The luakeys package

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

0.11.0 from 2022/12/23

```lua
local result = luakeys.parse(
  'level1={level2={naked,dim=1cm,bool=false,num=-0.001,str="lua,{}"}}',
  { convert_dimensions = true })
luakeys.debug(result)
```

Result:

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

Contents

1 Introduction **4**
1.1 Pros of luakeys . . . 4
1.2 Cons of luakeys . . . 4

2 How the package is loaded **4**
2.1 Using the Lua module luakeys.lua . . . 4
2.2 Using the LuaLaTeX wrapper luakeys.sty . . . 5
2.3 Using the plain LuaTeX wrapper luakeys.tex . . . 5

3 Lua interface / API **5**
3.1 Lua indentifier names . . . 6
3.2 Function "get_private_instance(): table" . . . 6
3.3 Function "parse(kv_string, opts): result, unknown, raw" . 6
3.4 Options to configure the parse function . . . 7
3.5 Table "opts" . . . 8
3.5.1 Option "assignment_operator" . . . 9
3.5.2 Option "convert_dimensions" . . . 9
3.5.3 Option "debug" . . . 10
3.5.4 Option "default" . . . 10
3.5.5 Option "defaults" . . . 10
3.5.6 Option "defs" . . . 11
3.5.7 Option "false_aliases" . . . 11
3.5.8 Option "format_keys" . . . 11
3.5.9 Option "group_begin" . . . 12
3.5.10 Option "group_end" . . . 12
3.5.11 Option "invert_flag" . . . 12
3.5.12 Option "hooks" . . . 13
3.5.13 Option "list_separator" . . . 14
3.5.14 Option "naked_as_value" . . . 14
3.5.15 Option "no_error" . . . 14
3.5.16 Option "quotation_begin" . . . 15
3.5.17 Option "quotation_end" . . . 15
3.5.18 Option "true_aliases" . . . 15
3.5.19 Option "unpack" . . . 15
3.6 Function "define(defs, opts): parse" . . . 15
3.7 Attributes to define a key-value pair . . . 16
3.7.1 Attribute "alias" . . . 17
3.7.2 Attribute "always_present" . . . 17
3.7.3 Attribute "choices" . . . 17
3.7.4 Attribute "data_type" . . . 18
3.7.5 Attribute "default" . . . 18
3.7.6 Attribute "exclusive_group" . . . 18
3.7.7 Attribute "opposite_keys" . . . 19
3.7.8 Attribute "macro" . . . 19
3.7.9 Attribute "match" . . . 19
3.7.10 Attribute "name" . . . 20
3.7.11 Attribute "pick" . . . 20
3.7.12 Attribute "process" . . . 21

3.7.13 Attribute “required” 22
3.7.14 Attribute “sub_keys” 23
3.8 Function “render(result): string” 23
3.9 Function “debug(result): void” 24
3.10 Function “save(identifier, result): void” 24
3.11 Function “get(identifier): result” 24
3.12 Table “is” 24
3.12.1 Function “is.boolean(value): boolean” 24
3.12.2 Function “is.dimension(value): boolean” 25
3.12.3 Function “is.integer(value): boolean” 25
3.12.4 Function “is.number(value): boolean” 25
3.12.5 Function “is.string(value): boolean” 25
3.12.6 Function “is.any(value): boolean” 25
3.13 Table “utils” 26
3.13.1 Function “utils.scan_oarg(initial_delimiter?, end_-delimiter?): string” 26
3.14 Table “version” 26

4 Syntax of the recognized key-value format 27
4.1 An attempt to put the syntax into words 27
4.2 An (incomplete) attempt to put the syntax into the Extended Backus-Naur Form 27
4.3 Recognized data types 28
4.3.1 boolean 28
4.3.2 number 28
4.3.3 dimension 28
4.3.4 string 30
4.3.5 Naked keys 30

5 Examples 32
5.1 Extend and modify keys of existing macros 32
5.2 Process document class options 33

6 Debug packages 34
6.1 For plain TeX: luakeys-debug.tex 34
6.2 For LaTeX: luakeys-debug.sty 34

7 Activity diagramm of the parsing process 35

8 Implementation 36
8.1 luakeys.lua 36
8.2 luakeys.tex 59
8.3 luakeys.sty 60
8.4 luakeys-debug.tex 61
8.5 luakeys-debug.sty 62

1 Introduction

luakeys is a Lua module / LuaTEXpackage that can parse key-value options like the TEX packages keyval, kvsetkeys, kvoptions, xkeyval, pgfkeys etc. luakeys, however, accomplishes this task 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).

1.1 Pros of luakeys

- Key-value pairs can be parsed independently of the macro collection (LaTEX or ConTEXt).
- Even in plain LuaTEX keys can be parsed.
- luakeys can handle nested lists of key-value pairs, i.e. it can handle a recursive data structure of keys.
- Keys do not have to be defined, but can they can be defined.

1.2 Cons of luakeys

- The package works only in combination with LuaTEX.
- You need to know two languages: TEX and Lua.

2 How the package is loaded

2.1 Using the Lua module luakeys.lua

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

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

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

2.2 Using the LuaLaTeX wrapper luakeys.sty

For example, the MiKTeX package manager downloads packages only when needed. It has been reported that this automatic download only works with this wrapper files. Probably MiKTeX is searching for an occurrence of the LaTeX macro "\usepackage {luakeys}". 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', { naked_as_value = true })
    tex.print(keys[1])
    tex.print(keys[2])
    tex.print(keys[3])
  } % one two three
\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', { naked_as_value = true })
  tex.print(keys[1])
  tex.print(keys[2])
  tex.print(keys[3])
} % one two three
\bye
```

3 Lua interface / API

The Lua module exports this functions and tables:

```
local luakeys = require('luakeys')
local version = luakeys.version
local get_private_instance = luakeys.get_private_instance
local opts = luakeys.opts
local stringify = luakeys.stringify
local define = luakeys.define
local parse = luakeys.parse
local render = luakeys.render
local debug = luakeys.debug
```

```
local save = luakeys.save
local get = luakeys.get
local is = luakeys.is
local utils = luakeys.utils
```

This documentation presents only the public functions and tables. To learn more about the private, not exported functions, please read the source code documentation, which was created with LDoc.

3.1 Lua indentifier names

The project uses a few abbreviations for variable names that are hopefully unambiguous and familiar to external readers.

Abbreviation	spelled out	Example
`kv_string`	Key-value string	`'key=value'`
`opts`	Options (for the parse function)	`{ no_error = false }`
`defs`	Definitions	
`def`	Definition	
`attr`	Attributes (of a definition)	

These unabbreviated variable names are commonly used.

`result`	The final result of all individual parsing and normalization steps.
`unknown`	A table with unknown, undefinied key-value pairs.
`raw`	The raw result of the Lpeg grammar parser.

3.2 Function "`get_private_instance(): table`"

The default options can be changed globally via the `opts` table (3.5). To prevent another modules from changing the default settings, a new instance of the `luakeys` module can be created using the `get_private_instance()` function.

```
local my_luakeys = require('luakeys').get_private_instance()
local result = my_luakeys.parse('key=value')
```

The `require()` function always returns the same table when called multiple times.

```
local l1 = require('luakeys') -- table: 0x564ea6ca4160
local l2 = require('luakeys') -- table: 0x564ea6ca4160
local l3 = require('luakeys').get_private_instance() -- table: 0x563574d51470
local l4 = require('luakeys').get_private_instance() -- table: 0x563574d86ac0
```

3.3 Function "`parse(kv_string, opts): result, unknown, raw`"

The function `parse(kv_string, opts)` is the most important function of the package. It converts a key-value string into a Lua table.

```
\documentclass{article}
\usepackage{luakeys}
\begin{document}
\newcommand{\mykeyvalcmd}[2][]{
  \directlua{
    local result = luakeys.parse('#1')
    tex.print('The key "one" has the value ' .. tostring(result.one) .. '.')
```

```
  }
  marg: #2
}
\mykeyvalcmd[one=1]{test}
\end{document}
```

In plain TEX:

```
\input luakeys.tex
\def\mykeyvalcmd#1{
  \directlua{
    local result = luakeys.parse('#1')
    tex.print('The key "one" has the value ' .. tostring(result.one) .. '.')
  }
}
\mykeyvalcmd{one=1}
\bye
```

3.4 Options to configure the parse function

The `parse` function can be called with an options table. This options are supported: assignment_operator, convert_dimensions, debug, default, defaults, defs, false_aliases, format_keys, group_begin, group_end, hooks, invert_flag, list_separator, naked_as_value, no_error, quotation_begin, quotation_end, true_aliases, unpack

```
local opts = {
  -- Configure the delimiter that assigns a value to a key.
  assignment_operator = '=',

  -- Automatically convert dimensions into scaled points (1cm -> 1864679).
  convert_dimensions = false,

  -- Print the result table to the console.
  debug = false,

  -- The default value for naked keys (keys without a value).
  default = true,

  -- A table with some default values. The result table is merged with
  -- this table.
  defaults = { key = 'value' },

  -- Key-value pair definitions.
  defs = { key = { default = 'value' } },

  -- Specify the strings that are recognized as boolean false values.
  false_aliases = { 'false', 'FALSE', 'False' },

  -- lower, snake, upper
  format_keys = { 'snake' },

  -- Configure the delimiter that marks the beginning of a group.
  group_begin = '{',

  -- Configure the delimiter that marks the end of a group.
  group_end = '}',

  -- Listed in the order of execution
```

```
  hooks = {
    kv_string = function(kv_string)
      return kv_string
    end,

    -- Visit all key-value pairs recursively.
    keys_before_opts = function(key, value, depth, current, result)
      return key, value
    end,

    -- Visit the result table.
    result_before_opts = function(result)
    end,

    -- Visit all key-value pairs recursively.
    keys_before_def = function(key, value, depth, current, result)
      return key, value
    end,

    -- Visit the result table.
    result_before_def = function(result)
    end,

    -- Visit all key-value pairs recursively.
    keys = function(key, value, depth, current, result)
      return key, value
    end,

    -- Visit the result table.
    result = function(result)
    end,
  },

  invert_flag = '!',

  -- Configure the delimiter that separates list items from each other.
  list_separator = ',',

  -- If true, naked keys are converted to values:
  -- { one = true, two = true, three = true } -> { 'one', 'two', 'three' }
  naked_as_value = false,

  -- Throw no error if there are unknown keys.
  no_error = false,

  -- Configure the delimiter that marks the beginning of a string.
  quotation_begin = '"',

  -- Configure the delimiter that marks the end of a string.
  quotation_end = '"',

  -- Specify the strings that are recognized as boolean true values.
  true_aliases = { 'true', 'TRUE', 'True' },

  -- { key = { 'value' } } -> { key = 'value' }
  unpack = false,
}
```

3.5 Table "opts"

The options can also be set globally using the exported table `opts`:

```
local result = luakeys.parse('dim=1cm') -- { dim = '1cm' }
```

```
luakeys.opts.convert_dimensions = true
local result2 = luakeys.parse('dim=1cm') -- { dim = 1234567 }
```

To avoid interactions with other packages that also use luakeys and set the options globally, it is recommended to use the `get_private_instance()` function (3.2) to load the package.

3.5.1 Option "assignment_operator"

The option `assignment_operator` configures the delimiter that assigns a value to a key. The default value of this option is `"="`.

The code example below demonstrates all six delimiter related options.

```
local result = luakeys.parse(
  'level1: ( key1: value1; key2: "A string;" )', {
    assignment_operator = ':',
    group_begin = '(',
    group_end = ')',
    list_separator = ';',
    quotation_begin = '"',
    quotation_end = '"',
  })
luakeys.debug(result) -- { level1 = { key1 = 'value1', key2 = 'A string;' } }
```

Delimiter options	Section
assignment_operator	3.5.1
group_begin	3.5.9
group_end	3.5.10
list_separator	3.5.13
quotation_begin	3.5.16
quotation_end	3.5.17

3.5.2 Option "convert_dimensions"

If you set the option `convert_dimensions` to `true`, luakeys detects the TEX dimensions and converts them into scaled points using the function `tex.sp(dim)`.

