

The luakeys package

Josef Friedrich

josef@friedrich.rocks

github.com/Josef-Friedrich/luakeys

0.9.0 from 2022/11/21

```
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 Lua ^A TeX wrapper luakeys.sty	5
2.3	Using the plain LuaTeX wrapper luakeys.tex	5
3	Lua interface / API	5
3.1	Lua identifier names	6
3.2	Function “ <code>parse(kv_string, opts): result, unknown, raw</code> ”	6
3.3	Options to configure the parse function	7
3.3.1	Option “ <code>assignment_operator</code> ”	8
3.3.2	Option “ <code>convert_dimensions</code> ”	9
3.3.3	Option “ <code>debug</code> ”	9
3.3.4	Option “ <code>default</code> ”	10
3.3.5	Option “ <code>defaults</code> ”	10
3.3.6	Option “ <code>defs</code> ”	10
3.3.7	Option “ <code>format_keys</code> ”	11
3.3.8	Option “ <code>group_begin</code> ”	11
3.3.9	Option “ <code>group_end</code> ”	11
3.3.10	Option “ <code>hooks</code> ”	11
3.3.11	Option “ <code>list_separator</code> ”	13
3.3.12	Option “ <code>naked_as_value</code> ”	13
3.3.13	Option “ <code>no_error</code> ”	13
3.3.14	Option “ <code>quotation_begin</code> ”	13
3.3.15	Option “ <code>quotation_end</code> ”	13
3.3.16	Option “ <code>unpack</code> ”	13
3.4	Function “ <code>define(defs, opts): parse</code> ”	14
3.5	Attributes to define a key-value pair	14
3.5.1	Attribute “ <code>alias</code> ”	15
3.5.2	Attribute “ <code>always_present</code> ”	16
3.5.3	Attribute “ <code>choices</code> ”	16
3.5.4	Attribute “ <code>data_type</code> ”	16
3.5.5	Attribute “ <code>default</code> ”	17
3.5.6	Attribute “ <code>exclusive_group</code> ”	17
3.5.7	Attribute “ <code>opposite_keys</code> ”	17
3.5.8	Attribute “ <code>macro</code> ”	18
3.5.9	Attribute “ <code>match</code> ”	18
3.5.10	Attribute “ <code>name</code> ”	18
3.5.11	Attribute “ <code>pick</code> ”	19
3.5.12	Attribute “ <code>process</code> ”	20
3.5.13	Attribute “ <code>required</code> ”	21
3.5.14	Attribute “ <code>sub_keys</code> ”	21
3.6	Function “ <code>render(result): string</code> ”	22
3.7	Function “ <code>debug(result): void</code> ”	22
3.8	Function “ <code>save(identifier, result): void</code> ”	22

3.9	Function “get(identifier): result”	22
3.10	Table “is”	23
3.10.1	Function “is.boolean(value): boolean”	23
3.10.2	Function “is.dimension(value): boolean”	23
3.10.3	Function “is.integer(value): boolean”	23
3.10.4	Function “is.number(value): boolean”	23
3.10.5	Function “is.string(value): boolean”	24
3.10.6	Function “is.any(value): boolean”	24
3.11	Table “utils”	24
3.11.1	Function “utils.scan_oarg(initial_delimiter?, end_delimiter?): string”	24
3.12	Table “version”	25
4	Syntax of the recognized key-value format	25
4.1	An attempt to put the syntax into words	25
4.2	An (incomplete) attempt to put the syntax into the Extended Backus-Naur Form	25
4.3	Recognized data types	26
4.3.1	boolean	26
4.3.2	number	26
4.3.3	dimension	28
4.3.4	string	28
4.3.5	Naked keys	29
5	Examples	30
5.1	Extend and modify keys of existing macros	30
5.2	Process document class options	31
6	Debug packages	32
6.1	For plain T _E X: luakeys-debug.tex	32
6.2	For L ^A T _E X: luakeys-debug.sty	32
7	Implementation	33
7.1	luakeys.lua	33
7.2	luakeys.tex	56
7.3	luakeys.sty	57
7.4	luakeys-debug.tex	58
7.5	luakeys-debug.sty	59

1 Introduction

luakeys is a Lua module / LuaTeX package 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 Lua^AT_EX wrapper `luakeys.sty`

For example, the MiK_TE_X package manager downloads packages only when needed. It has been reported that this automatic download only works with this wrapper files. Probably MiK_TE_X is searching for an occurrence of the L^AT_EX macro “`\usepackage {luakeys}`”. The supplied Lua^AT_EX 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 Lua_TE_X 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 Lua^AT_EX 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 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
```

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 identifier names

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

Abbreviation	spelled out	Example
<code>kv_string</code>	Key-value string	<code>'key=value'</code>
<code>opts</code>	Options (for the parse function)	<code>{ no_error = false }</code>
<code>defs</code>	Definitions	
<code>def</code>	Definition	
<code>attr</code>	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, undefined key-value pairs.
`raw` The raw result of the Lpeg grammar parser.

