The luakeys package

Josef Friedrich
josef@friedrich.rocks
github.com/Josef-Friedrich/luakeys

v0.4 from 2021/12/31

```lua
local luakeys = require('luakeys')
local kv = luakeys.parse('level1={level2={level3={dim=1cm,bool=true,num=-
↪  0.001,str=lua}}}')
luakeys.print(kv)
```

Result:

```
{
  ['level1'] = {
    ['level2'] = {
      ['level3'] = {
        ['dim'] = 1864679,
        ['bool'] = true,
        ['num'] = -0.001
        ['str'] = 'lua',
      }
    }
  }
}
```

Contents

1 Introduction

`luakeys` is a Lua module that can parse key-value options like the TEX packages keyval, kvsetkeys, kvoptions, xkeyval, pgfkeys etc. do. `luakeys`, however, accompilshes this task entirely, by using the Lua language and doesn't rely on TEX. Therefore this package can only be used with the TEX engine LuaTEX. Since `luakeys` uses LPeg, the parsing mechanism should be pretty robust.

The TUGboat article "Implementing key–value input: An introduction" (Volume 30 (2009), No. 1) by Joseph Wright and Christian Feuersänger gives a good overview of the available key-value packages.

This package would not be possible without the article Parsing complex data formats in LuaTEX with LPEG (Volume 40 (2019), No. 2).

2 Usage

2.1 Using the Lua module `luakeys.lua`

The core functionality of this package is realized in Lua. So you can use `luakeys` without using the wrapper TEX files `luakeys.sty` and `luakeys.tex`.

```
\documentclass{article}
\directlua{
  luakeys = require('luakeys')
}

\newcommand{\helloworld}[2][]{
  \directlua{
    local keys = luakeys.parse('\luaescapestring{\unexpanded{#1}}')
    luakeys.print(keys)
    local marg = '#2'
    tex.print(keys.greeting .. ', ' .. marg .. keys.punctuation)
  }
}
\begin{document}
\helloworld[greeting=hello,punctuation=!]{world}
\end{document}
```

2.2 Using the LuaLATEX wrapper `luakeys.sty`

The supplied LuaLATEX file is quite small:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{luakeys}
\directlua{luakeys = require('luakeys')}
```

It loads the Lua module into the global variable `luakeys`.

```
\documentclass{article}
\usepackage{luakeys}

\begin{document}
  \directlua{
    local keys = luakeys.parse('one,two,three')
    tex.print(keys[1])
    tex.print(keys[2])
    tex.print(keys[3])
  }
\end{document}
```

2.3 Using the plain LuaTEX wrapper `luakeys.tex`

Even smaller is the file `luakeys.tex`. It consists of only one line:

```
\directlua{luakeys = require('luakeys')}
```

It does the same as the LuaLATEX wrapper and loads the Lua module `luakeys.lua` into the global variable `luakeys`.