```
local result = luakeys.parse('dim=1cm', {
  convert_dimensions = true,
})
-- result = { dim = 1864679 }
```

By default the dimensions are not converted into scaled points.

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

If you want to convert a scaled points number into a dimension string you can use the module lualibs-util-dim.lua.

```
require('lualibs')
tex.print(number.todimen(tex.sp('1cm'), 'cm', '%0.0F%s'))
```

The default value of the option "`convert_dimensions`" is: `false`.

3.5.3 Option "debug"

If the option `debug` is set to ture, the result table is printed to the console.

```
\documentclass{article}
\usepackage{luakeys}
\begin{document}
\directlua{
  luakeys.parse('one,two,three', { debug = true })
}
Lorem ipsum
\end{document}
```

```
This is LuaHBTeX, Version 1.15.0 (TeX Live 2022)
...
(./debug.aux) (/usr/local/texlive/texmf-dist/tex/latex/base/ts1cmr.fd)
{
  ['three'] = true,
  ['two'] = true,
  ['one'] = true,
}
 [1{/usr/
local/texlive/2022/texmf-var/fonts/map/pdftex/updmap/pdftex.map}] (./debug.aux)
)
...
Transcript written on debug.log.
```

The default value of the option "`debug`" is: `false`.

3.5.4 Option "default"

The option `default` can be used to specify which value naked keys (keys without a value) get. This option has no influence on keys with values.

```
local result = luakeys.parse('naked', { default = 1 })
luakeys.debug(result) -- { naked = 1 }
```

By default, naked keys get the value `true`.

```
local result2 = luakeys.parse('naked')
luakeys.debug(result2) -- { naked = true }
```

The default value of the option "`default`" is: `true`.

3.5.5 Option "defaults"

The option "defaults" can be used to specify not only one default value, but a whole table of default values. The result table is merged into the defaults table. Values in the defaults table are overwritten by values in the result table.

```lua
local result = luakeys.parse('key1=new', {
  defaults = { key1 = 'default', key2 = 'default' },
})
luakeys.debug(result) -- { key1 = 'new', key2 = 'default' }
```

The default value of the option "`defaults`" is: `false`.

3.5.6 Option "defs"

For more informations on how keys are defined, see section 3.6. If you use the `defs` option, you don't need to call the `define` function. Instead of ...

```lua
local parse = luakeys.define({ one = { default = 1 }, two = { default = 2 } })
local result = parse('one,two') -- { one = 1, two = 2 }
```

we can write ...

```lua
local result2 = luakeys.parse('one,two', {
  defs = { one = { default = 1 }, two = { default = 2 } },
}) -- { one = 1, two = 2 }
```

The default value of the option "`defs`" is: `false`.

3.5.7 Option "false_aliases"

The `true_aliases` and `false_aliases` options can be used to specify the strings that will be recognized as boolean values by the parser. The following strings are configured by default.

```lua
local result = luakeys.parse('key=yes', {
  true_aliases = { 'true', 'TRUE', 'True' },
  false_aliases = { 'false', 'FALSE', 'False' },
})
luakeys.debug(result) -- { key = 'yes' }
```

```lua
local result2 = luakeys.parse('key=yes', {
  true_aliases = { 'on', 'yes' },
  false_aliases = { 'off', 'no' },
})
luakeys.debug(result2) -- { key = true }
```

```lua
local result3 = luakeys.parse('key=true', {
  true_aliases = { 'on', 'yes' },
  false_aliases = { 'off', 'no' },
})
luakeys.debug(result3) -- { key = 'true' }
```

See section 3.5.18 for the corresponding option.

3.5.8 Option "format_keys"

With the help of the option `format_keys` the keys can be formatted. The values of this option must be specified in a table.

lower To convert all keys to *lowercase*, specify `lower` in the options table.

```
local result = luakeys.parse('KEY=value', { format_keys = { 'lower' } })
luakeys.debug(result) -- { key = 'value' }
```

snake To make all keys *snake case* (The words are separated by underscores), specify `snake` in the options table.

```
local result2 = luakeys.parse('snake case=value', { format_keys = {
↪   'snake' } })
luakeys.debug(result2) -- { snake_case = 'value' }
```

upper To convert all keys to *uppercase*, specify `upper` in the options table.

```
local result3 = luakeys.parse('key=value', { format_keys = { 'upper' }
↪  })
luakeys.debug(result3) -- { KEY = 'value' }
```

You can also combine several types of formatting.

```
local result4 = luakeys.parse('Snake Case=value', { format_keys = { 'lower',
↪   'snake' } })
luakeys.debug(result4) -- { snake_case = 'value' }
```

The default value of the option “`format_keys`” is: `false`.

3.5.9 Option “group_begin”

The option `group_begin` configures the delimiter that marks the beginning of a group. The default value of this option is `"{"`. A code example can be found in section 3.5.1.

3.5.10 Option “group_end”

The option `group_end` configures the delimiter that marks the end of a group. The default value of this option is `"}"`. A code example can be found in section 3.5.1.

3.5.11 Option “invert_flag”

If a naked key is prefixed with an exclamation mark, its default value is inverted. Instead of `true` the key now takes the value `false`.

```
local result = luakeys.parse('naked1,!naked2')
luakeys.debug(result) -- { naked1 = true, naked2 = false }
```

The `invert_flag` option can be used to change this inversion character.

```
local result2 = luakeys.parse('naked1,~naked2', { invert_flag = '~' })
luakeys.debug(result2) -- { naked1 = true, naked2 = false }
```

For example, if the default value for naked keys is set to `false`, the naked keys prefixed with the invert flat take the value `true`.

```lua
local result3 = luakeys.parse('naked1,!naked2', { default = false })
luakeys.debug(result3) -- { naked1 = false, naked2 = true }
```

Set the `invert_flag` option to `false` to disable this automatic boolean value inversion.

```lua
local result4 = luakeys.parse('naked1,!naked2', { invert_flag = false })
luakeys.debug(result4) -- { naked1 = true, ['!naked2'] = true }
```

3.5.12 Option "hooks"

The following hooks or callback functions allow to intervene in the processing of the `parse` function. The functions are listed in processing order. `*_before_opts` means that the hooks are executed after the LPeg syntax analysis and before the options are applied. The `*_before_defs` hooks are executed before applying the key value definitions.

1. `kv_string` `= function(kv_string): kv_string`
2. `keys_before_opts` `= function(key, value, depth, current, result): key, value`
3. `result_before_opts` `= function(result): void`
4. `keys_before_def` `= function(key, value, depth, current, result): key, value`
5. `result_before_def` `= function(result): void`
6. (`process`) (has to be definied using defs, see 3.7.12)
7. `keys` `= function(key, value, depth, current, result): key, value`
8. `result` `= function(result): void`

`kv_string` The `kv_string` hook is called as the first of the hook functions before the LPeg syntax parser is executed.

```lua
local result = luakeys.parse('key=unknown', {
  hooks = {
    kv_string = function(kv_string)
      return kv_string:gsub('unknown', 'value')
    end,
  },
})
luakeys.debug(result) -- { key = 'value' }
```

`keys_*` The hooks `keys_*` are called recursively on each key in the current result table. The hook function must return two values: `key, value`. The following example returns `key` and `value` unchanged, so the result table is not changed.

```lua
local result = luakeys.parse('l1={l2=1}', {
  hooks = {
    keys = function(key, value)
      return key, value
    end,
```

```
  },
})
luakeys.debug(result) -- { l1 = { l2 = 1 } }
```

The next example demonstrates the third parameter `depth` of the hook function.

```
local result = luakeys.parse('x,d1={x,d2={x}}', {
  naked_as_value = true,
  unpack = false,
  hooks = {
    keys = function(key, value, depth)
      if value == 'x' then
        return key, depth
      end
      return key, value
    end,
  },
})
luakeys.debug(result) -- { 1, d1 = { 2, d2 = { 3 } } }
```

result_* The hooks `result_*` are called once with the current result table as a parameter.

3.5.13 Option "list_separator"

The option `list_separator` configures the delimiter that separates list items from each other. The default value of this option is ",". A code example can be found in section 3.5.1.

3.5.14 Option "naked_as_value"

With the help of the option `naked_as_value`, naked keys are not given a default value, but are stored as values in a Lua table.

```
local result = luakeys.parse('one,two,three')
luakeys.debug(result) -- { one = true, two = true, three = true }
```

If we set the option `naked_as_value` to `true`:

```
local result2 = luakeys.parse('one,two,three', { naked_as_value = true })
luakeys.debug(result2)
-- { [1] = 'one', [2] = 'two', [3] = 'three' }
-- { 'one', 'two', 'three' }
```

The default value of the option "`naked_as_value`" is: `false`.

3.5.15 Option "no_error"

By default the parse function throws an error if there are unknown keys. This can be prevented with the help of the `no_error` option.

```
  luakeys.parse('unknown', { defs = { 'key' } })
  -- Error message: Unknown keys: unknown,
```

If we set the option `no_error` to `true`:

```lua
luakeys.parse('unknown', { defs = { 'key' }, no_error = true })
-- No error message
```

The default value of the option "no_error" is: false.

3.5.16 Option "quotation_begin"

The option quotation_begin configures the delimiter that marks the beginning of a string. The default value of this option is '"' (double quotes). A code example can be found in section 3.5.1.

3.5.17 Option "quotation_end"

The option quotation_end configures the delimiter that marks the end of a string. The default value of this option is '"' (double quotes). A code example can be found in section 3.5.1.

3.5.18 Option "true_aliases"

See section 3.5.7.

3.5.19 Option "unpack"

With the help of the option unpack, all tables that consist of only a single naked key or a single standalone value are unpacked.

```lua
local result = luakeys.parse('key={string}', { unpack = true })
luakeys.debug(result) -- { key = 'string' }
```

```lua
local result2 = luakeys.parse('key={string}', { unpack = false })
luakeys.debug(result2) -- { key = { string = true } }
```

The default value of the option "unpack" is: true.

3.6 Function "define(defs, opts): parse"

The define function returns a parse function (see 3.3). The name of a key can be specified in three ways:

1. as a string.
2. as a key in a Lua table. The definition of the corresponding key-value pair is then stored under this key.
3. by the "name" attribute.

```lua
-- standalone string values
local defs = { 'key' }

-- keys in a Lua table
local defs = { key = {} }

-- by the "name" attribute
local defs = { { name = 'key' } }
```

```
local parse = luakeys.define(defs)
local result, unknown = parse('key=value,unknown=unknown', { no_error = true
↪  })
luakeys.debug(result) -- { key = 'value' }
luakeys.debug(unknown) -- { unknown = 'unknown' }
```

For nested definitions, only the last two ways of specifying the key names can be used.

```
local parse2 = luakeys.define({
  level1 = {
    sub_keys = { level2 = { sub_keys = { key = { } } } },
  },
}, { no_error = true })
local result2, unknown2 =
↪  parse2('level1={level2={key=value,unknown=unknown}}')
luakeys.debug(result2) -- { level1 = { level2 = { key = 'value' } } }
luakeys.debug(unknown2) -- { level1 = { level2 = { unknown = 'unknown' } } }
```

3.7 Attributes to define a key-value pair

The definition of a key-value pair can be made with the help of various attributes. The name *"attribute"* for an option, a key, a property ... (to list just a few naming possibilities) to define keys, was deliberately chosen to distinguish them from the options of the `parse` function. These attributes are allowed: alias, always_present, choices, data_type, default, exclusive_group, l3_tl_set, macro, match, name, opposite_keys, pick, process, required, sub_keys. The code example below lists all the attributes that can be used to define key-value pairs.