3.2 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
```

3.3 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`, `format_keys`, `group_begin`, `group_end`, `hooks`, `list_separator`, `naked_as_value`, `no_error`, `quotation_begin`, `quotation_end`, `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' } },

  -- 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,
},

-- 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 = '"',

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

```

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 }
```

3.3.1 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	3.3.1
group_begin	3.3.8
group_end	3.3.9
list_separator	3.3.11
quotation_begin	3.3.14
quotation_end	3.3.15

3.3.2 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`.

3.3.3 Option “debug”

If the option `debug` is set to `true`, 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`.

3.3.4 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`.

3.3.5 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.

```

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`.

3.3.6 Option “defs”

For more informations on how keys are defined, see section 3.4. If you use the `defs` option, you don’t need to call the `define` function. Instead of ...

```

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

```

we can write ...

```

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`.

3.3.7 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`.

3.3.8 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 [3.3.1](#).

3.3.9 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 [3.3.1](#).

3.3.10 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 defined using defs, see [3.5.12](#))
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.

```

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.

```

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.

3.3.11 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 [3.3.1](#).

3.3.12 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`.

3.3.13 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`:

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

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

3.3.14 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 [3.3.1](#).

3.3.15 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 [3.3.1](#).

3.3.16 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.

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

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

The default value of the option “unpack” is: true.

3.4 Function “define(defs, opts): parse”

The `define` function returns a `parse` function (see 3.2). 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.

```
-- 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' } } }
```

3.5 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
    -- defined, 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$',

    -- 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 = { } },
  }
}

```

3.5.1 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' }
```

3.5.2 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 defined, `true` is taken as the value.

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

3.5.3 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:

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

3.5.4 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.

```
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
```

```
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')
```

3.5.5 Attribute “default”

Use the `default` attribute to provide a default value for each naked key individually. With the global `default` attribute (3.3.4) a default value can be specified for all naked keys.

```
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 }
```

3.5.6 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.

```
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.

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

3.5.7 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`.

```
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`.

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

3.5.8 Attribute “macro”

The attribute macro stores the value in a \TeX macro.

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

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

3.5.9 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>).

```
local parse = luakeys.define({
  birthday = { match = '~%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$"!'
```

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' } })
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 rubbish') -- { year = '1978' }
```

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

3.5.10 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 }

```

3.5.11 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: `dimension`, `number`, `boolean`, `any`, `string`, `integer`. 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' }

```

3.5.12 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 associated 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:

```
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:

```
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:

```

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

```

```

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

```

3.5.13 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.

```

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.

```

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

```

A recursive example:

```

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

```

The important2 key on level 2 is missing.

```

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

```

The important1 key at the lowest key level is missing.

```

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

```

3.5.14 Attribute “sub_keys”

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

```

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' } }
```

3.6 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.

```
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.

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

3.7 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',
    },
  },
}
```

3.8 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.

3.9 Function “get(identifier): result”

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

3.10 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.

3.10.1 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)
```

3.10.2 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)
```

3.10.3 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)
```

3.10.4 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)

```

3.10.5 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)

