the same as the final pattern:

```
function extract_quote(openp,endp)
    openp = P(openp)
    endp = endp and P(endp) or openp
    local upto_endp = (1 - endp)^1
    return openp * C(upto_endp) * endp
end

> return  extract_quote('(',')'):match '(and more)'
"and more"
> = extract_quote('[[',']]'):match '[[long string]]'
"long string"
```

Now consider translating Markdown `code` (back-slash enclosed text) into the format understood by the Lua wiki (double-brace enclosed text). The naive way is to extract the string and concatenate the result, but this is clumsy and (as we will see) limit our options tremendously.

```
function subst(openp,repl,endp)
    openp = P(openp)
    endp = endp and P(endp) or openp
    local upto_endp = (1 - endp)^1
    return openp * C(upto_endp)/repl * endp
end

> =  subst('`','{{%1}}'):match '`code`'
"{{code}}"
> =  subst('_',"''%1''"):match '_italics_'
"''italics''"
```

We've come across the capture-processing operator `/` before, using `tonumber` to convert numbers. It also understands strings in a very similar format to `string.gsub`, where `%n` means the n-th capture.

This operation can be expressed exactly as:

```
> = string.gsub('_italics_','^_([^_]+)_',"''%1''")
"''italics''"
```

But the advantage is that we don't have to build up a custom string pattern and worry about escaping 'magic' characters like '(' and ').

`lpeg.Cs` is a *substitution capture*, and it provides a more general module of global string substitution. In the lpeg manual, there is this equivalent to `string.gsub`:

```
function gsub (s, patt, repl)
    patt = P(patt)
    local p = Cs ((patt / repl + 1)^0)
    return p:match(s)
end

> =  gsub('hello dog, dog!','dog','cat')
"hello cat, cat!"
```

To understand the difference, here's that pattern using plain `C`:

```
> p = C((P'dog'/'cat' + 1)^0)
> = p:match 'hello dog, dog!'
"hello dog, dog!"       "cat"   "cat"
```

The `C` here just captures the whole match, and each '/' adds a new capture with the value of the replacement string.

With `Cs`, *everything* gets captured, and a string is built out of all the captures. Some of those captures get modified by '/', and so we have substitutions.

In Markdown, block quoted lines begin with '> '.

```
lf = P'\n'
rest_of_line_nl = C((1 - lf)^0*lf)         -- capture chars upto \n
quoted_line = '> '*rest_of_line_nl       -- block quote lines start with '> '
-- collect the quoted lines and put inside [[[..]]]
quote = Cs (quoted_line^1)/"[[[\n%1]]]\n"

> = quote:match '> hello\n> dolly\n'
"[[[
> hello
> dolly
]]]
"
```

That's not quite right - `Cs` captures everything, including the '> '. But we can force some captures to return empty strings: }}}

```
function empty(p)
    return C(p)/''
end

quoted_line = empty ('> ') * rest_of_line_nl
...
```

Now things will work correctly!

Here is the program used to convert this document from Markdown to Lua wiki format:

```
local lpeg = require 'lpeg'

local P,S,C,Cs,Cg = lpeg.P,lpeg.S,lpeg.C,lpeg.Cs,lpeg.Cg

local test = [[
## A title

here _we go_ and `a:bonzo()`:

    one line
    two line
    three line

and `more_or_less_something`

[A reference](http://bonzo.dog)

> quoted
> lines

]]

function subst(openp,repl,endp)
    openp = P(openp)  -- make sure it's a pattern
    endp = endp and P(endp) or openp
    -- pattern is 'bracket followed by any number of non-bracket followed by bracket'
    local contents = C((1 - endp)^1)
    local patt = openp * contents * endp
    if repl then patt = patt/repl end
    return patt
end

function empty(p)
    return C(p)/''
end

lf = P'\n'
rest_of_line = C((1 - lf)^1)
rest_of_line_nl = C((1 - lf)^0*lf)

-- indented code block
indent = P'\t' + P'    '
indented = empty(indent)*rest_of_line_nl
-- which we'll assume are Lua code
block = Cs(indented^1)/'    [[[!Lua\n%1]]]\n'

-- use > to get simple quoted block
quoted_line = empty('> ')*rest_of_line_nl
quote = Cs (quoted_line^1)/"[[[\n%1]]]\n"

code = subst('`','{{%1}}')
italic = subst('_',"'''%1'''")
bold = subst('**',"''''%1''''")
rest_of_line = C((1 - lf)^1)
title1 = P'##' * rest_of_line/'=== %1 ==='
title2 = P'###' * rest_of_line/'== %1 =='

url = (subst('[',nil,']')*subst('(',nil,')'))/'[%2 %1]'

item = block + title1 + title2 + code + italic + bold + quote + url + 1
text = Cs(item^1)
```

```
if arg[1] then
    local f = io.open(arg[1])
    test = f:read '*a'
    f:close()
end

print(text:match(test))
```

Due to an escaping problem with this Wiki, I had to substitute '[' for '{', etc in this source. Be warned!