```
local defs = {
  key = {
    -- Allow different key names.
    -- or a single string: alias = 'k'
    alias = { 'k', 'ke' },

    -- The key is always included in the result. If no default value is
    -- definied, true is taken as the value.
    always_present = false,

    -- Only values listed in the array table are allowed.
    choices = { 'one', 'two', 'three' },

    -- Possible data types: boolean, dimension, integer, number, string
    data_type = 'string',

    default = true,

    -- The key belongs to a mutually exclusive group of keys.
    exclusive_group = 'name',

    -- > \MacroName
    macro = 'MacroName', -- > \MacroName

    -- See http://www.lua.org/manual/5.3/manual.html#6.4.1
    match = '^%d%d%d%d%-%d%d%-%d%d$',

    -- The name of the key, can be omitted
```

```
    name = 'key',
    opposite_keys = { [true] = 'show', [false] = 'hide' },

    -- Pick a value
    -- boolean
    pick = false,

    process = function(value, input, result, unknown)
      return value
    end,
    required = true,
    sub_keys = { key_level_2 = { } },
  }
}
```

3.7.1 Attribute "alias"

With the help of the `alias` attribute, other key names can be used. The value is always stored under the original key name. A single alias name can be specified by a string ...

```
-- a single alias
local parse = luakeys.define({ key = { alias = 'k' } })
local result = parse('k=value')
luakeys.debug(result) -- { key = 'value' }
```

multiple aliases by a list of strings.

```
-- multiple aliases
local parse = luakeys.define({ key = { alias = { 'k', 'ke' } } })
local result = parse('ke=value')
luakeys.debug(result) -- { key = 'value' }
```

3.7.2 Attribute "always_present"

The `default` attribute is used only for naked keys.

```
local parse = luakeys.define({ key = { default = 1 } })
local result = parse('') -- { }
```

If the attribute `always_present` is set to true, the key is always included in the result. If no default value is definied, true is taken as the value.

```
local parse = luakeys.define({ key = { default = 1, always_present = true } })
local result = parse('') -- { key =  1 }
```

3.7.3 Attribute "choices"

Some key values should be selected from a restricted set of choices. These can be handled by passing an array table containing choices.

```
local parse = luakeys.define({ key = { choices = { 'one', 'two', 'three' } }
↪  })
local result = parse('key=one') -- { key = 'one' }
```

When the key-value pair is parsed, values will be checked, and an error message will be displayed if the value was not one of the acceptable choices:

```lua
  parse('key=unknown')
  -- error message:
  --- 'The value "unknown" does not exist in the choices: one, two, three!'
```

3.7.4 Attribute "data_type"

The `data_type` attribute allows type-checking and type conversions to be performed. The following data types are supported: `'boolean'`, `'dimension'`, `'integer'`, `'number'`, `'string'`. A type conversion can fail with the three data types `'dimension'`, `'integer'`, `'number'`. Then an error message is displayed.

```lua
local function assert_type(data_type, input_value, expected_value)
  assert.are.same({ key = expected_value },
    luakeys.parse('key=' .. tostring(input_value),
      { defs = { key = { data_type = data_type } } }))
end
```

```lua
  assert_type('boolean', 'true', true)
  assert_type('dimension', '1cm', '1cm')
  assert_type('integer', '1.23', 1)
  assert_type('number', '1.23', 1.23)
  assert_type('string', 1.23, '1.23')
```

3.7.5 Attribute "default"

Use the `default` attribute to provide a default value for each naked key individually. With the global `default` attribute (3.5.4) a default value can be specified for all naked keys.

```lua
local parse = luakeys.define({
  one = {},
  two = { default = 2 },
  three = { default = 3 },
}, { default = 1, defaults = { four = 4 } })
local result = parse('one,two,three') -- { one = 1, two = 2, three = 3, four =
↪  4 }
```

3.7.6 Attribute "exclusive_group"

All keys belonging to the same exclusive group must not be specified together. Only one key from this group is allowed. Any value can be used as a name for this exclusive group.

```lua
local parse = luakeys.define({
  key1 = { exclusive_group = 'group' },
  key2 = { exclusive_group = 'group' },
})
local result1 = parse('key1') -- { key1 = true }
local result2 = parse('key2') -- { key2 = true }
```

If more than one key of the group is specified, an error message is thrown.

```lua
  parse('key1,key2') -- throws error message:
  -- 'The key "key2" belongs to a mutually exclusive group "group"
  -- and the key "key1" is already present!'
```

3.7.7 Attribute "opposite_keys"

The opposite_keys attribute allows to convert opposite (naked) keys into a boolean value and store this boolean under a target key. Lua allows boolean values to be used as keys in tables. However, the boolean values must be written in square brackets, e. g. `opposite_keys = { [true] = 'show', [false] = 'hide' }`. Examples of opposing keys are: show and hide, dark and light, question and solution. The example below uses the show and hide keys as the opposite key pair. If the key show is parsed by the parse function, then the target key visibility receives the value true.

```lua
local parse = luakeys.define({
  visibility = { opposite_keys = { [true] = 'show', [false] = 'hide' } },
})
local result = parse('show') -- { visibility = true }
```

If the key hide is parsed, then false.

```lua
local result = parse('hide') -- { visibility = false }
```

3.7.8 Attribute "macro"

The attribute macro stores the value in a TEX macro.

```lua
local parse = luakeys.define({
  key = {
    macro = 'MyMacro'
  }
})
parse('key=value')
```

```latex
\MyMacro % expands to "value"
```

3.7.9 Attribute "match"

The value of the key is first passed to the Lua function string.match(value, match) (http://www.lua.org/manual/5.3/manual.html#pdf-string.match) before being assigned to the key. You can therefore configure the match attribute with a pattern matching string used in Lua. Take a look at the Lua manual on how to write patterns (http://www.lua.org/manual/5.3/manual.html#6.4.1).

```lua
local parse = luakeys.define({
  birthday = { match = '^%d%d%d%d%-%d%d%-%d%d$' },
})
local result = parse('birthday=1978-12-03') -- { birthday = '1978-12-03' }
```

If the pattern cannot be found in the value, an error message is issued.

```
parse('birthday=1978-12-XX')
-- throws error message:
-- 'The value "1978-12-XX" of the key "birthday"
--  does not match "^%d%d%d%d%-%d%d%-%d%d$"!'
```

The key receives the result of the function `string.match(value, match)`, which means that the original value may not be stored completely in the key. In the next example, the entire input value is accepted:

```
local parse = luakeys.define({ year = { match = '%d%d%d%d' } })
local result = parse('year=1978') -- { year = '1978' }
```

The prefix "waste " and the suffix " rubbisch" of the string are discarded.

```
local result2 = parse('year=waste 1978 rubbisch') -- { year = '1978' }
```

Since function `string.match(value, match)` always returns a string, the value of the key is also always a string.

3.7.10 Attribute "name"

The `name` attribute allows an alternative notation of key names. Instead of ...

```
local parse1 = luakeys.define({
  one = { default = 1 },
  two = { default = 2 },
})
local result1 = parse1('one,two') -- { one = 1, two = 2 }
```

... we can write:

```
local parse = luakeys.define({
  { name = 'one', default = 1 },
  { name = 'two', default = 2 },
})
local result = parse('one,two') -- { one = 1, two = 2 }
```

3.7.11 Attribute "pick"

The attribute `pick` searches for a value not assigned to a key. The first value found, i.e. the one further to the left, is assigned to a key.

```
local parse = luakeys.define({ font_size = { pick = 'dimension' } })
local result = parse('12pt,13pt', { no_error = true })
luakeys.debug(result) -- { font_size = '12pt' }
```

Only the current result table is searched, not other levels in the recursive data structure.

```
local parse = luakeys.define({
  level1 = {
    sub_keys = { level2 = { default = 2 }, key = { pick = 'boolean' } },
  },
}, { no_error = true })
local result, unknown = parse('true,level1={level2,true}')
```

```
luakeys.debug(result) -- { level1 = { key = true, level2 = 2 } }
luakeys.debug(unknown) -- { true }
```

The search for values is activated when the attribute `pick` is set to a data type. These data types can be used to search for values: number, string, integer, any, boolean, dimension. Use the data type “any” to accept any value. If a value is already assigned to a key when it is entered, then no further search for values is performed.

```
local parse = luakeys.define({ font_size = { pick = 'dimension' } })
local result, unknown =
  parse('font_size=11pt,12pt', { no_error = true })
luakeys.debug(result) -- { font_size = '11pt' }
luakeys.debug(unknown) -- { '12pt' }
```

The `pick` attribute also accepts multiple data types specified in a table.

```
local parse = luakeys.define({
  key = { pick = { 'number', 'dimension' } },
})
local result = parse('string,12pt,42', { no_error = true })
luakeys.debug(result) -- { key = 42 }
local result2 = parse('string,12pt', { no_error = true })
luakeys.debug(result2) -- { key = '12pt' }
```

3.7.12 Attribute “process”

The `process` attribute can be used to define a function whose return value is passed to the key. Four parameters are passed when the function is called:

1. `value`: The current value asssociated with the key.
2. `input`: The result table cloned before the time the definitions started to be applied.
3. `result`: The table in which the final result will be saved.
4. `unknown`: The table in which the unknown key-value pairs are stored.

The following example demonstrates the `value` parameter:

```
local parse = luakeys.define({
  key = {
    process = function(value, input, result, unknown)
      if type(value) == 'number' then
        return value + 1
      end
      return value
    end,
  },
})
local result = parse('key=1') -- { key = 2 }
```

The following example demonstrates the `input` parameter:

```lua
local parse = luakeys.define({
  'one',
  'two',
  key = {
    process = function(value, input, result, unknown)
      value = input.one + input.two
      result.one = nil
      result.two = nil
      return value
    end,
  },
})
local result = parse('key,one=1,two=2') -- { key = 3 }
```

The following example demonstrates the `result` parameter:

```lua
local parse = luakeys.define({
  key = {
    process = function(value, input, result, unknown)
      result.additional_key = true
      return value
    end,
  },
})
local result = parse('key=1') -- { key = 1, additional_key = true }
```

The following example demonstrates the `unknown` parameter:

```lua
local parse = luakeys.define({
  key = {
    process = function(value, input, result, unknown)
      unknown.unknown_key = true
      return value
    end,
  },
})
```

```lua
    parse('key=1') -- throws error message: 'Unknown keys: unknown_key=true,'
```

3.7.13 Attribute "`required`"

The `required` attribute can be used to enforce that a specific key must be specified. In the example below, the key `important` is defined as mandatory.

```lua
local parse = luakeys.define({ important = { required = true } })
local result = parse('important') -- { important = true }
```

If the key `important` is missing in the input, an error message occurs.

```lua
    parse('unimportant')
    -- throws error message: 'Missing required key "important"!'
```

A recursive example:

```lua
local parse2 = luakeys.define({
  important1 = {
    required = true,
    sub_keys = { important2 = { required = true } },
  },
})
```

The `important2` key on level 2 is missing.

```lua
parse2('important1={unimportant}')
-- throws error message: 'Missing required key "important2"!'
```

The `important1` key at the lowest key level is missing.

```lua
parse2('unimportant')
-- throws error message: 'Missing required key "important1"!'
```

3.7.14 Attribute "sub_keys"

The `sub_keys` attribute can be used to build nested key-value pair definitions.

```lua
local result, unknown = luakeys.parse('level1={level2,unknown}', {
  no_error = true,
  defs = {
    level1 = {
      sub_keys = {
        level2 = { default = 42 }
      }
    }
  },
})
luakeys.debug(result) -- { level1 = { level2 = 42 } }
luakeys.debug(unknown) -- { level1 = { 'unknown' } }
```

3.8 Function "render(result): string"

The function `render(result)` 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.

```lua
local result = luakeys.parse('one=1,two=2,three=3,')
local kv_string = 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.

```lua
local result2 = luakeys.parse('one,two,three', { naked_as_value = true })
local kv_string2 = luakeys.render(result2) --- one,two,three, (always)
```

3.9 Function “debug(result): void”

The function `debug(result)` 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.

