Parsing complex data formats in LuaTeX with LPEG

Henri Menke

Abstract

Even though it is possible to read external files in TeX, extracing information from them is rather difficult. Ad-hoc solutions tend to use nested if statements or regular expressions provided by several macro packages. However, these quick hacks don't scale well and quickly become unmaintainable.

LuaTeX comes to the rescue with its embedded LPEG library for Lua. LPEG provides a Domain Specific Embedded Language (DSEL) that allows to write grammars in a natural way. In this article I will give a quick introducing to Parsing Expression Grammars (PEG) and then show how to write simple parsers in Lua with LPEG. Finally we will build a JSON parser to demonstrate how easy it is to even parse complex data formats.

1 Quick introduction to LPEG and Lua

The LPEG library [1] is an implementation of Parsing Expression Grammars for the Lua language. It provides a Domain Specific Embedded Language for this task. Its domain is obviously parsing. It is embedded in Lua using overloading of arithmetic operators to give it a natural syntax. The language it implements is PEG. The LPEG library has been included in LuaTeX since the beginning [2]. The examples in this article are based on the talk "Using Spirit X3 to Write Parsers" which was given by Michael Caisse at CppCon 2015 [3], where the speaker introduces the Spirit X3 library for C++ to write parsers using PEG. The Spirit library is not too dissimilar from LPEG and if you are looking for a parser generator for C++, I recommend it.

To make sure that we are all on the same page and the reader can easily understand the syntactic constructions used throughout this manuscript, we review some aspects of the Lua language. First of all, it is to note that all variables are global by default, whereas local variables have to be preceded by the `local` keyword.

```
local x = 1
```

Most of the time we want definitions to be scoped so this pattern will show up very often. Another important thing to note about the Lua language is that in contrast to many other programming languages, functions are first class variables. That means that when we declare a function, what we actually do is assign a value of type `function` to a variable. That is to say, that these two statements are equivalent.

```
function f(...) end    f = function(...) end
```

Lua implements only a single complex datastructure, the table. Tables in Lua act as arrays and key-value storage at the same time, in fact it is possible to mix both forms of access within a single instance as in the following example.

```
local t = { 11, 22, 33, foo = "bar" }
print(t[2], t["foo"], t.foo) -- 22 bar bar
```

Note that array indexing in Lua starts at `1`. For tables and strings Lua offers a useful shortcut. When calling a function with a single literal string or table, parentheses can be omitted. In the following snippet the statements on the left are equivalent to the ones on the right.

```
f("foo")            f"foo"
f({ 11, 22, 33 })   f{ 11, 22, 33 }
```

Especially when programming with LPEG this shortcut can save a lot of typing and, when used to it, makes the code a lot more readable. I will make extensive use of this technique.

2 Why use PEG?

Before we delve into the inner workings of LPEG, let me first give some motivation as to why we would like to build parsers using PEG. Imagine trying to verify that input has a certain format, e.g. a date in the form day-month-year: `09-08-2019`. One approach might be to split the input at the hyphens and verify that each field only contains numbers, which is simple enough to implement using TeX macro code. However, the task quickly becomes more complicated when further requirements come into play. Only because something is made up of three groups of numbers doesn't make it a valid date. In situations like these, regular expressions (regex) sound like a good solution and in fact, the regex to parse a "valid" date looks faily innocent.

```
[0-3][0-9]-[0-1][0-9]-[0-9]{4}
```

I put "valid" in quotation marks, because obviously this regex misses several cases, such as different number of days in different months or leap years. I encourage the reader to look up a regular expression which covers these special cases, to get an impression as to how quickly regex gets out of hand. To top it off, neither a pure TeX solution nor regex implementations in TeX are fully expandable which is often desirable. Maybe they can be made fully expandable but not without tremendous effort.

3 What is PEG?

The question remains, how does PEG help us here? Let's first look at a more or less formal definition of PEG, adapted from Wikipedia [4]. A parsing expression grammar consists of:

- A finite set N of non-terminal symbols.
- A finite set Σ of terminal symbols that is disjoint from N.
- A finite set P of parsing rules.
- An expression e_S termed the starting expression.