SteveDonovan, 12 June 2012

Group and back captures

This section will dissect the behavior of group and back captures (`Cg()` and `Cb()` respectively).

Group captures (hereafter "groups") come in two flavors: named and anonymous.

```
Cg(C"baz" * C"qux", "name") -- named group.

Cg(C"foo" * C"bar")         -- anonymous group.
```

Let's first get the easy one out of the way: named groups inside table captures.

```
Ct(Cc"foo" * Cg(Cc"bar" * Cc"baz", "TAG")* Cc"qux"):match""
--> { "foo", "qux", TAG = "bar" }
```

In a table capture, the value of the first capture inside the group (`"bar"`) is assigned to the corresponding key (`"TAG"`) in the table. As you can see, `Cc"baz"` got lost in the process. The label must be a string (or a number that will be automatically converted to a string).

Note that the group must be a direct child of the table, otherwise, the table capture will not handle it:

```
Ct(C(Cg(1,"foo"))):match"a"
--> {"a"}
```

Of captures and values

Before delving into groups proper, we must first explore a subtlety in the way captures handle their subcaptures.

Some captures operate on the values produced by their subcaptures, while others operate on the capture objects. This is sometimes counter-intuitive.

Let's take the following pattern:

```
(1 * C( C"b" * C"c" ) * 1):match"abcd"
--> "bc", "b", "c"
```

As you can see, it inserts three values in the capture stream.

Let's wrap it in a table capture:

```
Ct(1 * C( C"b" * C"c" ) * 1):match"abcd"
--> { "bc", "b", "c" }
```

`Ct()` operates on values. In the last example, the three values that are inserted in order in the table.

Now, let's try a substitution capture:

```
Cs(1 * C( C"b" * C"c" ) * 1):match"abcd"
--> "abcd"
```

`Cs()` operates on captures. It scans the first level of its nested captures, and only takes the first value of each one. In the above example, `"b"` and `"c"` are thus discarded. Here's another example that may make things more clear:

```
function the_func (bcd)
    assert(bcd == "bcd")
    return "B", "C", "D"
end

Ct(1 * ( C"bcd" / the_func ) * 1):match"abcde"
--> {"B", "C", "D"}  -- All values are inserted.

Cs(1 * ( C"bcd" / the_func ) * 1):match"abcde"
--> "aBe"   -- the "C" and "D" have been discarded.
```

A more detailed account of the by value / by capture behaviour of each kind of capture will be the topic of another section.

Capture opacity

Another important thing to realise is that most captures shadow their subcaptures, but some don't. As you can see in the last example, the value of `C"bcd"` is passed to the `/function` capture, but it doesn't end in the final capture list. `Ct()` and `Cs()` are also opaque in this regard. They only produce, resectively, one table or one string.

On the other hand, `C()` is transparent. As we've seen above, the subcaptures of `C()` are also inserted in the stream.

```
C(C"b" * C"c"):match"bc" --> "bc", "b", "c"
```

The only transparent captures are `C()` and the anonymous `Cg()`.

Anonymous groups

`Cg()` wraps its subcaptures in a single capture object, but doesn't produce anything of its own. Depending on the context, either all of its values will be inserted, or only the first one.

Here are a few examples for the anonymous groups:

```
(1 * Cg(C"b" * C"c" * C"d") * 1):match"abcde"
--> "b", "c", "d"

Ct(1 * Cg(C"b" * C"c" * C"d") * 1):match"abcde"
--> { "b", "c", "d" }

Cs(1 * Cg(C"b" * C"c" * C"d") * 1):match"abcde"
--> "abe" -- "c" and "d" are dropped.
```

Where this behavior is useful? In folding captures.