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

The output should look like this:

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

3.10 Function “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.

3.11 Function “get(identifier): result”

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

3.12 Table “is”

In the table `is` some functions are summarized, which check whether an input corresponds to a certain data type. All functions accept not only the corresponding Lua data types, but also input as strings. For example, the string `'true'` is recognized by the `is.boolean()` function as a boolean value.

3.12.1 Function “is.boolean(value): boolean”

```
  -- true
  equal(luakeys.is.boolean('true'), true) -- input: string!
  equal(luakeys.is.boolean('True'), true) -- input: string!
  equal(luakeys.is.boolean('TRUE'), true) -- input: string!
  equal(luakeys.is.boolean('false'), true) -- input: string!
  equal(luakeys.is.boolean('False'), true) -- input: string!
  equal(luakeys.is.boolean('FALSE'), true) -- input: string!
  equal(luakeys.is.boolean(true), true)
  equal(luakeys.is.boolean(false), true)
  -- false
  equal(luakeys.is.boolean('xxx'), false)
  equal(luakeys.is.boolean('trueX'), false)
  equal(luakeys.is.boolean('1'), false)
  equal(luakeys.is.boolean('0'), false)
  equal(luakeys.is.boolean(1), false)
  equal(luakeys.is.boolean(0), false)
```

```
  equal(luakeys.is.boolean(nil), false)
end)
```

3.12.2 Function "is.dimension(value): boolean"

```
-- true
equal(luakeys.is.dimension('1 cm'), true)
equal(luakeys.is.dimension('- 1 mm'), true)
equal(luakeys.is.dimension('-1.1pt'), true)
-- false
equal(luakeys.is.dimension('1cmX'), false)
equal(luakeys.is.dimension('X1cm'), false)
equal(luakeys.is.dimension(1), false)
equal(luakeys.is.dimension('1'), false)
equal(luakeys.is.dimension('xxx'), false)
equal(luakeys.is.dimension(nil), false)
```

3.12.3 Function "is.integer(value): boolean"

```
-- true
equal(luakeys.is.integer('42'), true) -- input: string!
equal(luakeys.is.integer(1), true)
-- false
equal(luakeys.is.integer('1.1'), false)
equal(luakeys.is.integer('xxx'), false)
```

3.12.4 Function "is.number(value): boolean"

```
-- true
equal(luakeys.is.number('1'), true) -- input: string!
equal(luakeys.is.number('1.1'), true) -- input: string!
equal(luakeys.is.number(1), true)
equal(luakeys.is.number(1.1), true)
-- false
equal(luakeys.is.number('xxx'), false)
equal(luakeys.is.number('1cm'), false)
```

3.12.5 Function "is.string(value): boolean"

```
-- true
equal(luakeys.is.string('string'), true)
equal(luakeys.is.string(''), true)
-- false
equal(luakeys.is.string(true), false)
equal(luakeys.is.string(1), false)
equal(luakeys.is.string(nil), false)
```

3.12.6 Function "is.any(value): boolean"

The function `is.any(value)` always returns `true` and therefore accepts any data type.

3.13 Table "utils"

3.13.1 Function "utils.scan_oarg(initial_delimiter?, end_delimiter?): string"

Plain TeX does not know optional arguments (oarg). The function `utils.scan_oarg(initial_delimiter?, end_delimiter?): string` allows to search for optional arguments not only in LaTeX but also in Plain TeX. The function uses the token library built into LuaTeX. The two parameters `initial_delimiter` and `end_delimiter` can be omitted. Then square brackets are assumed to be delimiters. For example, this Lua code `utils.scan_oarg('(', ')')` searches for an optional argument in round brackets The function returns the string between the delimiters or `nil` if no delimiters could be found. The delimiters themselves are not included in the result. After the `\directlua {}`, the macro using `utils.scan_oarg` must not expand to any characters.

```
\input luakeys.tex

\def\mycmd{\directlua{
    local oarg = luakeys.utils.scan_oarg('[', ']')
    if oarg then
      local keys = luakeys.parse(oarg)
      for key, value in pairs(keys) do
        tex.print('oarg: key: "' .. key .. '" value: "' .. value .. '";')
      end
    end
    local marg = token.scan_argument()
    tex.print('marg: "' .. marg .. '"')
  }% <- important
}

\mycmd[key=value]{marg}
% oarg: key: "key" value: "value"; marg: "marg"

\mycmd{marg without oarg}
% marg: "marg without oarg"

end
\bye
```

3.14 Table "version"

The luakeys project uses semantic versioning. The three version numbers of the semantic versioning scheme are stored in a table as integers in the order MAJOR, MINOR, PATCH. This table can be used to check whether the correct version is installed.

```
local v = luakeys.version
local version_string = v[1] .. '.' .. v[2] .. '.' .. v[3]
print(version_string) -- 0.7.0

if v[1] >= 1 and v[2] > 2 then
  print('You are using the right version.')
end
```

4 Syntax of the recognized key-value format

4.1 An attempt to put the syntax into words

A key-value pair is definied by an equal sign (`key=value`). Several key-value pairs or keys without values (naked keys) are lined up with commas (`key=value,naked`) 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,naked}}`).

4.2 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*⟩

⟨*dimension*⟩ ::= ⟨*number*⟩ ⟨*unit*⟩

⟨*number*⟩ ::= ⟨*sign*⟩ (⟨*integer*⟩ [⟨*fractional*⟩] | ⟨*fractional*⟩)

⟨*fractional*⟩ ::= ‘.’ ⟨*integer*⟩

⟨*sign*⟩ ::= ‘-’ | ‘+’

⟨*integer*⟩ ::= ⟨*digit*⟩ { ⟨*digit*⟩ }

⟨*digit*⟩ ::= ‘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’
| ‘mu’ | ‘MU’
| ‘nc’ | ‘NC’
| ‘nd’ | ‘ND’
| ‘pc’ | ‘PC’
| ‘pt’ | ‘PT’

| 'px' | 'PX'
| 'sp' | 'SP'

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

⟨*boolean-true*⟩ ::= 'true' | 'TRUE' | 'True'

⟨*boolean-false*⟩ ::= 'false' | 'FALSE' | 'False'

... to be continued

4.3 Recognized data types

4.3.1 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,
}
```

4.3.2 number

```
\luakeysdebug{
  num0 = 042,
  num1 = 42,
  num2 = -42,
  num3 = 4.2,
  num4 = 0.42,
  num5 = .42,
  num6 = 0 . 42,
}
```

```
{
  ['num0'] = 42,
  ['num1'] = 42,
  ['num2'] = -42,
  ['num3'] = 4.2,
  ['num4'] = 0.42,
  ['num5'] = 0.42,
  ['num6'] = '0 . 42', -- string
}
```

4.3.3 dimension

`luakeys` tries to recognize all units used in the TeX world. According to the LuaTeX source code (source/texk/web2c/luatexdir/lua/ltexlib.c) and the dimension module of the lualibs library (lualibs-util-dim.lua), all units should be recognized.

	Description
bp	big point
cc	cicero
cm	centimeter
dd	didot
em	horizontal measure of M
ex	vertical measure of x
in	inch
mm	milimeter
mu	math unit
nc	new cicero
nd	new didot
pc	pica
pt	point
px	x height current font
sp	scaledpoint

```
\luakeysdebug[convert_dimensions=true]{
  bp = 1bp,
  cc = 1cc,
  cm = 1cm,
  dd = 1dd,
  em = 1em,
  ex = 1ex,
  in = 1in,
  mm = 1mm,
  mu = 1mu,
  nc = 1nc,
  nd = 1nd,
  pc = 1pc,
  pt = 1pt,
  px = 1px,
  sp = 1sp,
}
```

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

The next example illustrates the different notations of the dimensions.

```
\luakeysdebug[convert_dimensions=true]{
  upper = 1CM,
  lower = 1cm,
  space = 1 cm,
  plus = + 1cm,
  minus = -1cm,
  nodim = 1 c m,
}
```

```
{
  ['upper'] = 1864679,
  ['lower'] = 1864679,
  ['space'] = 1864679,
  ['plus'] = 1864679,
  ['minus'] = -1864679,
  ['nodim'] = '1 c m', -- string
}
```

4.3.4 string

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

```
local kv_string = [[
  without double quotes = no commas and equal signs are allowed,
  with double quotes = ", and = are allowed",
  escape quotes = "a quote \" sign",
  curly braces = "curly { } braces are allowed",
]]
local result = luakeys.parse(kv_string)
luakeys.debug(result)
-- {
--   ['without double quotes'] = 'no commas and equal signs are allowed',
--   ['with double quotes'] = ', and = are allowed',
--   ['escape quotes'] = 'a quote \" sign',
--   ['curly braces'] = 'curly { } braces are allowed',
-- }
```

4.3.5 Naked keys

Naked keys are keys without a value. Using the option `naked_as_value` they can be converted into values and stored into an array. In Lua an array is a table with numeric indexes (The first index is 1).

```
\luakeysdebug[naked_as_value=true]{one,two,three}
% {
%   [1] = 'one',
%   [2] = 'two',
%   [3] = 'three',
% }
% =
% { 'one', 'two', 'three' }
```

All recognized data types can be used as standalone values.

```
\luakeysdebug[naked_as_value=true]{one,2,3cm}
% {
%   [1] = 'one',
%   [2] = 2,
%   [3] = '3cm',
% }
```

5 Examples

5.1 Extend and modify keys of existing macros

Extend the includegraphics macro with a new key named `caption` and change the accepted values of the `width` key. A number between 0 and 1 is allowed and converted into `width=0.5\linewidth`

```
local luakeys = require('luakeys')

local parse = luakeys.define({
  caption = { alias = 'title' },
  width = {
    process = function(value)
      if type(value) == 'number' and value >= 0 and value <= 1 then
        return tostring(value) .. '\\linewidth'
      end
      return value
    end,
  },
})

local function print_image_macro(image_path, kv_string)
  local caption = ''
  local options = ''
  local keys, unknown = parse(kv_string)
  if keys['caption'] ~= nil then
    caption = '\\ImageCaption{' .. keys['caption'] .. '}'
  end
  if keys['width'] ~= nil then
    unknown['width'] = keys['width']
  end
  options = luakeys.render(unknown)

  tex.print('\\includegraphics[' .. options .. ']{' .. image_path .. '}' ..
              caption)
end

return print_image_macro
```

```
\documentclass{article}
\usepackage{graphicx}
\begin{document}
\newcommand{\ImageCaption}[1]{%
  \par\textit{#1}%
}

\newcommand{\myincludegrahics}[2][]{
  \directlua{
    print_image_macro = require('extend-keys.lua')
    print_image_macro('#2', '#1')
  }
}

\myincludegrahics{test.png}

\myincludegrahics[width=0.5]{test.png}

\myincludegrahics[caption=A caption]{test.png}
```

```
\end{document}
```

5.2 Process document class options

The options of a LaTeX document class can be accessed via the `\@classoptionslist` macro. The string of the macro `\@classoptionslist` is already expanded and normalized. In addition to spaces, the curly brackets are also removed. It is therefore not possible to process nested hierarchical key-value pairs via the option.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{test-class}[2022/05/26 Test class to access the class options]
\DeclareOption*{\PassOptionsToClass{\CurrentOption}{article}}
\ProcessOptions\relax
\directlua{
  luakeys = require('luakeys')
  local kv = luakeys.parse('\@classoptionslist', {
    convert_dimensions = false,
    naked_as_value = true
  })
  luakeys.debug(kv)
}
\LoadClass{article}
```

```
\documentclass[12pt,landscape]{test-class}

\begin{document}
This document uses the class "test-class".
\end{document}
```

```
{
  [1] = '12pt',
  [2] = 'landscape',
}
```