Each parsing rule in P has the form $A \leftarrow e$, where A is a nonterminal symbol and e is a parsing expression.

To illustrate this, we have a look at the following imaginary PEG for an email address.

$$\begin{aligned} \langle\text{name}\rangle &\leftarrow [\mathtt{a}-\mathtt{z}]+\ (\texttt{"."}\ [\mathtt{a}-\mathtt{z}]+)^* \\ \langle\text{host}\rangle &\leftarrow [\mathtt{a}-\mathtt{z}]+\ \texttt{"."}\ (\texttt{"com"}/\texttt{"org"}/\texttt{"net"}) \\ \langle\text{email}\rangle &\leftarrow \langle\text{name}\rangle\ \texttt{"@"}\ \langle\text{host}\rangle \end{aligned}$$

The symbols in angle brackets are the non-terminal symbols. The quoted strings and expressions in square brackets are terminal symbols. The entry point e_S is the rule named email (although the entry point is not specially marked). The present grammar translates into natural language rather nicely. We start at the entry point, the email rule. The email rule tells us that an email is a name, followed by a literal @, followed by a host. The symbols name and host are non-terminal, so they can't be parsed without furhter information so we have to resolve them. A name is specified as one or more characters in the range a to z, followed by zero or more groups of a literal dot, followed by one or more characters a to z. A host is one or more characters a to z, followed by a literal dot, followed by one of the literals com, org, or net. Here the range of characters and the string literals are terminal symbols, because they can be parsed from the input without further information.

As a little teaser, we will have a look how the above grammar translated into LPEG.

```lua
local name  = R"az"^1 * (P"." * R"az"^1)^0
local host  = R"az"^1 * P"."
            * (P"com" + P"org" + P"net")
local email = name * P"@" * host
```

We can already see that there is sort of a mapping to translate PEG into LPEG, but at first sight it seems like this translation is almost 1:1. We will learn what the symbols mean in the next section.

4 Basic parsers

LPEG provides some basic parsers to make our life a little easier. These map the terminal symbols in the grammer. Here they are with examples:

- `lpeg.P(string)` Matches the provided `string` exactly. This matches "hello" but not "world":

  ```lua
  lpeg.P("hello")
  ```

- `lpeg.P(n)` Matches exactly `n` characters. To match any single character we could use

  ```lua
  lpeg.P(1)
  ```

  There is a special character which is not mapped by any encoding which is the end of input. In LPEG there is a special rule for it:

  ```lua
  lpeg.P(-1)
  ```

- `lpeg.S(string)` Matches any character in `string` (Set). To match any whitespace we use:

  ```lua
  lpeg.S(" \t\r\n")
  ```

- `lpeg.R("xy")` Matches any character between `x` and `y` (Range). Matching any digit is done using

  ```lua
  lpeg.R("09")
  ```

  To match any character in the ASCII range we can combine lowercase and uppercase letters:

  ```lua
  lpeg.R("az", "AZ")
  ```

It is tedious to constantly type the `lpeg.` prefix which is why we omit it from now on. This can be achieved by assigning the members of the `lpeg` table to the corresponding variables.

```lua
local lpeg = require"lpeg"
local P, R = lpeg.P, lpeg.R -- etc.
```

5 Parsing expressions

By themselves these basic parsers are rather useless. The real power of LPEG comes from the ability to arbitrarily combine parsers. This is achieved by means of parsing expressions. The available parsing expressions are listed in table 1. Below I show some examples where the quoted strings in the comments represent input that is parsed successfully by the associated parser unless stated otherwise.

Description	PEG	LPEG
Sequence	$e_1 e_2$	`patt1 * patt2`
Ordered choice	$e_1 \| e_2$	`patt1 + patt2`
Zero or more	$e*$	`patt^0`
One or more	$e+$	`patt^1`
Optional	$e?$	`patt^-1`
And predicate	$\&e$	`#patt`
Not predicate	$!e$	`-patt`
Difference		`patt1 - patt2`

Table 1 Available parsing expressions in LPEG with their name and corresponding symbol in PEG. Note that the difference operation is an extension by LPEG and not available in PEG.