Let's write a very basic calculator, that adds or substracts one digit numbers.

```
function calc(a, op, b)
    a, b = tonumber(a), tonumber(b)
    if op == "+" then
        return a + b
    else
        return a - b
    end
end

digit = R"09"

calculate = Cf(
    C(digit) * Cg( C(S"+-") * C(digit) )^0
    , calc
)
calculate:match"1+2-3+4"
--> 4
```

The capture tree will look like this [*]:

```
{"Cf", func = calc, children = {
    {"C", val = "1"},
    {"Cg", children = {
        {"C", val = "+"},
        {"C", val = "2"}
    } },
    {"Cg", children = {
        {"C", val = "-"},
        {"C", val = "3"}
    } },
    {"Cg", children = {
        {"C", val = "+"},
        {"C", val = "4"}
    } }
} }
```

You probably see where this is going... Like `Cs()`, `Cf()` operates on capture objects. It will first extract the first value of the first capture, and use it as the initial value. If there are no more captures, this value becomes the value of the `Cf()`.

But we have more captures. In our case, it will pass all the values of the second capture (the group) to `calc()`, tacked after the value of the first one. Here's the evaluation of the above `Cf()`

```
first_arg = "1"
next_ones: "+", "2"
```

```
first_arg = calc("1", "+", "2") -- 3, calc() returns numbers

next_ones: "-", "3"
first_arg = calc(3, "-", "3")

next_ones: "+", "4"
first_arg = calc(0, "+", "4")

return first_arg -- Tadaaaa.
```

[*] Actually, at match time, the capture objects only store their bounds and auxiliary data (like `calc()` for the `Cf()`). The actual values are produced sequencially after the match has completed, but, it makes things more clear as displayed above. In the above example, the values of the nested `C()` and `Cg(C(),C())` are actually produced one at a time, at each corresponding cycle of the folding process.

Named groups

The (named `Cg()` / `Cb()`) pair has a behavior similar to the anonymous `Cg()`, but the values captured in the named `Cg()` are not inserted locally. They are teleported, and end up inserted in the stream at the place of the `Cb()`.

Here's an example:

```
( 1 * Cg(C"bc", "FOOO") * C"d" * 1 * Cb"FOOO" * Cb"FOOO"):match"abcde"
-- > "d", "bc", "bc"
```

Warp... and duplication if there is more than one `Cb()`. Another example:

```
( 1 * Cg(C"b" * C"c" * C"d", "FOOO") * C"e" * Ct(Cb"FOOO") ):match"abcde"
--> "e", { "b", "c", "d" }
```

Usually, for the sake of clarty, in my code, I alias `Cg()` to `Tag()`. I use the former for anonymous groups, and the latter for named groups.

`Cb"FOOO"` will look back for a corresponding `Cg()` that has succeeded. It goes back and up in the tree, and consumes captures. In other words, it searches its elder siblings, and the elder siblings of its parents, but not the parents themselves. Neither does it test the children of the siblings/siblings of ancestors.

It proceeds as follows (start from the `[ #### ] <--- [[ START ]]` and follow the numbers back up).

The `[ numbered ]` captures are the captures that are tested on order. The ones marked with `[ ** ]` are not, for the various reasons listed. This is hairy, but AFAICT complete.

```
Cg(-- [ ** ] ... This one would have been seen,
   -- if the search hadn't stopped at *the one*.
   "Too late, mate."
    , "~@~"
)

* Cg( -- [ 3 ] The search ends here. <--------------[[ Stop ]]
```

```
        "This is *the one*!"
        , "~@~"
    )

    * Cg(--  [ ** ] ... The great grand parent.
                    -- Cg with the right tag, but direct ancestor,
                    -- thus not checked.


        Cg( -- [ 2 ] ... Cg, but not the right tag. Skipped.
            Cg( -- [ ** ] good tag but masked by the parent (whatever its type)
                "Masked"
                , "~@~"
            )
            , "BADTAG"
        )

        * C( -- [ ** ] ... grand parent. Not even checked.

            (
                Cg( -- [ ** ] ... This subpattern will fail after Cg() succeeds.
                    -- The group is thus removed from the capture tree, and will
                    -- not be found dureing the lookup.
                    "FAIL"
                    , "~@~"
                )
                * false
            )

            + Cmt(  -- [ ** ] ... Direct parent. Not assessed.
                C(1) -- [ 1 ] ... Not a Cg. Skip.

                * Cb"~@~"   -- [ #### ]  <----------------- [[ START HERE ]] --
                , function(subject, index, cap1, cap2)
                    return assert(cap2 == "This is *the one*!")
                end
            )
        )
        , "~@~" -- [ ** ] This label goes with the great grand parent.
    )
```

-- PierreYvesGerardy

RecentChanges · preferences
edit · history
Last edited February 18, 2019 6:39 pm GMT (diff)