6 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:

```
{
  ['one'] = true,
  ['three'] = true,
  ['two'] = true,
}
```

6.1 For plain TeX: luakeys-debug.tex

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

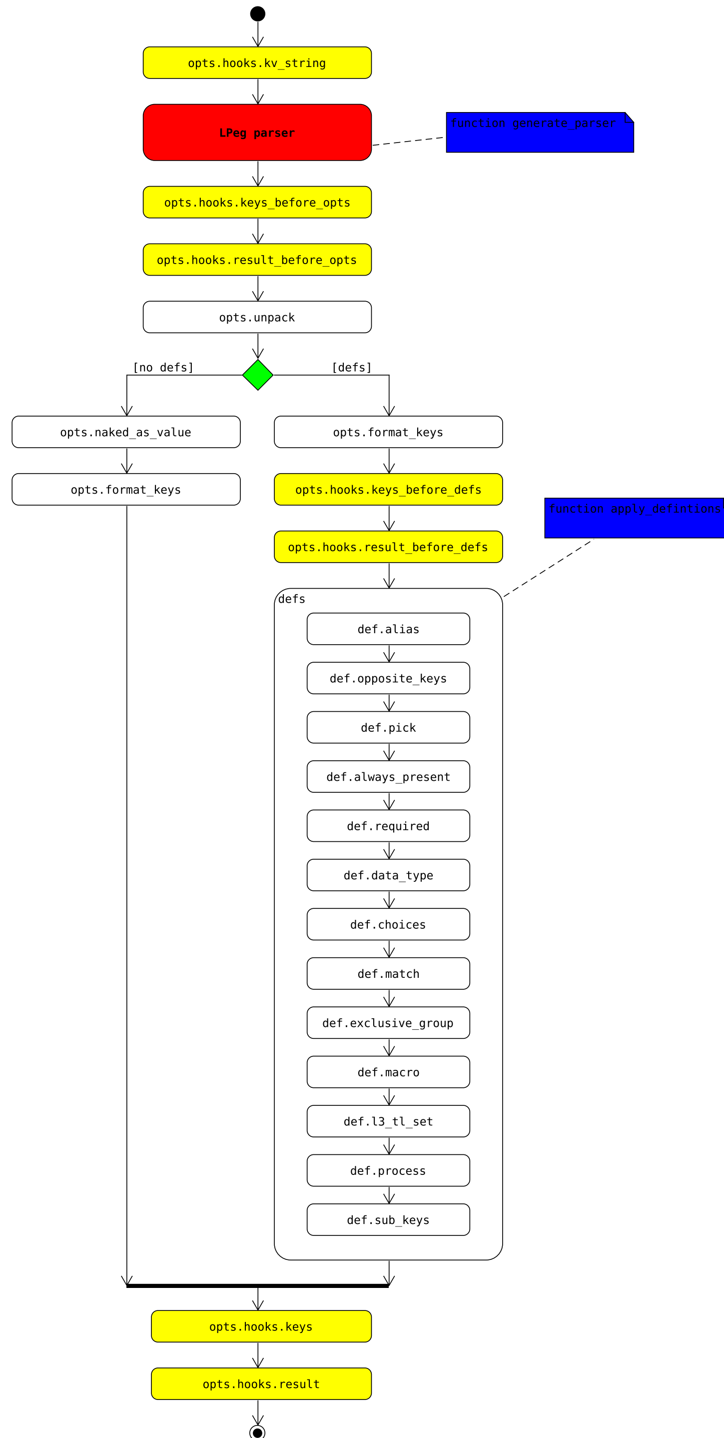
```
\input luakeys-debug.tex
\luakeysdebug{one,two,three}
\bye
```

6.2 For LaTeX: luakeys-debug.sty

An example of how to use the command in LaTeX:

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

7 Activity diagramm of the parsing process



8 Implementation

8.1 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.
--
-- @module luakeys
local lpeg = require('lpeg')

if not tex then
  tex = {
    -- Dummy function for the tests.
    sp = function(input)
      return 1234567
    end,
  }
end

if not token then
  token = {
    set_macro = function(csname, content, global)
    end,
  }
end

--- Merge two tables in the first specified table.
--- The `merge_tables` function copies all keys from the `source` table
--- to a target table. It returns the modified target table.
---
---@see https://stackoverflow.com/a/1283608/10193818
---
---@param target table
---@param source table
---
---@return table target The modified target table.
local function merge_tables(target, source)
  for key, value in pairs(source) do
    if type(value) == 'table' then
      if type(target[key] or false) == 'table' then
        merge_tables(target[key] or {}, source[key] or {})
      elseif target[key] == nil then
        target[key] = value
      end
    elseif target[key] == nil then
      target[key] = value
```

```
59      end
60    end
61    return target
62  end
63
64  ---Clone a table.
65  ---
66  ---@see http://lua-users.org/wiki/CopyTable
67  ---
68  ---@param orig table
69  ---
70  ---@return table
71  local function clone_table(orig)
72    local orig_type = type(orig)
73    local copy
74    if orig_type == 'table' then
75      copy = {}
76      for orig_key, orig_value in next, orig, nil do
77        copy[clone_table(orig_key)] = clone_table(orig_value)
78      end
79      setmetatable(copy, clone_table(getmetatable(orig)))
80    else -- number, string, boolean, etc
81      copy = orig
82    end
83    return copy
84  end
85
86  local utils = {
87    --- Get the size of an array like table `{ 'one', 'two', 'three' }` = 3.
88    ---
89    ---@param value table # A table or any input.
90    ---
91    ---@return number # The size of the array like table. 0 if the input is no table
      ↪  or the table is empty.
92    get_array_size = function(value)
93      local count = 0
94      if type(value) == 'table' then
95        for _ in ipairs(value) do
96          count = count + 1
97        end
98      end
99      return count
100   end,
101
102   ---Get the size of a table `{ one = 'one', 'two', 'three' }` = 3.
103   ---
104   ---@param value table|any # A table or any input.
105   ---
106   ---@return number # The size of the array like table. 0 if the input is no table
      ↪  or the table is empty.
107   get_table_size = function(value)
108     local count = 0
109     if type(value) == 'table' then
110       for _ in pairs(value) do
111         count = count + 1
112       end
113     end
114     return count
115   end,
116
117   merge_tables = merge_tables,
118
```

```
119     clone_table = clone_table,
120
121     remove_from_array = function(array, element)
122       for index, value in pairs(array) do
123         if element == value then
124           array[index] = nil
125           return value
126         end
127       end
128     end,
129
130     ---Scan for an optional argument.
131     ---
132     ---@param initial_delimiter? string # The character that marks the beginning of an
        ↪  optional argument (by default `[`).
133     ---@param end_delimiter? string # The character that marks the end of an optional
        ↪  argument (by default `]`).
134     ---
135     ---@return string|nil # The string that was enclosed by the delimiters. The
        ↪  delimiters themselves are not returned.
136     scan_oarg = function(initial_delimiter, end_delimiter)
137       if initial_delimiter == nil then
138         initial_delimiter = '['
139       end
140
141       if end_delimiter == nil then
142         end_delimiter = ']'
143       end
144
145       local function convert_token(t)
146         if t.index ~= nil then
147           return utf8.char(t.index)
148         else
149           return '\\' .. t.csname
150         end
151       end
152
153       local function get_next_char()
154         local t = token.get_next()
155         return convert_token(t), t
156       end
157
158       local char, t = get_next_char()
159       if char == initial_delimiter then
160         local oarg = {}
161         char = get_next_char()
162         while char ~= end_delimiter do
163           table.insert(oarg, char)
164           char = get_next_char()
165         end
166         return table.concat(oarg, '')
167       else
168         token.put_next(t)
169       end
170     end,
171   }
172
173   local namespace = {
174     opts = {
175       assignment_operator = '=',
176       convert_dimensions = false,
177       debug = false,
```

```
178      default = true,
179      defaults = false,
180      defs = false,
181      false_aliases = { 'false', 'FALSE', 'False' },
182      format_keys = false,
183      group_begin = '{',
184      group_end = '}',
185      hooks = {},
186      invert_flag = '!',
187      list_separator = ',',
188      naked_as_value = false,
189      no_error = false,
190      quotation_begin = '"',
191      quotation_end = '"',
192      true_aliases = { 'true', 'TRUE', 'True' },
193      unpack = true,
194    },
195
196    hooks = {
197      kv_string = true,
198      keys_before_opts = true,
199      result_before_opts = true,
200      keys_before_def = true,
201      result_before_def = true,
202      keys = true,
203      result = true,
204    },
205
206    attrs = {
207      alias = true,
208      always_present = true,
209      choices = true,
210      data_type = true,
211      default = true,
212      exclusive_group = true,
213      l3_tl_set = true,
214      macro = true,
215      match = true,
216      name = true,
217      opposite_keys = true,
218      pick = true,
219      process = true,
220      required = true,
221      sub_keys = true,
222    },
223  }
224
225  --- The default options.
226  local default_options = clone_table(namespace.opts)
227
228  local function throw_error(message)
229    if type(tex.error) == 'function' then
230      tex.error(message)
231    else
232      error(message)
233    end
234  end
235
236  --- Normalize the parse options.
237  ---
238  ---@param opts? table # Options in a raw format. The table may be empty or some keys
     ↪  are not set.
```

```
239   ---
240   ---@return table
241   local function normalize_opts(opts)
242     if type(opts) ~= 'table' then
243       opts = {}
244     end
245     for key, _ in pairs(opts) do
246       if namespace.opts[key] == nil then
247         throw_error('Unknown parse option: ' .. tostring(key) .. '!')
248       end
249     end
250     local old_opts = opts
251     opts = {}
252     for name, _ in pairs(namespace.opts) do
253       if old_opts[name] ~= nil then
254         opts[name] = old_opts[name]
255       else
256         opts[name] = default_options[name]
257       end
258     end
259
260     for hook in pairs(opts.hooks) do
261       if namespace.hooks[hook] == nil then
262         throw_error('Unknown hook: ' .. tostring(hook) .. '!')
263       end
264     end
265     return opts
266   end
267
268   local l3_code_cctab = 10
269
270   --- Convert back to strings
271   -- @section
272
273   --- The function `render(tbl)` reverses the function
274   --- `parse(kv_string)`. It takes a Lua table and converts this table
275   --- into a key-value string. The resulting string usually has a
276   --- different order as the input table. In Lua only tables with
277   --- 1-based consecutive integer keys (a.k.a. array tables) can be
278   --- parsed in order.
279   ---
280   ---@param result table # A table to be converted into a key-value string.
281   ---
282   ---@return string # A key-value string that can be passed to a TeX macro.
283   local function render(result)
284     local function render_inner(result)
285       local output = {}
286       local function add(text)
287         table.insert(output, text)
288       end
289       for key, value in pairs(result) do
290         if (key and type(key) == 'string') then
291           if (type(value) == 'table') then
292             if (next(value)) then
293               add(key .. '={')
294               add(render_inner(value))
295               add('},')
296             else
297               add(key .. '={},')
298             end
299           else
300             add(key .. '=' .. tostring(value) .. ',')
```