- Sequence: This implements the "followed by" operation, i.e. the parser matches only if the first pattern is followed directly by the second pattern.

```
P"pizza" * R"09" -- "pizza4"
P(1) * P":" * R"09" -- "a:9"
```

- Ordered choice: The ordered choice parses the first operand first and only if it fails continues to the next operand. So the ordering is indeed important.

```
R"az" + R"09" + R".,;:?!"
-- "a", "9", ";"
-- "+" fails to parse
```

- Zero or more, one or more, and optional: These are all captured by the same contruct in LPEG, the exponentiation operator. A positive exponent n parses at least n occurences of the pattern, a negative exponent $-n$ parses at most n occurences of the pattern.

```
R"az"^0 + R"09"^1
-- "z86", "abcde99", "99"
R"az"^1 + R"09"^1
-- "z86", "abcde99"
-- "99" fails to parse
R"az"^-1 + R"09"^1
-- "z86", "99"
-- "abcde99" fails to parse
```

- And predicate and not predicate: These two expressions are special in that they don't consume any input. For the not predicate this is obvious because it only matches if the parser it negates does not match.

```
R"09"^1 * #P";"
-- "86;"
-- "99" fails to parse
P"for" * -(R"az"^1)
-- "for()"
-- "forty" fails to parse
```

- Difference: The difference expression will match the first operand only if the second operand does not match. This can be useful to match C style comments which collect everything between the first `/*` and the first `*/`. However, care must be taken that the second operand cannot successfully parse parts of the first operand. If that is the case, the resulting rule will never match.

```
P"/*" * (1 - P"*/")^0 * P"*/"
-- "/* comment */"
P"helloworld" - P"hell"
-- will never match!
```

6 Simple examples

Let us study a simple example which parses two words separated by a space. The LPEG grammar is stored in the variable `rule`. The rest of the example shows the boilerplate that is necessary.

```
local lpeg = require"lpeg"
local P, R = lpeg.P, lpeg.R

local input = "cosmic pizza"

local rule = R"az"^1 * P" " * R"az"^1
print(rule:match(input) .. " of " .. #input)
```

This will print on the terminal "13 of 12" because all the input has been consumed and the parser stopped at the end of input which is the 13th "character" in this string. As we can see the function `rule:match` parses a given input string using a given parser and returns the number of characters parsed. Another way to invoke a parse is using `lpeg.match(rule, input)`, which is equivalent to `rule:match(input)`.

The next example will be slightly more complicated. We will parse a comma-separated list of colon-separated key-value pairs.

```
local input = [[foo : bar ,
gorp : smart ,
falcou : "crazy frenchman" ,
name : sam]]
```

The double square brackets denote one of Lua's long strings, which can have embedded newlines. The colons and commas that separate keys and values, and entries, respectively, are surrounded by whitespace. To match all possible optional whitespace we use the set parser and the optional expression.

```
local ws = S" \t\r\n"^0
```

With this the specification for the key field is simply one or more letters or digits surrounded by optional whitespace.

```
local name = ws * R("az", "AZ", "09")^1 * ws
```

The value field on the other hand can have either the same specification as the key field, which does not allow embedded whitespaces, or it can be a quoted string, which allows anything between the quotes. To this end we specify the grammar for a quoted string, which is simply the double quotes character, followed by anything that is not double quotes, followed by double quotes. The whole thing may be surrounded by optional whitespace.

```
local quote =
  ws * P'"' *  (1 - P'"')^0 * P'"' * ws
```

Therefore an entry in the key-value list is a `name`, followed by a colon, followed by either a `quote` or a `name`, followed by at most one comma. The whole key-value list is of course just any number of entries, so we apply the zero or more expression to the aforementioned rule.

```
local keyval =
  (name * P":" * (quote + name) * P","^-1)^0
```

Matching the rule against the input in the same way as the previous example gives "67 of 66".

