

The luakeyval Module

Version: 0.2, 2026-06-25

Udi Fogiel, 2025, 2026

The luakeyval LuTeX module is a minimal key/value parser for LuaTeX formats based on `token.scan_key_cs`. As such, the keyword parsing is similar to how TeX parses keywords in special primitives like `\hrule`.

Unlike TeX's scanner, luakeyval scans key/val pairs inside braces. This avoids the need to append a `\relax` to the key/val list, which would otherwise be required to stop TeX from scanning too far and potentially creating expansion problems.

1 Usage

The module provides the `process` function, which accepts a table of keys, and a table of error messages.

Each key in the table should have a table of parameters as a value. The possible parameters are

- **scanner:** a function that will scan the value of the key from TeX to Lua. The value will usually be a function from LuaTeX's token library, but you can use your own.
- **args:** a table of arguments that will be passed to the scanner function.
- **default:** default: a value returned if the key is not followed by a `=`.
- **func:** a function that will be executed each time the key appears.

None of the parameters is mandatory, but a key must have at least one of `default` or `scanner`.

The error messages table can have the following entries

- **error1:** a message that will be displayed if something went wrong while processing a key/val list.
- **error2:** a message that will be displayed after `error1` in case a user press the H key for more information.
- **value_required:** a message that will be displayed if no value given to a key (when there is no `=` after the key) and the key does not have a default value.
- **value_forbidden:** a message that will be displayed if a key was given a value and the key does not have a `scanner` function.

2 Example

The following code defines the `\coloredrule` macro, which accepts the keys `width`, `height`, `depth` and `color`, and prints a colored rule according to the values given.

```
local keyval = require('luakeyval')
local rule_keys = {
  width = {scanner = token.scan_dimen, default = tex.sp('1cm')},
  height = {scanner = token.scan_dimen, default = tex.sp('1ex')},
  depth = {scanner = token.scan_dimen, default = 0},
  color = {scanner = token.scan_string},
}

local messages = {
```

```

    error1 = "colored rule: Wrong syntax in \\coloredrule",
    value_forbidden = 'colored rule: The key "%s" does not accept a value',
    value_required = 'colored rule: The key "%s" requires a value',
}

local function make_rule()
    local opts = keyval.process(rule_keys, messages)
    local rule = node.new('rule')
    rule.width = opts.width or tex.sp('1cm')
    rule.height = opts.height or tex.sp('1ex')
    rule.depth = opts.depth or 0
    local color_start = node.new('whatsit', 'pdf_literal')
    color_start.mode = 0
    color_start.data = opts.color .. " rg"
    local color_end = node.new('whatsit', 'pdf_literal')
    color_end.mode = 0
    color_end.data = '0 g'
    rule.next = color_end
    color_start.next = rule
    rule = color_start
    node.write(rule)
end

token.set_lua('coloredrule', #lua.get_functions_table() +1, 'protected')
lua.get_functions_table()[#lua.get_functions_table()+1] = make_rule

```

Now `\coloredrule{width = 10pt height = 5pt color={1 0 0}}` prints ■

3 Important Limitations

Since the key/val parser is only a minimal layer on top of `token.scan_keyword`, key/val pairs should be separated with spaces, not commas, and avoid defining a key that is a prefix of another key (unless you control the scanning order, see [Section 4](#)).

4 Scanning Order

The `process` function loops over the keys table using LuaTeX's `token.scan_key_cs`. Since Lua tables do not guarantee key order when iterating with `pairs()`, the scanning order is not deterministic if you just provide a plain table.

This is important when one key is a prefix of another (for example, `long` and `longer`). If the shorter key is checked first, it may match part of the input and leave extra characters unprocessed, leading to errors.

To control the scanning order explicitly, pass a third argument to `process`: a list of keys in the exact order they should be scanned. This also allows scanning only a subset of keys if desired.

```

local keys = {
    long = {scanner = token.scan_string},
    longer = {scanner = token.scan_string},
}

-- Default scan (order not guaranteed)
local opts = keyval.process(keys, messages)
-- input: {longer="test"} may incorrectly match 'long' first

-- Controlled scan with explicit order
local order = {"longer", "long"}
local opts = keyval.process(keys, messages, order)

```

5 Implementation

The full Lua implementation is shown below for reference.

luakeyval.lua

```
1 -- luakeyval Version: 0.2, 2026-06-25
2
3 local put_next = token.unchecked_put_next
4 local get_next = token.get_next
5 local scan_toks = token.scan_toks
6 local scan_keyword = token.scan_keyword_cs
7
8 local texerror, utfchar = tex.error, utf8.char
9 local format = string.format
10
11 -- local relax = token.new(token.biggest_char() + 1)
12 local relax
13 do
14 -- initialization of the new primitives.
15 local prefix = '@lua~key&val_' -- unlikely prefix...
16 while token.is_defined(prefix .. 'relax') do
17 prefix = prefix .. '@lua~key&val_'
18 end
19 tex.enableprimitives(prefix,{'relax'})
20 -- Now we create new tokens with the meaning of
21 -- the primitive.
22 relax = token.create(prefix .. 'relax')
23 end
24
25 local function check_delimiter(error1, error2, key)
26 local tok = get_next()
27 if tok.tok ~= relax.tok then
28 local tok_name = tok.csname or utfchar(tok.mode)
29 texerror(format(error1, key, tok_name),{format(error2, key, tok_name)})
30 put_next({tok})
31 end
32 end
33
34 local unpack = table.unpack
35 local function process_keys(keys, messages, order)
36 assert(type(keys) == 'table')
37 local matched, vals, curr_key = true, { }
38 messages = messages or { }
39 local value_forbidden = messages.value_forbidden
40 or 'luakeyval: The key "%s" does not accept a value'
41 local value_required = messages.value_required
42 or 'luakeyval: The key "%s" requires a value'
43 local error1 = messages.error1
44 or 'luakeyval: Wrong syntax when processing keys'
45 local error2 = messages.error2
46 or 'luakeyval: The last scanned key was "%s".\nUnexpected token "%s" encountered.'
47 local key_list = { }
48 if order then
49 for _, k in ipairs(order) do
50 key_list[#key_list+1] = k
51 end
52 else
53 for k in pairs(keys) do
54 key_list[#key_list+1] = k
55 end
56 end
57 local toks = scan_toks()
58 toks[#toks+1] = relax
59 put_next(toks)
60 while matched do
61 matched = false
62 for _, key in ipairs(key_list) do
63 local param = keys[key]
64 if scan_keyword(key) then
65 matched = true
```

```

66         curr_key = key
67         local args = param.args or { }
68         local scanner = param.scanner
69         local val
70         if scan_keyword('=') then
71             if scanner then
72                 val = scanner(unpack(args))
73             else
74                 texerror(format(value_forbidden, key))
75             end
76         else
77             val = param.default
78             if val == nil then
79                 texerror(format(value_required, key))
80             end
81         end
82         local func = param.func
83         if func then func(key, val) end
84         vals[key] = val
85         break
86     end
87 end
88 end
89 check_delimiter(error1, error2, curr_key or '<none>')
90 return vals
91 end
92
93 local function scan_choice(...)
94     local choices = {...}
95     for _, choice in ipairs(choices) do
96         if scan_keyword(choice) then
97             return choice
98         end
99     end
100     return nil
101 end
102
103 local function scan_bool()
104     if scan_keyword('true') then
105         return true
106     elseif scan_keyword('false') then
107         return false
108     end
109     return nil
110 end
111
112 return {
113     process = process_keys,
114     choices = scan_choice,
115     bool = scan_bool,
116 }

```