The luakeys package

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

v0.2 from 2021/09/19

```
local luakeys = require('luakeys')
local kv = luakeys.parse('level1={level2={level3={dim=1cm,bool=true,num=-1e-
↪  03,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

```
\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}
```

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-item*⟩ ⟨*list*⟩

⟨*list-item*⟩ ::= (⟨*key-value-pair*⟩ | ⟨*value-without-key*⟩) [‘,’]

⟨*list-container*⟩ ::== ‘{’ ⟨*list*⟩ ‘}’

⟨*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'
```

... 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,
  num4 = 4.57e-3,
  num5 = 0.3e12,
  num6 = 5e+20
}
```

```
{
  ['num1'] = 4,
  ['num2'] = -4,
  ['num3'] = 0.4,
  ['num4'] = 0.00457,
  ['num5'] = 300000000000.0,
  ['num6'] = 5e+20
}
```

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-debug.tex
-- Copyright 2021 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-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-lpeg.pdf)
--
-- @module luakeys

local lpeg = require('lpeg')

if not tex then
  tex = {}

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

--- A table to store parsed key-value results.
local result_store = {}

--- Generate the PEG parser using Lpeg.
--
-- @treturn userdata The parser.
local function generate_parser(options)
  -- Optional whitespace
  local white_space = lpeg.S(' \t\n\r')
```

```lua

--- Match literal string surrounded by whitespace
local ws = function(match)
  return white_space^0 * lpeg.P(match) * white_space^0
end

local boolean_true =
  lpeg.P('true') +
  lpeg.P('TRUE') +
  lpeg.P('True')

local boolean_false =
  lpeg.P('false') +
  lpeg.P('FALSE') +
  lpeg.P('False')

local number = lpeg.P({'number',
  number =
    lpeg.V('int') *
    lpeg.V('frac')^-1 *
    lpeg.V('exp')^-1,

  int = lpeg.V('sign')^-1 * (
    lpeg.R('19') * lpeg.V('digits') + lpeg.V('digit')
  ),

  sign = lpeg.S('+-'),
  digit = lpeg.R('09'),
  digits = lpeg.V('digit') * lpeg.V('digits') + lpeg.V('digit'),
  frac = lpeg.P('.') * lpeg.V('digits'),
  exp = lpeg.S('eE') * lpeg.V('sign')^-1 * lpeg.V('digits'),
})

--- Define data type dimension.
--
-- @return Lpeg patterns
local function build_dimension_pattern()
  local sign = lpeg.S('-+')
  local integer = lpeg.R('09')^1
  local tex_number = (integer^1 * (lpeg.P('.') * integer^1)^0) + (lpeg.P('.') *
  ↪  integer^1)

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

  local dimension = (sign^0 * white_space^0 * tex_number * white_space^0 * unit)

  if options.convert_dimensions then
    return dimension / tex.sp
  else
    return lpeg.C(dimension)
  end
end

--- Add values to a table in two modes:
--
-- # Key value pair
--
-- If arg1 and arg2 are not nil, then arg1 is the key and arg2 is the
-- value of a new table entry.
--
-- # Index value
--
-- If arg2 is nil, then arg1 is the value and is added as an indexed
-- (by an integer) value.
--
-- @tparam table table
-- @tparam mixed arg1
-- @tparam mixed arg2
--
-- @treturn table
local add_to_table = function(table, arg1, arg2)
  if arg2 == nil then
    local index = #table + 1
    return rawset(table, index, arg1)
  else
    return rawset(table, arg1, arg2)
  end
end

return lpeg.P({
  'list',

  list = lpeg.Cf(
    lpeg.Ct('') * lpeg.V('list_item')^0,
    add_to_table
  ),

  list_container =
    ws('{') * lpeg.V('list') * ws('}'),

  list_item =
    lpeg.Cg(
      lpeg.V('key_value_pair') +
      lpeg.V('value')
    ) * ws(',')^-1,

  key_value_pair =
    (lpeg.V('value') * ws('=')) * (lpeg.V('list_container') + lpeg.V('value')),

  value =
      lpeg.V('boolean') +
      lpeg.V('dimension') +
      lpeg.V('number') +
      lpeg.V('string_quoted') +
      lpeg.V('string_unquoted'),

    boolean =
      boolean_true * lpeg.Cc(true) +
      boolean_false * lpeg.Cc(false),

    dimension = build_dimension_pattern(),

    string_quoted =
      white_space^0 * lpeg.P('"') *
      lpeg.C((lpeg.P('\\"') + 1 - lpeg.P('"'))^0) *
      lpeg.P('"') * white_space^0,

    string_unquoted =
      white_space^0 *
      lpeg.C(
        lpeg.V('word_unquoted')^1 *
        (lpeg.S(' \t')^1 * lpeg.V('word_unquoted')^1)^0) *
      white_space^0,

    word_unquoted = (1 - white_space - lpeg.S('{},='))^1;

    number =
      white_space^0 * (number / tonumber) * white_space^0,

  })
end

--- Get the size of an array like table `{ '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_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
```

```lua
  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)
    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