```
\input luakeys.tex

\directlua{
  local keys = luakeys.parse('one,two,three')
  tex.print(keys[1])
  tex.print(keys[2])
  tex.print(keys[3])
}
\bye
```

3 Syntax of the recognized key-value format

3.1 A attempt to put the syntax into words

A key-value pair is definied by an equal sign (`key=value`). Several key-value pairs or values without keys are lined up with commas (`key=value,value`) and build a key-value list. Curly brackets can be used to create a recursive data structure of nested key-value lists (`level1={level2={key=value,value}}`).

3.2 An (incomplete) attempt to put the syntax into the Extended Backus-Naur Form

⟨*list*⟩ ::= { ⟨*list-item*⟩ }

⟨*list-container*⟩ ::= '`{`' ⟨*list*⟩ '`}`'

⟨*list-item*⟩ ::= (⟨*list-container*⟩ | ⟨*key-value-pair*⟩ | ⟨*value*⟩) ['`,`']

⟨*key-value-pair*⟩ ::= ⟨*value*⟩ '`=`' (⟨*list-container*⟩ | ⟨*value*⟩)

⟨*value*⟩ ::= ⟨*boolean*⟩
| ⟨*dimension*⟩
| ⟨*number*⟩
| ⟨*string-quoted*⟩
| ⟨*string-unquoted*⟩

⟨*sign*⟩ ::= '`-`' | '`+`'

⟨*integer*⟩ ::= '`0`' | '`1`' | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`'

⟨*unit*⟩ ::= '`bp`' | '`BP`'
| '`cc`' | '`CC`'
| '`cm`' | '`CM`'
| '`dd`' | '`DD`'
| '`em`' | '`EM`'
| '`ex`' | '`EX`'
| '`in`' | '`IN`'

| ‘mm’ | ‘MM’
| ‘nc’ | ‘NC’
| ‘nd’ | ‘ND’
| ‘pc’ | ‘PC’
| ‘pt’ | ‘PT’
| ‘sp’ | ‘SP’

⟨*boolean*⟩ ::= ⟨*boolean-true*⟩ | ⟨*boolean-false*⟩

⟨*boolean-true*⟩ ::= ‘`true`’ | ‘`TRUE`’ | ‘`True`’

⟨*boolean-false*⟩ ::= ‘`false`’ | ‘`FALSE`’ | ‘`False`’

... to be continued

3.3 Recognized data types

3.3.1 boolean

The strings `true`, `TRUE` and `True` are converted into Lua's boolean type `true`, the strings `false`, `FALSE` and `False` into `false`.

```
\luakeysdebug{
  lower case true = true,
  upper case true = TRUE,
  title case true = True
  lower case false = false,
  upper case false = FALSE,
  title case false = False,
}
```

```
{
  ['lower case true'] = true,
  ['upper case true'] = true,
  ['title case true'] = true,
  ['lower case false'] = false,
  ['upper case false'] = false
  ['title case false'] = false,
}
```

3.3.2 number

```
\luakeysdebug{
  num1 = 4,
  num2 = -4,
  num3 = 0.4
}
```

```
{
  ['num1'] = 4,
  ['num2'] = -4,
  ['num3'] = 0.4
}
```

3.3.3 dimension

`luakeys` detects TEX dimensions and automatically converts the dimensions into scaled points using the function `tex.sp(dim)`. Use the option `convert_dimensions` of the function `parse(kv_string, options)` to disalbe the automatic conversion.

```
local result = parse('dim=1cm', {
  convert_dimensions = false,
})
```

If you want to convert a scale point into a unit string you can use the module lualibs-util-dim.lua.

```
\begin{luacode}
require('lualibs')
tex.print(number.todimen(tex.sp('1cm'), 'cm', '%0.0F%s'))
\end{luacode}
```

Unit name	Description
bp	big point
cc	cicero
cm	centimeter
dd	didot
em	horizontal measure of M
ex	vertical measure of x
in	inch
mm	milimeter
nc	new cicero
nd	new didot
pc	pica
pt	point
sp	scaledpoint

```
\luakeysdebug{
  bp = 1bp,
  cc = 1cc,
  cm = 1cm,
  dd = 1dd,
  em = 1em,
  ex = 1ex,
  in = 1in,
  mm = 1mm,
  nc = 1nc,
  nd = 1nd,
  pc = 1pc,
  pt = 1pt,
  sp = 1sp,
}
```

```
{
  ['bp'] = 65781,
  ['cc'] = 841489,
  ['cm'] = 1864679,
  ['dd'] = 70124,
  ['em'] = 655360,
  ['ex'] = 282460,
  ['in'] = 4736286,
  ['mm'] = 186467,
  ['nc'] = 839105,
  ['nd'] = 69925,
  ['pc'] = 786432,
  ['pt'] = 65536,
  ['sp'] = 1,
}
```

3.3.4 string

There are two ways to specify strings: With or without quotes. If the text have to contain commas or equal signs, then double quotation marks must be used.