```
301         end
302       else
303         add(tostring(value) .. ',')
304       end
305     end
306     return table.concat(output)
307   end
308   return render_inner(result)
309 end
310
311 --- The function `stringify(tbl, for_tex)` converts a Lua table into a
312 --- printable string. Stringify a table means to convert the table into
313 --- a string. This function is used to realize the `debug` function.
314 --- `stringify(tbl, true)` (`for_tex = true`) generates a string which
315 --- can be embeded into TeX documents. The macro `\luakeysdebug{}` uses
316 --- this option. `stringify(tbl, false)` or `stringify(tbl)` generate a
317 --- string suitable for the terminal.
318 ---
319 ---@see https://stackoverflow.com/a/54593224/10193818
320 ---
321 ---@param result table # A table to stringify.
322 ---@param for_tex? boolean # 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.
323 ---
324 ---@return string
325 local function stringify(result, for_tex)
326   local line_break, start_bracket, end_bracket, indent
327
328   if for_tex then
329     line_break = '\\par'
330     start_bracket = '$\\{$'
331     end_bracket = '$\\}$'
332     indent = '\\ \\ '
333   else
334     line_break = '\n'
335     start_bracket = '{'
336     end_bracket = '}'
337     indent = '  '
338   end
339
340   local function stringify_inner(input, depth)
341     local output = {}
342     depth = depth or 0
343
344     local function add(depth, text)
345       table.insert(output, string.rep(indent, depth) .. text)
346     end
347
348     local function format_key(key)
349       if (type(key) == 'number') then
350         return string.format('[%s]', key)
351       else
352         return string.format('[\'%s\']', key)
353       end
354     end
355
356     if type(input) ~= 'table' then
357       return tostring(input)
358     end
359
360     for key, value in pairs(input) do
```

```
361          if (key and type(key) == 'number' or type(key) == 'string') then
362            key = format_key(key)
363
364            if (type(value) == 'table') then
365              if (next(value)) then
366                add(depth, key .. ' = ' .. start_bracket)
367                add(0, stringify_inner(value, depth + 1))
368                add(depth, end_bracket .. ',');
369              else
370                add(depth,
371                  key .. ' = ' .. start_bracket .. end_bracket .. ',')
372              end
373            else
374              if (type(value) == 'string') then
375                value = string.format('\'%s\'', value)
376              else
377                value = tostring(value)
378              end
379
380              add(depth, key .. ' = ' .. value .. ',')
381            end
382          end
383        end
384
385        return table.concat(output, line_break)
386      end
387
388      return start_bracket .. line_break .. stringify_inner(result, 1) ..
389               line_break .. end_bracket
390    end
391
392    --- The function `debug(result)` pretty prints a Lua table to standard
393    --   output (stdout). It is a utility function that can be used to
394    --   debug and inspect the resulting Lua table of the function
395    --   `parse`. You have to compile your TeX document in a console to
396    --   see the terminal output.
397    --
398    ---@param result table # A table to be printed to standard output for debugging
       ↪  purposes.
399    local function debug(result)
400      print('\n' .. stringify(result, false))
401    end
402
403    --- Parser / Lpeg related
404    -- @section
405
406    --- Generate the PEG parser using Lpeg.
407    ---
408    --- Explanations of some LPeg notation forms:
409    ---
410    --- * `patt ^ 0` = `expression *`
411    --- * `patt ^ 1` = `expression +`
412    --- * `patt ^ -1` = `expression ?`
413    --- * `patt1 * patt2` = `expression1 expression2`: Sequence
414    --- * `patt1 + patt2` = `expression1 / expression2`: Ordered choice
415    ---
416    --- * [TUGboat article: Parsing complex data formats in LuaTEX with
       ↪  LPEG](https://tug.or-g/TUGboat/tb40-2/tb125menke-Patterndf)
417    ---
418    ---@param initial_rule string # The name of the first rule of the grammar table
       ↪  passed to the `lpeg.P(attern)` function (e. g. `list`, `number`).
419    ---@param opts? table # Whether the dimensions should be converted to scaled points
       ↪  (by default `false`).
```

```
420   ---
421   ---@return userdata # The parser.
422   local function generate_parser(initial_rule, opts)
423     if type(opts) ~= 'table' then
424       opts = normalize_opts(opts)
425     end
426
427     local Variable = lpeg.V
428     local Pattern = lpeg.P
429     local Set = lpeg.S
430     local Range = lpeg.R
431     local CaptureGroup = lpeg.Cg
432     local CaptureFolding = lpeg.Cf
433     local CaptureTable = lpeg.Ct
434     local CaptureConstant = lpeg.Cc
435     local CaptureSimple = lpeg.C
436
437     -- Optional whitespace
438     local white_space = Set(' \t\n\r')
439
440     --- Match literal string surrounded by whitespace
441     local ws = function(match)
442       return white_space ^ 0 * Pattern(match) * white_space ^ 0
443     end
444
445     local line_up_pattern = function(patterns)
446       local result
447       for _, pattern in ipairs(patterns) do
448         if result == nil then
449           result = Pattern(pattern)
450         else
451           result = result + Pattern(pattern)
452         end
453       end
454       return result
455     end
456
457     --- Convert a dimension to an normalized dimension string or an
458     --- integer in the scaled points format.
459     ---
460     ---@param input string
461     ---
462     ---@return integer|string # A dimension as an integer or a dimension string.
463     local capture_dimension = function(input)
464       -- Remove all whitespaces
465       input = input:gsub('%s+', '')
466       -- Convert the unit string into lowercase.
467       input = input:lower()
468       if opts.convert_dimensions then
469         return tex.sp(input)
470       else
471         return input
472       end
473     end
474
475     --- Add values to a table in two modes:
476     --
477     -- Key-value pair:
478     --
479     -- If `arg1` and `arg2` are not nil, then `arg1` is the key and `arg2` is the
480     -- value of a new table entry.
481     --
```

```
482  -- Indexed value:
483  --
484  -- If `arg2` is nil, then `arg1` is the value and is added as an indexed
485  -- (by an integer) value.
486  --
487  ---@param result table # The result table to which an additional key-value pair or
     ↪  value should to be added
488  ---@param arg1 any # The key or the value.
489  ---@param arg2? any # Always the value.
490  ---
491  ---@return table # The result table to which an additional key-value pair or value
     ↪  has been added.
492  local add_to_table = function(result, arg1, arg2)
493    if arg2 == nil then
494      local index = #result + 1
495      return rawset(result, index, arg1)
496    else
497      return rawset(result, arg1, arg2)
498    end
499  end
500
501  -- LuaFormatter off
502  return Pattern({
503    [1] = initial_rule,
504
505    -- list_item*
506    list = CaptureFolding(
507      CaptureTable('') * Variable('list_item')^0,
508      add_to_table
509    ),
510
511    -- '{' list '}'
512    list_container =
513      ws(opts.group_begin) * Variable('list') * ws(opts.group_end),
514
515    -- ( list_container / key_value_pair / value ) ','?
516    list_item =
517      CaptureGroup(
518        Variable('list_container') +
519        Variable('key_value_pair') +
520        Variable('value')
521      ) * ws(opts.list_separator)^-1,
522
523    -- key '=' (list_container / value)
524    key_value_pair =
525      (Variable('key') * ws(opts.assignment_operator)) *
       ↪  (Variable('list_container') + Variable('value')),
526
527    -- number / string_quoted / string_unquoted
528    key =
529      Variable('number') +
530      Variable('string_quoted') +
531      Variable('string_unquoted'),
532
533    -- boolean !value / dimension !value / number !value / string_quoted !value /
       ↪  string_unquoted
534    -- !value -> Not-predicate -> * -Variable('value')
535    value =
536      Variable('boolean') * -Variable('value') +
537      Variable('dimension') * -Variable('value') +
538      Variable('number') * -Variable('value')  +
539      Variable('string_quoted') * -Variable('value') +
```

```
540        Variable('string_unquoted'),
541
542      -- for is.boolean()
543      boolean_only = Variable('boolean') * -1,
544
545      -- boolean_true / boolean_false
546      boolean =
547        (
548          Variable('boolean_true') * CaptureConstant(true) +
549          Variable('boolean_false') * CaptureConstant(false)
550        ),
551
552      boolean_true = line_up_pattern(opts.true_aliases),
553
554      boolean_false = line_up_pattern(opts.false_aliases),
555
556      -- for is.dimension()
557      dimension_only = Variable('dimension') * -1,
558
559      dimension = (
560        Variable('tex_number') * white_space^0 *
561        Variable('unit')
562      ) / capture_dimension,
563
564      -- for is.number()
565      number_only = Variable('number') * -1,
566
567      -- capture number
568      number = Variable('tex_number') / tonumber,
569
570      -- sign? white_space? (integer+ fractional? / fractional)
571      tex_number =
572        Variable('sign')^0 * white_space^0 *
573        (Variable('integer')^1 * Variable('fractional')^0) +
574        Variable('fractional'),
575
576      sign = Set('-+'),
577
578      fractional = Pattern('.') * Variable('integer')^1,
579
580      integer = Range('09')^1,
581
582      -- 'bp' / 'BP' / 'cc' / etc.
583      -- https://raw.githubusercontent.com/latex3/lualibs/master/lualibs-util-dim.lua
584      -- https://github.com/TeX-
         ↪  Live/luatex/blob/51db1985f5500dafd2393aa2e403fefa57d3cb76/source/texk/web2c/luatexdir/lua/ltexlib.c
         ↪  L625
585      unit =
586        Pattern('bp') + Pattern('BP') +
587        Pattern('cc') + Pattern('CC') +
588        Pattern('cm') + Pattern('CM') +
589        Pattern('dd') + Pattern('DD') +
590        Pattern('em') + Pattern('EM') +
591        Pattern('ex') + Pattern('EX') +
592        Pattern('in') + Pattern('IN') +
593        Pattern('mm') + Pattern('MM') +
594        Pattern('mu') + Pattern('MU') +
595        Pattern('nc') + Pattern('NC') +
596        Pattern('nd') + Pattern('ND') +
597        Pattern('pc') + Pattern('PC') +
598        Pattern('pt') + Pattern('PT') +
599        Pattern('px') + Pattern('PX') +
```

```
600       Pattern('sp') + Pattern('SP'),
601
602     -- '"' ('\"' / !'"')* '"'
603     string_quoted =
604       white_space^0 * Pattern(opts.quotation_begin) *
605       CaptureSimple((Pattern('\\' .. opts.quotation_end) + 1 -
      ↪   Pattern(opts.quotation_end))^0) *
606       Pattern(opts.quotation_end) * white_space^0,
607
608     string_unquoted =
609       white_space^0 *
610       CaptureSimple(
611         Variable('word_unquoted')^1 *
612         (Set(' \t')^1 * Variable('word_unquoted')^1)^0) *
613       white_space^0,
614
615     word_unquoted = (1 - white_space - Set(
616       opts.group_begin ..
617       opts.group_end ..
618       opts.assignment_operator  ..
619       opts.list_separator))^1
620   })
621 -- LuaFormatter on
622 end
623
624 local function visit_tree(tree, callback_func)
625   if type(tree) ~= 'table' then
626     throw_error('Parameter "tree" has to be a table, got: ' ..
627                   tostring(tree))
628   end
629   local function visit_tree_recursive(tree,
630     current,
631     result,
632     depth,
633     callback_func)
634     for key, value in pairs(current) do
635       if type(value) == 'table' then
636         value = visit_tree_recursive(tree, value, {}, depth + 1,
637           callback_func)
638       end
639
640       key, value = callback_func(key, value, depth, current, tree)
641
642       if key ~= nil and value ~= nil then
643         result[key] = value
644       end
645     end
646     if next(result) ~= nil then
647       return result
648     end
649   end
650
651   local result = visit_tree_recursive(tree, tree, {}, 1, callback_func)
652
653   if result == nil then
654     return {}
655   end
656   return result
657 end
658
659 local is = {
660   boolean = function(value)
```

```
661      if value == nil then
662        return false
663      end
664      if type(value) == 'boolean' then
665        return true
666      end
667      local parser = generate_parser('boolean_only')
668      local result = parser:match(tostring(value))
669      return result ~= nil
670    end,
671
672    dimension = function(value)
673      if value == nil then
674        return false
675      end
676      local parser = generate_parser('dimension_only')
677      local result = parser:match(tostring(value))
678      return result ~= nil
679    end,
680
681    integer = function(value)
682      local n = tonumber(value)
683      if n == nil then
684        return false
685      end
686      return n == math.floor(n)
687    end,
688
689    number = function(value)
690      if value == nil then
691        return false
692      end
693      if type(value) == 'number' then
694        return true
695      end
696      local parser = generate_parser('number_only')
697      local result = parser:match(tostring(value))
698      return result ~= nil
699    end,
700
701    string = function(value)
702      return type(value) == 'string'
703    end,
704
705    any = function(value)
706      return true
707    end,
708  }
709
710  --- Apply the key-value-pair definitions (defs) on an input table in a
711  --- recursive fashion.
712  ---
713  ---@param defs table # A table containing all definitions.
714  ---@param opts table # The parse options table.
715  ---@param input table # The current input table.
716  ---@param output table # The current output table.
717  ---@param unknown table # Always the root unknown table.
718  ---@param key_path table # An array of key names leading to the current
719  ---@param input_root table # The root input table input and output table.
720  local function apply_definitions(defs,
721    opts,
722    input,
```