7 Grammars

The literal parser `P` has a second function. If its argument is a table, the table is processed as a *grammar*. The table has the following layout:

```
P{"<entry point>",
    <non-terminal> = <parsing expression>
    ...
}
```

The string "entry point" is the name of the rule to be processed first. Afterwards the rules are listed in the same manner as they were assigned to variables in the previous example. To refer to non-terminal symbols from within the grammar, the `lpeg.V` funtion is used. Collecting the aforementioned rules into a grammar could look like this:

```
local rule = P{"keyval",
  keyval =
    (V"name" * P":" * (V"quote" + V"name")
      * P","^-1)^0,
  name =
    V"ws" * R("az", "AZ", "09")^1 * V"ws",
  quote =
    V"ws" * P'"' *  (1 - P'"')^0 * P'"'
      * V"ws",
  ws = S" \t\r\n"^0,
}
```

It becomes a little more verbose because names of non-terminal symbols have to be wrapped in `V"..."`. That is why I personally do not normally include general-purpose rules like the `ws` rule in the example into the grammar, because chances are high I want to use it elsewhere again. The level of verbosity might seem like a disadvantage but the encapsulation is much better that way. It also makes it much easier to define recursive rules, as we will see later.

8 Attributes

In the previous section we have parsed some inputs and confirmed their vailidity by a successful parse and we received the length of the parsed input. An important question remains, how do we extract information from the input? When a parse is successful, the basic parsers synthesize the value they encountered which I am going to call their *attribute*. These attributes can be extracted using LPEG's capture operations.

The simplest capture operation is `lpeg.C(patt)` which simply returns the match of `patt`. Here we parse a strip of only lowercase letters and print the result.

```
local rule = C(R"az"^1)
print(rule:match"pizza") -- pizza
```

Another, very powerful capture is the table capture `lpeg.Ct(patt)` which returns a table with all captures from `patt`. This allows us to write a very simple parser for comma separated values (CSV) in only three lines.

```
local cell = C((1 - P"," - P"\n")^0)
local row = Ct(cell * (P"," * cell)^0)
local csv = Ct(row * (P"\n" * row)^0)

local t = csv:match[[a,b,c
d,e,f
g,,h]]
```

The variable `t` now holds the table representing the CSV file and we can access the elements by `t[<row>][<column>]`, e.g. to access the "e" in the middle of the table we can use `t[2][2]`.

There are two more captures which I think are worth mentioning, the grouping capture and the folding capture. The grouping capture `lpeg.Cg(patt [, name])` groups the values produced by `patt`, optionally tagged with `name`. The grouping capture is mostly used in conjunction with the folding capture `lpeg.Cf(patt, func)` which folds the captures from `patt` with the functions `func`. The most common application is parsing of key-value lists. The key and the value are captured independently at first but are then grouped together. Finally they are folded together with an empty table capture.

```
local key = C(R"az"^1)
local val = C(R"09"^1)

local kv = Cg(key * P":" * val) * P","^-1
local kvlist = Cf(Ct"" * kv^0, rawset)

kvlist:match"foo:1,bar:2"
```

9 Actually useful parsers

Now that we know how to parse input and extract data, we can go ahead and start constructing parsers that are acutally useful. We will now construct a parser for floating point numbers. The parser presented here has some limitations. It doesn't handle an integer part that only contains a sign, i.e. `-.1` will not parse. It also doesn't handle hexadecimal, octal, or binary literals. This is left as an exercise to the reader. To construct a possible grammar for floating point numbers, let's take a look at what they look like.

$$\underbrace{\overbrace{\texttt{+123}}^{\text{integer part}}\overbrace{\texttt{.45678}}^{\text{fractional part}}}_{\text{mantissa}}\underbrace{\texttt{e-90}}_{\text{exponent}}$$

With that we formulate the first rule in our grammar, namely

```
number = (V"int" * V"frac"^-1 * V"exp"^-1)
         / tonumber,
```

i.e. a number has an integer part, followed by an optional fractional part, followed by an optional exponent. The division by `number` that we see here is called a *semantic action*. A semantic action is applied to the result of the parser *ad-hoc*. In general it is a bad idea to use semantic actions, because they don't fit into the concept of recursive parsing and introduce additional state to keep track of. Nevertheless there are some cases when semantic actions are useful, like in this case, where we know that what we just parsed is a number and we merely convert the resulting string into Lua's number type.