```
\luakeysdebug{
  without quotes = no commas and
  ↪  equal signs are allowed,
  with double quotes = ", and = are
  ↪  allowed",
}
```

```
{
  ['without quotes'] = 'no commas
  ↪  and equal signs are allowed',
  ['with double quotes'] = ', and =
  ↪  are allowed',
}
```

3.3.5 Standalone values

Standalone values are values without a key. They are converted into an array. In Lua an array is a table with numeric indexes (The first index is 1).

```
\luakeysdebug{one,two,three}
```

```
{ 'one', 'two', 'three' }
```

is equivalent to

```
{
  [1] = 'one',
  [2] = 'two',
  [3] = 'three',
}
```

All recognized data types can be used as standalone values.

```
\luakeysdebug{one,2,3cm}
```

```
{ 'one', 2, 5594039 }
```

4 Exported functions of the Lua module `luakeys.lua`

To learn more about the individual functions (local functions), please read the source code documentation, which was created with LDoc. The Lua module exports this functions:

```
local luakeys = require('luakeys')
local parse = luakeys.parse
local render = luakeys.render
--local print = luakeys.print -- That would overwrite the built-in Lua function
local save = luakeys.save
local get = luakeys.get
```

4.1 `parse(kv_string, options): table`

The function `parse(input_string, options)` is the main method of this module. It parses a key-value string into a Lua table.

```
\newcommand{\mykeyvalcmd}[1][]{
  \directlua{
    result = luakeys.parse('#1')
    luakeys.print(result)
  }
  #2
}
\mykeyvalcmd[one=1]{test}
```

In plain TEX:

```
\def\mykeyvalcommand#1{
  \directlua{
    result = luakeys.parse('#1')
    luakeys.print(result)
  }
}
\mykeyvalcmd{one=1}
```

The function can be called with a options table. This two options are supported.

```
local result = parse('one,two,three', {
  convert_dimensions = false,
  unpack_single_array_value = false
})
```

4.2 `render(tbl): string`

The function `render(tbl)` reverses the function `parse(kv_string)`. It takes a Lua table and converts this table into a key-value string. The resulting string usually has a different order as the input table.

```
result = luakeys.parse('one=1,two=2,tree=3,')
print(luakeys.render(result))
--- one=1,two=2,tree=3,
--- or:
--- two=2,one=1,tree=3,
--- or:
--- ...
```

In Lua only tables with 1-based consecutive integer keys (a.k.a. array tables) can be parsed in order.

```
result = luakeys.parse('one,two,three')
print(luakeys.render(result))
--- one,two,three, (always)
```

4.3 print(tbl): void

The function `print(tbl)` pretty prints a Lua table to standard output (stdout). It is a utility function that can be used to debug and inspect the resulting Lua table of the function `parse`. You have to compile your TEX document in a console to see the terminal output.

```
result = luakeys.parse('level1={level2={key=value}}')
luakeys.print(result)
```

The output should look like this:

```
{
  ['level1'] = {
    ['level2'] = {
      ['key'] = 'value',
  },
}
```

4.4 save(identifier, result): void

The function `save(identifier, result)` saves a result (a table from a previous run of `parse`) under an identifier. Therefore, it is not necessary to pollute the global namespace to store results for the later usage.

4.5 get(identifier): table

The function `get(identifier)` retrieves a saved result from the result store.

5 Debug packages

Two small debug packages are included in `luakeys`. One debug package can be used in LaTeX (luakeys-debug.sty) and one can be used in plain TeX (luakeys-debug.tex). Both packages provide only one command: `\luakeysdebug{kv-string}`

```
\luakeysdebug{one,two,three}
```

Then the following output should appear in the document:

```
{
  [1] = 'one',
  [2] = 'two',
  [3] = 'three',
}
```

5.1 For plain TeX: luakeys-debug.tex

An example of how to use the command in plain TeX:

```
\input luakeys-debug.tex
\luakeysdebug{one,two,three}
\bye
```

5.2 For LaTeX: luakeys-debug.sty

An example of how to use the command in LaTeX:

```
\documentclass{article}
\usepackage{luakeys-debug}
\begin{document}
\luakeysdebug[
  unpack single array values=false,
  convert dimensions=false
]{one,two,three}
\end{document}
```

6 Implementation

6.1 luakeys.lua

```lua
-- luakeys.lua
-- Copyright 2021-2022 Josef Friedrich
--
-- This work may be distributed and/or modified under the
-- conditions of the LaTeX Project Public License, either version 1.3c
-- of this license or (at your option) any later version.
-- The latest version of this license is in
--   http://www.latex-project.org/lppl.txt
-- and version 1.3c or later is part of all distributions of LaTeX
-- version 2008/05/04 or later.
--
-- This work has the LPPL maintenance status `maintained'.
--
-- The Current Maintainer of this work is Josef Friedrich.
--
-- This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
-- luakeys-debug.sty and luakeys-debug.tex.

--- A key-value parser written with Lpeg.
--
-- Explanations of some LPeg notation forms:
--
-- * `patt ^ 0` = `expression *`
-- * `patt ^ 1` = `expression +`
-- * `patt ^ -1` = `expression ?`
-- * `patt1 * patt2` = `expression1 expression2`: Sequence
-- * `patt1 + patt2` = `expression1 / expression2`: Ordered choice
--
-- * [TUGboat article: Parsing complex data formats in LuaTEX with
↪  LPEG](https://tug.org/TUGboat/tb40-2/tb125menke-Patterndf)
--
-- @module luakeys

local lpeg = require('lpeg')
local Variable = lpeg.V
local Pattern = lpeg.P
local Set = lpeg.S
local Range = lpeg.R
local CaptureGroup = lpeg.Cg
local CaptureFolding = lpeg.Cf
local CaptureTable = lpeg.Ct
local CaptureConstant = lpeg.Cc
local CaptureSimple = lpeg.C

if not tex then
  tex = {}

  -- Dummy function for the tests.
  tex['sp'] = function (input)
    return 1234567
  end
end

```

```
53  --- A table to store parsed key-value results.
54  local result_store = {}
55
56  --- Generate the PEG parser using Lpeg.
57  --
58  -- @treturn userdata The parser.
59  local function generate_parser(options)
60    -- Optional whitespace
61    local white_space = Set(' \t\n\r')
62
63    --- Match literal string surrounded by whitespace
64    local ws = function(match)
65      return white_space^0 * Pattern(match) * white_space^0
66    end
67
68    local capture_dimension = function (input)
69      if options.convert_dimensions then
70        return tex.sp(input)
71      else
72        return input
73      end
74    end
75
76    --- Add values to a table in two modes:
77    --
78    -- # Key value pair
79    --
80    -- If arg1 and arg2 are not nil, then arg1 is the key and arg2 is the
81    -- value of a new table entry.
82    --
83    -- # Index value
84    --
85    -- If arg2 is nil, then arg1 is the value and is added as an indexed
86    -- (by an integer) value.
87    --
88    -- @tparam table table
89    -- @tparam mixed arg1
90    -- @tparam mixed arg2
91    --
92    -- @treturn table
93    local add_to_table = function(table, arg1, arg2)
94      if arg2 == nil then
95        local index = #table + 1
96        return rawset(table, index, arg1)
97      else
98        return rawset(table, arg1, arg2)
99      end
100   end
101
102   return Pattern({
103     'list',
104
105     -- list_item*
106     list = CaptureFolding(
107       CaptureTable('') * Variable('list_item')^0,
108       add_to_table
109     ),
```