```
723   output,
724   unknown,
725   key_path,
726   input_root)
727   local exclusive_groups = {}
728
729   local function add_to_key_path(key_path, key)
730     local new_key_path = {}
731
732     for index, value in ipairs(key_path) do
733       new_key_path[index] = value
734     end
735
736     table.insert(new_key_path, key)
737     return new_key_path
738   end
739
740   local function get_default_value(def)
741     if def.default ~= nil then
742       return def.default
743     elseif opts ~= nil and opts.default ~= nil then
744       return opts.default
745     end
746     return true
747   end
748
749   local function find_value(search_key, def)
750     if input[search_key] ~= nil then
751       local value = input[search_key]
752       input[search_key] = nil
753       return value
754       -- naked keys: values with integer keys
755     elseif utils.remove_from_array(input, search_key) ~= nil then
756       return get_default_value(def)
757     end
758   end
759
760   local apply = {
761     alias = function(value, key, def)
762       if type(def.alias) == 'string' then
763         def.alias = { def.alias }
764       end
765       local alias_value
766       local used_alias_key
767       -- To get an error if the key and an alias is present
768       if value ~= nil then
769         alias_value = value
770         used_alias_key = key
771       end
772       for _, alias in ipairs(def.alias) do
773         local v = find_value(alias, def)
774         if v ~= nil then
775           if alias_value ~= nil then
776             throw_error(string.format(
777               'Duplicate aliases "%s" and "%s" for key "%s"!',
778               used_alias_key, alias, key))
779           end
780           used_alias_key = alias
781           alias_value = v
782         end
783       end
784       if alias_value ~= nil then
```

```
785           return alias_value
786         end
787       end,
788
789       always_present = function(value, key, def)
790         if value == nil and def.always_present then
791           return get_default_value(def)
792         end
793       end,
794
795       choices = function(value, key, def)
796         if value == nil then
797           return
798         end
799         if def.choices ~= nil and type(def.choices) == 'table' then
800           local is_in_choices = false
801           for _, choice in ipairs(def.choices) do
802             if value == choice then
803               is_in_choices = true
804             end
805           end
806           if not is_in_choices then
807             throw_error('The value "' .. value ..
808                            '" does not exist in the choices: ' ..
809                            table.concat(def.choices, ', ') .. '!')
810           end
811         end
812       end,
813
814       data_type = function(value, key, def)
815         if value == nil then
816           return
817         end
818         if def.data_type ~= nil then
819           local converted
820           -- boolean
821           if def.data_type == 'boolean' then
822             if value == 0 or value == '' or not value then
823               converted = false
824             else
825               converted = true
826             end
827             -- dimension
828           elseif def.data_type == 'dimension' then
829             if is.dimension(value) then
830               converted = value
831             end
832             -- integer
833           elseif def.data_type == 'integer' then
834             if is.number(value) then
835               local n = tonumber(value)
836               if type(n) == 'number' and n ~= nil then
837                 converted = math.floor(n)
838               end
839             end
840             -- number
841           elseif def.data_type == 'number' then
842             if is.number(value) then
843               converted = tonumber(value)
844             end
845             -- string
846           elseif def.data_type == 'string' then
```

```
847           converted = tostring(value)
848         else
849           throw_error('Unknown data type: ' .. def.data_type)
850         end
851         if converted == nil then
852           throw_error(
853             'The value "' .. value .. '" of the key "' .. key ..
854               '" could not be converted into the data type "' ..
855               def.data_type .. '"!')
856         else
857           return converted
858         end
859       end
860     end,
861 
862     exclusive_group = function(value, key, def)
863       if value == nil then
864         return
865       end
866       if def.exclusive_group ~= nil then
867         if exclusive_groups[def.exclusive_group] ~= nil then
868           throw_error('The key "' .. key ..
869                         '" belongs to a mutually exclusive group "' ..
870                         def.exclusive_group .. '" and the key "' ..
871                         exclusive_groups[def.exclusive_group] ..
872                         '" is already present!')
873         else
874           exclusive_groups[def.exclusive_group] = key
875         end
876       end
877     end,
878 
879     l3_tl_set = function(value, key, def)
880       if value == nil then
881         return
882       end
883       if def.l3_tl_set ~= nil then
884         tex.print(l3_code_cctab,
885           '\\tl_set:Nn \\g_' .. def.l3_tl_set .. '_tl')
886         tex.print('{' .. value .. '}')
887       end
888     end,
889 
890     macro = function(value, key, def)
891       if value == nil then
892         return
893       end
894       if def.macro ~= nil then
895         token.set_macro(def.macro, value, 'global')
896       end
897     end,
898 
899     match = function(value, key, def)
900       if value == nil then
901         return
902       end
903       if def.match ~= nil then
904         if type(def.match) ~= 'string' then
905           throw_error('def.match has to be a string')
906         end
907         local match = string.match(value, def.match)
908         if match == nil then
```

```
909             throw_error(
910               'The value "' .. value .. '" of the key "' .. key ..
911                 '" does not match "' .. def.match .. '"!')
912           else
913             return match
914           end
915         end
916       end,
917
918       opposite_keys = function(value, key, def)
919         if def.opposite_keys ~= nil then
920           local true_value = def.opposite_keys[true]
921           local false_value = def.opposite_keys[false]
922           if true_value == nil or false_value == nil then
923             throw_error(
924               'Usage opposite_keys = { [true] = "...", [false] = "..." }')
925           end
926           if utils.remove_from_array(input, true_value) ~= nil then
927             return true
928           elseif utils.remove_from_array(input, false_value) ~= nil then
929             return false
930           end
931         end
932       end,
933
934       process = function(value, key, def)
935         if value == nil then
936           return
937         end
938         if def.process ~= nil and type(def.process) == 'function' then
939           return def.process(value, input_root, output, unknown)
940         end
941       end,
942
943       pick = function(value, key, def)
944         if def.pick then
945           local pick_types
946
947           -- Allow old deprecated attribut pick = true
948           if def.pick == true then
949             pick_types = { 'any' }
950           elseif type(def.pick) == 'table' then
951             pick_types = def.pick
952           else
953             pick_types = { def.pick }
954           end
955
956           -- Check if the pick attribute is valid
957           for _, pick_type in ipairs(pick_types) do
958             if type(pick_type) == 'string' and is[pick_type] == nil then
959               throw_error(
960                 'Wrong data type in the "pick" attribute: ' ..
961                   tostring(pick_type) ..
962                   '. Allowed are: any, boolean, dimension, integer, number, string.')
963             end
964           end
965
966           -- The key has already a value. We leave the function at this
967           -- point to be able to check the pick attribute for errors
968           -- beforehand.
969           if value ~= nil then
970             return value
```

```
971          end
972
973          for _, pick_type in ipairs(pick_types) do
974            for i, v in pairs(input) do
975              -- We can not use ipairs here. `ipairs(t)` iterates up to the
976              -- first absent index. Values are deleted from the `input`
977              -- table.
978              if type(i) == 'number' then
979                local picked_value = nil
980                if is[pick_type](v) then
981                  picked_value = v
982                end
983
984                if picked_value ~= nil then
985                  input[i] = nil
986                  return picked_value
987                end
988              end
989            end
990          end
991        end
992      end,
993
994      required = function(value, key, def)
995        if def.required ~= nil and def.required and value == nil then
996          throw_error(string.format('Missing required key "%s"!', key))
997        end
998      end,
999
1000     sub_keys = function(value, key, def)
1001       if def.sub_keys ~= nil then
1002         local v
1003         -- To get keys defined with always_present
1004         if value == nil then
1005           v = {}
1006         elseif type(value) == 'string' then
1007           v = { value }
1008         elseif type(value) == 'table' then
1009           v = value
1010         end
1011         v = apply_definitions(def.sub_keys, opts, v, output[key],
1012           unknown, add_to_key_path(key_path, key), input_root)
1013         if utils.get_table_size(v) > 0 then
1014           return v
1015         end
1016       end
1017     end,
1018   }
1019
1020   --- standalone values are removed.
1021   -- For some callbacks and the third return value of parse, we
1022   -- need an unchanged raw result from the parse function.
1023   input = clone_table(input)
1024   if output == nil then
1025     output = {}
1026   end
1027   if unknown == nil then
1028     unknown = {}
1029   end
1030   if key_path == nil then
1031     key_path = {}
1032   end
```

```
1033
1034   for index, def in pairs(defs) do
1035     -- Find key and def
1036     local key
1037     -- `{ key1 = { }, key2 = { } }`
1038     if type(def) == 'table' and def.name == nil and type(index) ==
1039       'string' then
1040       key = index
1041       -- `{ { name = 'key1' }, { name = 'key2' } }`
1042     elseif type(def) == 'table' and def.name ~= nil then
1043       key = def.name
1044       -- Definitions as strings in an array: `{ 'key1', 'key2' }`
1045     elseif type(index) == 'number' and type(def) == 'string' then
1046       key = def
1047       def = { default = get_default_value({}) }
1048     end
1049
1050     if type(def) ~= 'table' then
1051       throw_error('Key definition must be a table!')
1052     end
1053
1054     for attr, _ in pairs(def) do
1055       if namespace.attrs[attr] == nil then
1056         throw_error('Unknown definition attribute: ' .. tostring(attr))
1057       end
1058     end
1059
1060     if key == nil then
1061       throw_error('Key name couldn't be detected!')
1062     end
1063
1064     local value = find_value(key, def)
1065
1066     for _, def_opt in ipairs({
1067       'alias',
1068       'opposite_keys',
1069       'pick',
1070       'always_present',
1071       'required',
1072       'data_type',
1073       'choices',
1074       'match',
1075       'exclusive_group',
1076       'macro',
1077       'l3_tl_set',
1078       'process',
1079       'sub_keys',
1080     }) do
1081       if def[def_opt] ~= nil then
1082         local tmp_value = apply[def_opt](value, key, def)
1083         if tmp_value ~= nil then
1084           value = tmp_value
1085         end
1086       end
1087     end
1088
1089     output[key] = value
1090   end
1091
1092   if utils.get_table_size(input) > 0 then
1093     -- Move to the current unknown table.
1094     local current_unknown = unknown
```

```
1095      for _, key in ipairs(key_path) do
1096        if current_unknown[key] == nil then
1097          current_unknown[key] = {}
1098        end
1099        current_unknown = current_unknown[key]
1100      end
1101
1102      -- Copy all unknown key-value-pairs to the current unknown table.
1103      for key, value in pairs(input) do
1104        current_unknown[key] = value
1105      end
1106    end
1107
1108    return output, unknown
1109  end
1110
1111  --- Parse a LaTeX/TeX style key-value string into a Lua table.
1112  ---
1113  ---@param kv_string string # A string in the TeX/LaTeX style key-value format as
      ↪  described above.
1114  ---@param opts? table # A table containing the settings:
1115  ---    `convert_dimensions`, `unpack`, `naked_as_value`, `converter`,
1116  ---    `debug`, `preprocess`, `postprocess`.
1117  --
1118  ---@return table result # The final result of all individual parsing and
      ↪  normalization steps.
1119  ---@return table unknown # A table with unknown, undefinied key-value pairs.
1120  ---@return table raw # The unprocessed, raw result of the LPeg parser.
1121  local function parse(kv_string, opts)
1122    if kv_string == nil then
1123      return {}, {}, {}
1124    end
1125
1126    opts = normalize_opts(opts)
1127
1128    if type(opts.hooks.kv_string) == 'function' then
1129      kv_string = opts.hooks.kv_string(kv_string)
1130    end
1131
1132    local result = generate_parser('list', opts):match(kv_string)
1133    local raw = clone_table(result)
1134
1135    local function apply_hook(name)
1136      if type(opts.hooks[name]) == 'function' then
1137        if name:match('^keys') then
1138          result = visit_tree(result, opts.hooks[name])
1139        else
1140          opts.hooks[name](result)
1141        end
1142
1143        if opts.debug then
1144          print('After the execution of the hook: ' .. name)
1145          debug(result)
1146        end
1147      end
1148    end
1149
1150    local function apply_hooks(at)
1151      if at ~= nil then
1152        at = '_' .. at
1153      else
1154        at = ''
```