Now let's parse the integer part. Here I show all the rules that go into it at once.

```
int = V"sign"^-1 * (R"19" * V"digits"
                               + V"digit"),
sign = S"+-",
digit = R"09",
digits = V"digit" * V"digits" + V"digit",
```

So the integer part is an optional sign, followed by a number between 1 and 9, followed by more digits or just a single digit. A sign is of course just the character `+` or `-`. A single digit is just a number between 0 and 9. The `digits` rule is recursive, because many digits are either a single digit followed by more digits, or just that single digit.

Next is the fractional part, which is very easy. It is just a period followed by digits.

```
frac = P"." * V"digits",
```

Last the exponential part, which is also simple. It is either a lower- or uppercase E, followed by an optional sign, followed by digits.

```
exp = S"eE" * V"sign"^-1 * V"digits",
```

Now let's check this parser with some test input. We expect the result to be the same number that we input and we expect it to be of Lua type `number`.

```
local x = number:match("+123.45678e-90")
print(x .. " " .. type(x))
```

Output: 1.2345678e-88 number

The full code of the number parser is given as part of the JSON parser in the Appendix in lines 5–14.

10 Complex Data Formats: JSON

JSON is short for JavaScript Object Notation and is a lightweight data format that is easy to read and write for both humans and machines. JSON knows six different data types of which two are collections. These are `null`, `bool`, `string`, `number`, `array`, and `object`. This maps nicely to Lua where `null` maps to `nil`, `bool` maps to `boolean`, `string` and `number` map to their eponymous counterparts, and `array` and `object` both map to Lua's `table` type.

On the top level there is always an object, i.e. a JSON file looks roughly like this [5]

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New",
       "onclick": "CreateNewDoc()"},
      {"value": "Open",
       "onclick": "OpenDoc()"},
      {"value": "Close",
       "onclick": "CloseDoc()"}
    ]
  }
}}
```


Before we begin writing a parser for this, we introduce a few general purpose parsers first, which are also not part of the grammar.

```
local ws = S" \t\n\r"^0
```

This rule matches zero or more whitespace characters, where whitespace characters are space, tab, newline and carrige return.

```
local lit = function(str)
    return ws * P(str) * ws
end
```

This function returns a rule that matches a literal string surrounded by optional whitespace. This is useful to match keywords.

```
local attr = function(str,attr)
    return ws * P(str) / function()
            return attr
        end * ws
end
```

This function returns an extension of the previous rule, in that it matches a literal string and if it matched returns an attribute using a semantic action. This is very useful for parsing a string but returning something unrelated, e.g. the `null` value of JSON will be represented by Lua's `nil`.

As mentioned before, at the top level a JSON file expects an object, so this will be the entry point.

```
local json = P{"object",
```

As discussed before JSON supports different kinds of values, so we want to map these in our parsing grammar.

```
value =
    V"null_value" +
    V"bool_value" +
    V"string_value" +
    V"number_value" +
    V"array" +
    V"object",
```

So a `value` is any of the value types defined by the JSON format. That was easy, but now we have to define what these values are and how to parse them. We begin with the easiest ones, the `null` and `bool` values:

```
null_value = attr("null", nil),
bool_value = attr("true", true)
           + attr("false", false),
```

These two types are defined entirely by keyword matching. We use the `attr` function to return a suitable Lua value. Next we define how to parse strings:

```
string_value = ws * P'"'
             * C((P'\\"' + 1 - P'"')^0)
             * P'"' * ws,
```

A string may be surrounded by whitespace and is enclosed in double quotes. Inside the double quotes we can use any character that is not the double quote, unless we escape it \". The value of the string without surrounding quotes is captured. To parse number values, we will reuse the number parser defined in the previous section

```
number_value = ws * number * ws,
```

This concludes the parsing of all the simple datatypes and we move on to the aggregate types, starting with the array.