```
110
111       -- '{' list '}'
112       list_container =
113         ws('{') * Variable('list') * ws('}'),
114
115       -- ( list_container / key_value_pair / value ) ','?
116       list_item =
117         CaptureGroup(
118           Variable('list_container') +
119           Variable('key_value_pair') +
120           Variable('value')
121         ) * ws(',')^-1,
122
123       -- key '=' (list_container / value)
124       key_value_pair =
125         (Variable('key') * ws('=')) * (Variable('list_container') +
          ↪  Variable('value')),
126
127       -- number / string_quoted / string_unquoted
128       key =
129         Variable('number') +
130         Variable('string_quoted') +
131         Variable('string_unquoted'),
132
133       -- boolean !value / dimension !value / number !value / string_quoted !value /
          ↪  string_unquoted
134       -- !value -> Not-predicate -> * -Variable('value')
135       value =
136         Variable('boolean') * -Variable('value') +
137         Variable('dimension') * -Variable('value') +
138         Variable('number') * -Variable('value')  +
139         Variable('string_quoted') * -Variable('value') +
140         Variable('string_unquoted'),
141
142       -- boolean_true / boolean_false
143       boolean =
144         (
145           Variable('boolean_true') * CaptureConstant(true) +
146           Variable('boolean_false') * CaptureConstant(false)
147         ),
148
149       boolean_true =
150         Pattern('true') +
151         Pattern('TRUE') +
152         Pattern('True'),
153
154       boolean_false =
155         Pattern('false') +
156         Pattern('FALSE') +
157         Pattern('False'),
158
159       dimension = (
160         Variable('sign')^0 * white_space^0 *
161         Variable('tex_number') * white_space^0 *
162         Variable('unit')
163       ) / capture_dimension,
164
```

```
    number =
      (white_space^0 * (Variable('lua_number') / tonumber) * white_space^0) ,

    tex_number =
      (Variable('integer')^1 * (Pattern('.') * Variable('integer')^1)^0) +
      (Pattern('.') * Variable('integer')^1),

    -- 'bp' / 'BP' / 'cc' / etc.
    -- https://raw.githubusercontent.com/latex3/lualibs/master/lualibs-util-dim.lua
    unit =
      Pattern('bp') + Pattern('BP') +
      Pattern('cc') + Pattern('CC') +
      Pattern('cm') + Pattern('CM') +
      Pattern('dd') + Pattern('DD') +
      Pattern('em') + Pattern('EM') +
      Pattern('ex') + Pattern('EX') +
      Pattern('in') + Pattern('IN') +
      Pattern('mm') + Pattern('MM') +
      Pattern('nc') + Pattern('NC') +
      Pattern('nd') + Pattern('ND') +
      Pattern('pc') + Pattern('PC') +
      Pattern('pt') + Pattern('PT') +
      Pattern('sp') + Pattern('SP'),

    lua_number =
      Variable('int') *
      Variable('frac')^-1,

    int = Variable('sign')^-1 * (
      Range('19') * Variable('integer') + Variable('integer')
    ),

    frac = Pattern('.') * Variable('integer'),
    sign = Set('-+'),
    integer = Range('09')^1,

    -- '"' ('\"' / !'"')* '"'
    string_quoted =
      white_space^0 * Pattern('"') *
      CaptureSimple((Pattern('\\"') + 1 - Pattern('"'))^0) *
      Pattern('"') * white_space^0,

    string_unquoted =
      white_space^0 *
      CaptureSimple(
        Variable('word_unquoted')^1 *
        (Set(' \t')^1 * Variable('word_unquoted')^1)^0) *
      white_space^0,

    word_unquoted = (1 - white_space - Set('{},='))^1
  })
end

--- Get the size of an array like table `{ 'one', 'two', 'three' }` = 3.
--
-- @tparam table value A table or any input.
--
```

```lua
-- @treturn number The size of the array like table. 0 if the input is
-- no table or the table is empty.
local function get_array_size(value)
  local count = 0
  if type(value) == 'table' then
    for _ in ipairs(value) do count = count + 1 end
  end
  return count
end

--- Get the size of a table `{ one = 'one', 'two', 'three' }` = 3.
--
-- @tparam table value A table or any input.
--
-- @treturn number The size of the array like table. 0 if the input is
-- no table or the table is empty.
local function get_table_size(value)
  local count = 0
  if type(value) == 'table' then
    for _ in pairs(value) do count = count + 1 end
  end
  return count
end