    if type(input) ~= 'table' then
      return tostring(input)
    end

    for key, value in pairs(input) do
      if (key and type(key) == 'number' or type(key) == 'string') then
        key = string.format('[\'%s\']', key);

        if (type(value) == 'table') then
          if (next(value)) then
            add(depth, key .. ' = ' .. start_bracket);
            add(0, stringify_inner(value, depth + 1, for_tex));
            add(depth, end_bracket .. ',');
          else
            add(depth, key .. ' = ' .. start_bracket .. end_bracket .. ',');
          end
        else
          if (type(value) == 'string') then
            value = string.format('\'%s\'', value);
          else
            value = tostring(value);
          end

          add(depth, key .. ' = ' .. value .. ',');
        end
```

```lua
      end
    end

    return table.concat(output, line_break)
  end

  return start_bracket .. line_break .. stringify_inner(input, 1) .. line_break ..
  ↪  end_bracket
end

--- For the LaTeX version of the macro
--  `\luakeysdebug[options]{kv-string}`.
--
-- @tparam table options_raw Options in a raw format. The table may be
-- empty or some keys are not set.
--
-- @treturn table
local function normalize_parse_options (options_raw)
  if options_raw == nil then
    options_raw = {}
  end
  local options = {}

  if options_raw['unpack single array values'] ~= nil then
    options['unpack_single_array_values'] = options_raw['unpack single array
    ↪  values']
  end

  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'
  --         }
  --       },
  --     }
  --
  -- @tparam string kv_string A string in the TeX/LaTeX style key-value
  --   format as described above.
  --
  -- @tparam table options A table containing
  -- settings: `convert_dimensions` `unpack_single_array_values`
  --
  -- @treturn table A hopefully properly parsed table you can do
  -- something useful with.
  parse = function (kv_string, options)
    if kv_string == nil then
      return {}
    end
    options = normalize_parse_options(options)

    local parser = generate_parser(options)
    return normalize(parser:match(kv_string), options)
  end,

  --- 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. In Lua only tables with
  --  1-based consecutive integer keys (a.k.a. array tables) can be
  --  parsed in order.
  --
  -- @tparam table tbl A table to be converted into a key-value string.
  --
  -- @treturn string A key-value string that can be passed to a TeX
  -- macro.
  render = function (tbl)
    local function render_inner(tbl)
      local output = {}
      local function add(text)
        table.insert(output, text)
      end
      for key, value in pairs(tbl) do
        if (key and type(key) == 'string') then
          if (type(value) == 'table') then
            if (next(value)) then
              add(key .. '={');
              add(render_inner(value));
              add('},');
            else
              add(key .. '={},');
            end
          else
            add(key .. '=' .. tostring(value) .. ',');
          end
        else
          add(tostring(value) .. ',')
        end
      end
      return table.concat(output)
    end
    return render_inner(tbl)
  end,

  --- 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.
  --
  -- @tparam table tbl A table to be printed to standard output for
  -- debugging purposes.
  print = function(tbl)
    print(stringify(tbl, false))
  end,

  --- The function `save(identifier, result): void` 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.
  --
  -- @tparam string identifier The identifier under which the result is
  --   saved.
  --
  -- @tparam table result A result to be stored and that was created by
  --   the key-value parser.
  save = function(identifier, result)
    result_store[identifier] = result
  end,

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

```lua
  --
  -- @tparam string identifier The identifier under which the result was
  --   saved.
  get = function(identifier)
    return result_store[identifier]
  end,

}
```

6.2 luakeys-debug.tex

```
%% luakeys-debug.tex
%% Copyright 2021 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-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 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-debug.sty
% and luakeys-debug.tex.

\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{luakeys-debug}[2021/09/19 Debug package for luakeys.]

\input luakeys-debug.tex
```

Change History

v0.1
  General: Inital release 24
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) 24