```
1155      end
1156      apply_hook('keys' .. at)
1157      apply_hook('result' .. at)
1158    end
1159
1160    apply_hooks('before_opts')
1161
1162    --- Normalize the result table of the LPeg parser. This normalization
1163    --  tasks are performed on the raw input table coming directly from
1164    --  the PEG parser:
1165    --
1166    ---@param result table # The raw input table coming directly from the PEG parser
1167    ---@param opts table # Some options.
1168    local function apply_opts(result, opts)
1169      local callbacks = {
1170        unpack = function(key, value)
1171          if type(value) == 'table' and utils.get_array_size(value) == 1 and
1172            utils.get_table_size(value) == 1 and type(value[1]) ~= 'table' then
1173            return key, value[1]
1174          end
1175          return key, value
1176        end,
1177
1178        process_naked = function(key, value)
1179          if type(key) == 'number' and type(value) == 'string' then
1180            return value, opts.default
1181          end
1182          return key, value
1183        end,
1184
1185        format_key = function(key, value)
1186          if type(key) == 'string' then
1187            for _, style in ipairs(opts.format_keys) do
1188              if style == 'lower' then
1189                key = key:lower()
1190              elseif style == 'snake' then
1191                key = key:gsub('[^%w]+', '_')
1192              elseif style == 'upper' then
1193                key = key:upper()
1194              else
1195                throw_error('Unknown style to format keys: ' ..
1196                            tostring(style) ..
1197                            ' Allowed styles are: lower, snake, upper')
1198              end
1199            end
1200          end
1201          return key, value
1202        end,
1203
1204        apply_invert_flag = function(key, value)
1205          if type(key) == 'string' and key:find(opts.invert_flag) then
1206            return key:gsub(opts.invert_flag, ''), not value
1207          end
1208          return key, value
1209        end,
1210      }
1211
1212      if opts.unpack then
1213        result = visit_tree(result, callbacks.unpack)
1214      end
1215
1216      if not opts.naked_as_value and opts.defs == false then
```

```
1217        result = visit_tree(result, callbacks.process_naked)
1218      end
1219
1220      if opts.format_keys then
1221        if type(opts.format_keys) ~= 'table' then
1222          throw_error(
1223            'The option "format_keys" has to be a table not ' ..
1224              type(opts.format_keys))
1225        end
1226        result = visit_tree(result, callbacks.format_key)
1227      end
1228
1229      if opts.invert_flag then
1230        result = visit_tree(result, callbacks.apply_invert_flag)
1231      end
1232
1233      return result
1234    end
1235    result = apply_opts(result, opts)
1236
1237    -- All unknown keys are stored in this table
1238    local unknown = nil
1239    if type(opts.defs) == 'table' then
1240      apply_hooks('before_defs')
1241      result, unknown = apply_definitions(opts.defs, opts, result, {}, {},
1242        {}, clone_table(result))
1243    end
1244
1245    apply_hooks()
1246
1247    if opts.defaults ~= nil and type(opts.defaults) == 'table' then
1248      merge_tables(result, opts.defaults)
1249    end
1250
1251    if opts.debug then
1252      debug(result)
1253    end
1254
1255    -- no_error
1256    if not opts.no_error and type(unknown) == 'table' and
1257      utils.get_table_size(unknown) > 0 then
1258      throw_error('Unknown keys: ' .. render(unknown))
1259    end
1260    return result, unknown, raw
1261  end
1262
1263  --- Store results
1264  -- @section
1265
1266  --- A table to store parsed key-value results.
1267  local result_store = {}
1268
1269  --- Exports
1270  -- @section
1271
1272  local export = {
1273    version = { 0, 11, 0 },
1274
1275    ---Get a new instance of the luakeys module.
1276    ---
1277    ---@return table # A new instance of the luakeys module.
1278    get_private_instance = function()
```

```
1279    package.loaded.luakeys = nil
1280    local luakeys = require('luakeys')
1281    package.loaded.luakeys = nil
1282    return luakeys
1283  end,
1284
1285  namespace = namespace,
1286
1287  ---This function is used in the documentation.
1288  ---
1289  ---@param from string # A key in the namespace table, either `opts`, `hook` or
    ↪  `attrs`.
1290  print_names = function(from)
1291    local names = {}
1292    for name in pairs(namespace[from]) do
1293      table.insert(names, name)
1294    end
1295    table.sort(names)
1296    tex.print(table.concat(names, ', '))
1297  end,
1298
1299  print_default = function(from, name)
1300    tex.print(tostring(namespace[from][name]))
1301  end,
1302
1303  --- @see default_options
1304  opts = default_options,
1305
1306  --- @see stringify
1307  stringify = stringify,
1308
1309  --- @see parse
1310  parse = parse,
1311
1312  define = function(defs, opts)
1313    return function(kv_string, inner_opts)
1314      local options
1315
1316      if inner_opts ~= nil and opts ~= nil then
1317        options = merge_tables(opts, inner_opts)
1318      elseif inner_opts ~= nil then
1319        options = inner_opts
1320      elseif opts ~= nil then
1321        options = opts
1322      end
1323
1324      if options == nil then
1325        options = {}
1326      end
1327
1328      options.defs = defs
1329
1330      return parse(kv_string, options)
1331    end
1332  end,
1333
1334  --- @see render
1335  render = render,
1336
1337  --- @see debug
1338  debug = debug,
1339
```

```
1340    --- The function `save(identifier, result): void` saves a result (a
1341    -- table from a previous run of `parse`) under an identifier.
1342    -- Therefore, it is not necessary to pollute the global namespace to
1343    -- store results for the later usage.
1344    --
1345    ---@param identifier string # The identifier under which the result is saved.
1346    --
1347    ---@param result table # A result to be stored and that was created by the
        ↪ key-value parser.
1348    save = function(identifier, result)
1349      result_store[identifier] = result
1350    end,
1351
1352    --- The function `get(identifier): table` retrieves a saved result
1353    -- from the result store.
1354    --
1355    ---@param identifier string # The identifier under which the result was saved.
1356    ---
1357    ---@return table
1358    get = function(identifier)
1359      -- if result_store[identifier] == nil then
1360      --   throw_error('No stored result was found for the identifier \'' ..
          ↪ identifier .. '\'')
1361      -- end
1362      return result_store[identifier]
1363    end,
1364
1365    is = is,
1366
1367    utils = utils,
1368  }
1369
1370  return export
```

8.2 luakeys.tex

```
1  %% luakeys.tex
2  %% Copyright 2021-2022 Josef Friedrich
3  %
4  % This work may be distributed and/or modified under the
5  % conditions of the LaTeX Project Public License, either version 1.3c
6  % of this license or (at your option) any later version.
7  % The latest version of this license is in
8  %   http://www.latex-project.org/lppl.txt
9  % and version 1.3c or later is part of all distributions of LaTeX
10 % version 2008/05/04 or later.
11 %
12 % This work has the LPPL maintenance status `maintained'.
13 %
14 % The Current Maintainer of this work is Josef Friedrich.
15 %
16 % This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
17 % luakeys-debug.sty and luakeys-debug.tex.
18
19 \directlua{luakeys = require('luakeys')}
```

8.3 luakeys.sty

```
1  %% luakeys.sty
2  %% Copyright 2021-2022 Josef Friedrich
3  %
4  % This work may be distributed and/or modified under the
5  % conditions of the LaTeX Project Public License, either version 1.3c
6  % of this license or (at your option) any later version.
7  % The latest version of this license is in
8  %   http://www.latex-project.org/lppl.txt
9  % and version 1.3c or later is part of all distributions of LaTeX
10 % version 2008/05/04 or later.
11 %
12 % This work has the LPPL maintenance status `maintained'.
13 %
14 % The Current Maintainer of this work is Josef Friedrich.
15 %
16 % This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
17 % luakeys-debug.sty and luakeys-debug.tex.
18
19 \NeedsTeXFormat{LaTeX2e}
20 \ProvidesPackage{luakeys}[2022/12/23 0.11.0 Parsing key-value options using Lua.]
21 \directlua{luakeys = require('luakeys')}
```

8.4 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')
}

\def\luakeysdebug%
{%
  \directlua%
  {
    local oarg = luakeys.utils.scan_oarg()
    local marg = token.scan_argument(false)
    local opts
    if oarg then
      opts = luakeys.parse(oarg, { format_keys = { 'snake', 'lower' } })
    end
    local result = luakeys.parse(marg, opts)
    luakeys.debug(result)
    tex.print(
      '{' ..
        '\string\\tt' ..
        '\string\\parindent=0pt' ..
        luakeys.stringify(result, true) ..
      '}'
    )
  }%
}
```

8.5 luakeys-debug.sty

```
1  %% luakeys-debug.sty
2  %% Copyright 2021-2022 Josef Friedrich
3  %
4  % This work may be distributed and/or modified under the
5  % conditions of the LaTeX Project Public License, either version 1.3c
6  % of this license or (at your option) any later version.
7  % The latest version of this license is in
8  %   http://www.latex-project.org/lppl.txt
9  % and version 1.3c or later is part of all distributions of LaTeX
10 % version 2008/05/04 or later.
11 %
12 % This work has the LPPL maintenance status `maintained'.
13 %
14 % The Current Maintainer of this work is Josef Friedrich.
15 %
16 % This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
17 % luakeys-debug.sty and luakeys-debug.tex.
18
19 \NeedsTeXFormat{LaTeX2e}
20 \ProvidesPackage{luakeys-debug}[2022/12/23 0.11.0 Debug package for luakeys.]
21
22 \input luakeys-debug.tex
```

Change History

0.1.0

General: Inital release 62

0.10.0

General: * Add support for an invert flat that flips the value of naked keys. * Add new options to specify which strings are recognized as Boolean values. 62

0.11.0

General: * Add a new function called "get_private_instance()" to load a private version of the luakeys module. 62

0.2.0

General: * Allow all recognized data types as keys. * Allow TeX macros in the values. * New public Lua functions: save(identifier, result), get(identifier). 62

0.3.0

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". 62

0.4.0

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. 62

0.5.0

General: * Add possibility to change options globally. * New option: standalone_as_true. * Add a recursive converter callback / hook to process the parse tree. * New option: case_insensitive_keys. 62

0.7.0

General: * The project now uses semantic versioning. * New definition attribute "pick" to pick standalone values and assign them to a key. * New function "utils.scan_oarg()" to search for an optional argument, that means scan for tokens that are enclosed in square brackets. * Extend and improve the documentation. . . 62

0.8.0

General: * Add 6 new options to change the delimiters: "assignment_operator", "group_begin", "group_end", "list_separator", "quotation_begin", "quotation_end". * Extend the documentation about the option "format_keys". 62

0.9.0

General: * The definition attibute "pick" accepts a new data type: "any". * The attribute value "true" for the attribute "pick" is deprecated. * The attribute "pick" accepts now multiple data types specified in a table. * Add a new function called "any(value)" in the "is" table that accepts any data type. . . 62