--- Unpack a single valued array table like `{ 'one' }` into `one` or
-- `{ 1 }` into `into`.
--
-- @treturn If the value is a array like table with one non table typed
-- value in it, the unpacked value, else the unchanged input.
local function unpack_single_valued_array_table(value)
  if
    type(value) == 'table' and
    get_array_size(value) == 1 and
    get_table_size(value) == 1 and
    type(value[1]) ~= 'table'
  then
    return value[1]
  end
  return value
end

--- This normalization tasks are performed on the raw input table coming
--  directly from the PEG parser:
--
-- 1. Trim all strings: ` text \n` into `text`
-- 2. Unpack all single valued array like tables: `{ 'text' }` into
--    `text`
--
-- @tparam table raw The raw input table coming directly from the PEG
--   parser
--
-- @tparam table options Some options. A table with the key
--   `unpack_single_array_values`
--
-- @treturn table A normalized table ready for the outside world.
local function normalize(raw, options)
  local function normalize_recursive(raw, result, options)
```

```lua
    for key, value in pairs(raw) do
      if options.unpack_single_array_values then
        value = unpack_single_valued_array_table(value)
      end
      if type(value) == 'table' then
        result[key] = normalize_recursive(value, {}, options)
      else
        result[key] = value
      end
    end
    return result
  end
  return normalize_recursive(raw, {}, options)
end

--- The function `stringify(tbl, for_tex)` converts a Lua table into a
--   printable string. Stringify a table means to convert the table into
--   a string. This function is used to realize the `print` function.
--   `stringify(tbl, true)` (`for_tex = true`) generates a string which
--   can be embeded into TeX documents. The macro `\luakeysdebug{}` uses
--   this option. `stringify(tbl, false)` or `stringify(tbl)` generate a
--   string suitable for the terminal.
--
-- @tparam table input A table to stringify.
--
-- @tparam boolean for_tex Stringify the table into a text string that
--   can be embeded inside a TeX document via tex.print(). Curly braces
--   and whites spaces are escaped.
--
-- https://stackoverflow.com/a/54593224/10193818
local function stringify(input, for_tex)
  local line_break, start_bracket, end_bracket, indent

  if for_tex then
    line_break = '\\par'
    start_bracket = '$\\{$'
    end_bracket = '$\\}$'
    indent = '\\ \\ '
  else
    line_break = '\n'
    start_bracket = '{'
    end_bracket = '}'
    indent = '  '
  end

  local function stringify_inner(input, depth)
    local output = {}
    depth = depth or 0

    local function add(depth, text)
      table.insert(output, string.rep(indent, depth) .. text)
    end

    local function format_key(key)
      if (type(key) == 'number') then
        return string.format('[%s]', key)
      else
```

```
336           return string.format('[\'%s\']', key)
337         end
338       end
339
340       if type(input) ~= 'table' then
341         return tostring(input)
342       end
343
344       for key, value in pairs(input) do
345         if (key and type(key) == 'number' or type(key) == 'string') then
346           key = format_key(key)
347
348           if (type(value) == 'table') then
349             if (next(value)) then
350               add(depth, key .. ' = ' .. start_bracket)
351               add(0, stringify_inner(value, depth + 1))
352               add(depth, end_bracket .. ',');
353             else
354               add(depth, key .. ' = ' .. start_bracket .. end_bracket .. ',')
355             end
356           else
357             if (type(value) == 'string') then
358               value = string.format('\'%s\'', value)
359             else
360               value = tostring(value)
361             end
362
363             add(depth, key .. ' = ' .. value .. ',')
364           end
365         end
366       end
367
368       return table.concat(output, line_break)
369     end
370
371     return start_bracket .. line_break .. stringify_inner(input, 1) .. line_break ..
      ↪  end_bracket
372   end
373
374   --- For the LaTeX version of the macro
375   --  `\luakeysdebug[options]{kv-string}`.
376   --
377   -- @tparam table options_raw Options in a raw format. The table may be
378   -- empty or some keys are not set.
379   --
380   -- @treturn table
381   local function normalize_parse_options (options_raw)
382     if options_raw == nil then
383       options_raw = {}
384     end
385     local options = {}
386
387     if options_raw['unpack single array values'] ~= nil then
388       options['unpack_single_array_values'] = options_raw['unpack single array
        ↪  values']
389     end
390
```