```
array = lit"["
      * Ct((V"value" * lit","^-1)^0)
      * lit"]",
```

An array is simply a comma-separated list of values that is enclosed in square brackets. The list is captured as a Lua table. The final and most complicated type to parse is the object.

```
member_pair = Cg(V"string_value" * lit":"
                 * V"value") * lit","^-1,
object = lit"{"
       * Cf(Ct"" * V"member_pair"^0, rawset)
       * lit"}"
```

An object is a comma-separated list of key-value pairs enclosed in curly braces, where a key-value pair is a string, followed by a colon, followed by a value. To pack this into a Lua table, we use the grouping and folding captures that we discussed before. This concludes the JSON grammar.

```
}
```

The full code of the parser is given in the Appendix with a little nicer formatting. Now we can go ahead an parse JSON files.

```
local result = json:match(input)
```

The variable `result` will hold a Lua table which can be indexed in a natural way. For example, if we had parsed the JSON example given in the beginning of this section, we could use

```
print(result.menu.popup.menuitem[2].onclick)
-- OpenDoc()
```

This way we could write configuration files for our document, parse them on-the-fly when firing up LuaTEX, and configure the style and content according to the specifications.

11 Summary and Outlook

Parsing even complex data formats like JSON is relatively easy using LPEG. A possible next step would be to parsing the LuaTEX input file in the `process_input_buffer` callback and replace templates in the file with values from JSON.

References

[1] R. Ierusalimschy, A text pattern-matching tool based on Parsing Expression Grammars, *Software: Practice and Experience* **39**(3), 221–258 (2009).

[2] T. Hoekwater, LuaTEX, *TUGboat* **28**(3), 312–313 (2007).

[3] M. Caisse, *Using Spirit X3 to Write Parsers*, https://www.youtube.com/watch?v=xSBWklPLRvw (2015). (CppCon)

[4] Wikipedia, *Parsing expression grammar*, https://en.wikipedia.org/wiki/Parsing_expression_grammar (online). (Accessed on July 15, 2019)

[5] D. Crockford, *JSON Example*, https://json.org/example.html (online). (Accessed on July 15, 2019)

Henri Menke
9016 Dunedin
New Zealand
`henrimenke@gmail.com`

12 Appendix: Full code listing of the JSON parser

```
local lpeg = require"lpeg"
local C, Cf, Cg, Ct, P, R, S, V =
    lpeg.C, lpeg.Cf, lpeg.Cg, lpeg.Ct, lpeg.P, lpeg.R, lpeg.S, lpeg.V

-- number parsing
local number = P{"number",
    number = (V"int" * V"frac"^-1 * V"exp"^-1) / tonumber,
    int = V"sign"^-1 * (R"19" * V"digits" + V"digit"),
    sign = S"+-",
    digit = R"09",
    digits = V"digit" * V"digits" + V"digit",
    frac = P"." * V"digits",
    exp = S"eE" * V"sign"^-1 * V"digits",
}

-- optional whitespace
local ws = S" \t\n\r"^0

-- match a literal string surrounded by whitespace
local lit = function(str)
    return ws * P(str) * ws
end

-- match a literal string and synthesize an attribute
local attr = function(str,attr)
    return ws * P(str) / function() return attr end * ws
end

-- JSON grammar
local json = P{
    "object",

    value =
        V"null_value" +
        V"bool_value" +
        V"string_value" +
        V"number_value" +
        V"array" +
        V"object",

    null_value =
        attr("null", nil),

    bool_value =
        attr("true", true) + attr("false", false),

    string_value =
        ws * P'"' * C((P'\\"' + 1 - P'"')^0) * P'"' * ws,

    number_value =
        ws * number * ws,

    array =
```

```
        lit"[" * Ct((V"value" * lit","^-1)^0) * lit"]",

    member_pair =
        Cg(V"string_value" * lit":" * V"value") * lit","^-1,

    object =
        lit"{" * Cf(Ct"" * V"member_pair"^0, rawset) * lit"}"
}
```