```lua
  if options_raw['convert dimensions'] ~= nil then
    options['convert_dimensions'] = options_raw['convert dimensions']
  end

  if options.convert_dimensions == nil then
    options.convert_dimensions = true
  end

  if options.unpack_single_array_values == nil then
    options.unpack_single_array_values = true
  end

  return options
end

return {
  stringify = stringify,

  --- Parse a LaTeX/TeX style key-value string into a Lua table. With
  -- this function you should be able to parse key-value strings like
  -- this example:
  --
  --     show,
  --     hide,
  --     key with spaces = String without quotes,
  --     string="String with double quotes: ,{}=",
  --     dimension = 1cm,
  --     number = -1.2,
  --     list = {one,two,three},
  --     key value list = {one=one,two=two,three=three},
  --     nested key = {
  --       nested key 2= {
  --         key = value,
  --       },
  --     },
  --
  -- The string above results in this Lua table:
  --
  --     {
  --       'show',
  --       'hide',
  --       ['key with spaces'] = 'String without quotes',
  --       string = 'String with double quotes: ,{}=',
  --       dimension = 1864679,
  --       number = -1.2,
  --       list = {'one', 'two', 'three'},
  --       key value list = {
  --         one = 'one',
  --         three = 'three',
  --         two = 'two'
  --       },
  --       ['nested key'] = {
  --         ['nested key 2'] = {
  --           key = 'value'
  --         }
  --       },
  --     }
```

```
448     --
449     -- @tparam string kv_string A string in the TeX/LaTeX style key-value
450     --   format as described above.
451     --
452     -- @tparam table options A table containing
453     -- settings: `convert_dimensions` `unpack_single_array_values`
454     --
455     -- @treturn table A hopefully properly parsed table you can do
456     -- something useful with.
457     parse = function (kv_string, options)
458       if kv_string == nil then
459         return {}
460       end
461       options = normalize_parse_options(options)
462
463       local parser = generate_parser(options)
464       return normalize(parser:match(kv_string), options)
465     end,
466
467     --- The function `render(tbl)` reverses the function
468     --  `parse(kv_string)`. It takes a Lua table and converts this table
469     --  into a key-value string. The resulting string usually has a
470     --  different order as the input table. In Lua only tables with
471     --  1-based consecutive integer keys (a.k.a. array tables) can be
472     --  parsed in order.
473     --
474     -- @tparam table tbl A table to be converted into a key-value string.
475     --
476     -- @treturn string A key-value string that can be passed to a TeX
477     -- macro.
478     render = function (tbl)
479       local function render_inner(tbl)
480         local output = {}
481         local function add(text)
482           table.insert(output, text)
483         end
484         for key, value in pairs(tbl) do
485           if (key and type(key) == 'string') then
486             if (type(value) == 'table') then
487               if (next(value)) then
488                 add(key .. '={')
489                 add(render_inner(value))
490                 add('},')
491               else
492                 add(key .. '={},')
493               end
494             else
495               add(key .. '=' .. tostring(value) .. ',')
496             end
497           else
498             add(tostring(value) .. ',')
499           end
500         end
501         return table.concat(output)
502       end
503       return render_inner(tbl)
504     end,
```

```
505
506     --- The function `print(tbl)` pretty prints a Lua table to standard
507     --   output (stdout). It is a utility function that can be used to
508     --   debug and inspect the resulting Lua table of the function
509     --   `parse`. You have to compile your TeX document in a console to
510     --   see the terminal output.
511     --
512     -- @tparam table tbl A table to be printed to standard output for
513     -- debugging purposes.
514     print = function(tbl)
515       print(stringify(tbl, false))
516     end,
517
518     --- The function `save(identifier, result): void` saves a result (a
519     --  table from a previous run of `parse`) under an identifier.
520     --  Therefore, it is not necessary to pollute the global namespace to
521     --  store results for the later usage.
522     --
523     -- @tparam string identifier The identifier under which the result is
524     --   saved.
525     --
526     -- @tparam table result A result to be stored and that was created by
527     --   the key-value parser.
528     save = function(identifier, result)
529       result_store[identifier] = result
530     end,
531
532     --- The function `get(identifier): table` retrieves a saved result
533     --  from the result store.
534     --
535     -- @tparam string identifier The identifier under which the result was
536     --   saved.
537     get = function(identifier)
538       return result_store[identifier]
539     end,
540
541   }
```

6.2 luakeys-debug.tex

```
%% luakeys-debug.tex
%% Copyright 2021-2022 Josef Friedrich
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3c
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3c or later is part of all distributions of LaTeX
% version 2008/05/04 or later.
%
% This work has the LPPL maintenance status `maintained'.
%
% The Current Maintainer of this work is Josef Friedrich.
%
% This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
% luakeys-debug.sty and luakeys-debug.tex.

\directlua{
  luakeys = require('luakeys')
}

% https://tex.stackexchange.com/a/418401/42311
\catcode`\@=11
\long\def\LuaKeysIfNextChar#1#2#3{%
  \let\@tmpa=#1%
  \def\@tmpb{#2}%
  \def\@tmpc{#3}%
  \futurelet\@future\LuaKeysIfNextChar@i%
}%
\def\LuaKeysIfNextChar@i{%
  \ifx\@tmpa\@future%
    \expandafter\@tmpb
  \else
    \expandafter\@tmpc
  \fi
}%
\def\luakeysdebug@parse@options#1{
  \directlua{
    luakeys.save('debug_options', luakeys.parse('#1'))
  }
}%
\def\luakeysdebug@output#1{
  {
    \tt
    \parindent=0pt
    \directlua{
      local result = luakeys.parse('\luaescapestring{\unexpanded{#1}}',
      ↪  luakeys.get('debug_options'))
      tex.print(luakeys.stringify(result, true))
      luakeys.print(result)
    }
  }
}%
\def\luakeysdebug@oarg[#1]#2{%
  \luakeysdebug@parse@options{#1}%
```

```
    \luakeysdebug@output{#2}%
  }%
  \def\luakeysdebug@marg#1{%
    \luakeysdebug@output{#1}%
  }%
  \def\luakeysdebug{\LuaKeysIfNextChar[{\luakeysdebug@oarg}{\luakeysdebug@marg}}%
  \catcode`\@=12
```

6.3 luakeys-debug.sty

```
%% luakeys-debug.sty
%% Copyright 2021-2022 Josef Friedrich
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3c
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3c or later is part of all distributions of LaTeX
% version 2008/05/04 or later.
%
% This work has the LPPL maintenance status `maintained'.
%
% The Current Maintainer of this work is Josef Friedrich.
%
% This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
% luakeys-debug.sty and luakeys-debug.tex.

\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{luakeys-debug}[2021/12/31 v0.4 Debug package for luakeys.]

\input luakeys-debug.tex
```

Change History

v0.1

General: Inital release 25

v0.2

General: * Allow all recognized data types as keys * Allow TeX macros in the values * New public Lua functions: save(identifier, result), get(identifier) 25

v0.3

General: * Add a LuaLaTeX wrapper “luakeys.sty” * Add a plain LuaTeX wrapper “luakeys.tex” * Rename the previous documentation file “luakeys.tex” to luakeys-doc.tex” 25

v0.4

General: * Parser: Add support for nested tables (for example ’a’, ’b’) * Parser: Allow only strings and numbers as keys * Parser: Remove support from Lua numbers with exponents (for example ’5e+20’) * Switch the Lua testing framework to busted 25