```

3.10.6 Function “is.any(value): boolean”

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

3.11 Table “utils”

3.11.1 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 Lua \TeX . 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
```

3.12 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 defined 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

$\langle list \rangle ::= \{ \langle list-item \rangle \}$

$\langle list-container \rangle ::= \{ \langle list \rangle \}$

$\langle list-item \rangle ::= (\langle list-container \rangle | \langle key-value-pair \rangle | \langle value \rangle) [', ']$

$\langle key-value-pair \rangle ::= \langle value \rangle '=' (\langle list-container \rangle | \langle value \rangle)$

$\langle value \rangle ::= \langle boolean \rangle$
| $\langle dimension \rangle$
| $\langle number \rangle$
| $\langle string-quoted \rangle$
| $\langle string-unquoted \rangle$

$\langle dimension \rangle ::= \langle number \rangle \langle unit \rangle$

$\langle number \rangle ::= \langle sign \rangle (\langle integer \rangle [\langle fractional \rangle] | \langle fractional \rangle)$

$\langle fractional \rangle ::= '.' \langle integer \rangle$

$\langle sign \rangle ::= '-' | '+'$

$\langle integer \rangle ::= \langle digit \rangle \{ \langle digit \rangle \}$

$\langle digit \rangle ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'$

$\langle unit \rangle ::= 'bp' | 'BP'$
| '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'

$\langle boolean \rangle ::= \langle boolean-true \rangle | \langle boolean-false \rangle$

$\langle boolean-true \rangle ::= 'true' | 'TRUE' | 'True'$

$\langle boolean-false \rangle ::= 'false' | 'FALSE' | 'False'$

... 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 T_EX world. According to the LuaT_EX source code (<source/texk/web2c/luatexdir/luatexlib.c>) and the dimension module of the lualibs library (<lualibs-util-dim.lua>), all units should be recognized.

	Description	<code>\luakeysdebug[convert_dimensions]</code>	
bp	big point	<code>bp = 1bp,</code>	<code>['bp'] = 65781,</code>
cc	cicero	<code>cc = 1cc,</code>	<code>['cc'] = 841489,</code>
cm	centimeter	<code>cm = 1cm,</code>	<code>['cm'] = 1864679,</code>
dd	didot	<code>dd = 1dd,</code>	<code>['dd'] = 70124,</code>
em	horizontal measure of M	<code>em = 1em,</code>	<code>['em'] = 655360,</code>
ex	vertical measure of x	<code>ex = 1ex,</code>	<code>['ex'] = 282460,</code>
in	inch	<code>in = 1in,</code>	<code>['in'] = 4736286,</code>
mm	millimeter	<code>mm = 1mm,</code>	<code>['mm'] = 186467,</code>
mu	math unit	<code>mu = 1mu,</code>	<code>['mu'] = 65536,</code>
nc	new cicero	<code>nc = 1nc,</code>	<code>['nc'] = 839105,</code>
nd	new didot	<code>nd = 1nd,</code>	<code>['nd'] = 69925,</code>
pc	pica	<code>pc = 1pc,</code>	<code>['pc'] = 786432,</code>
pt	point	<code>pt = 1pt,</code>	<code>['pt'] = 65536,</code>
px	x height current font	<code>px = 1px,</code>	<code>['px'] = 65781,</code>
sp	scaledpoint	<code>sp = 1sp,</code>	<code>['sp'] = 1,</code>

The next example illustrates the different notations of the dimensions.

<code>\luakeysdebug[convert_dimensions=true]</code>	<code>{</code>
<code>upper = 1CM,</code>	<code>['upper'] = 1864679,</code>
<code>lower = 1cm,</code>	<code>['lower'] = 1864679,</code>
<code>space = 1 cm,</code>	<code>['space'] = 1864679,</code>
<code>plus = + 1cm,</code>	<code>['plus'] = 1864679,</code>
<code>minus = -1cm,</code>	<code>['minus'] = -1864679,</code>
<code>nodim = 1 c m,</code>	<code>['nodim'] = '1 c m', -- string</code>
<code>}</code>	<code>}</code>

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

\myincludegraphics{test.png}

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

\myincludegraphics[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:

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

6.1 For plain \TeX : `luakeys-debug.tex`

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

```
\input luakeys-debug.tex
\uakeysdebug{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}
\uakeysdebug[
  unpack=false,
  convert dimensions=false
]{one,two,three}
\end{document}
```

7 Implementation

7.1 luakeys.lua

```
1  -- luakeys.lua
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 --- A key-value parser written with Lpeg.
19 --
20 -- @module luakeys
21 local lpeg = require('lpeg')
22
23 if not tex then
24   tex = {
25     -- Dummy function for the tests.
26     sp = function(input)
27       return 1234567
28     end,
29   }
30 end
31
32 if not token then
33   token = {
34     set_macro = function(csname, content, global)
35       end,
36   }
37 end
38
39 --- Merge two tables in the first specified table.
40 --- The `merge_tables` function copies all keys from the `source` table
41 --- to a target table. It returns the modified target table.
42 ---
43 ---@see https://stackoverflow.com/a/1283608/10193818
44 ---
45 ---@param target table
46 ---@param source table
47 ---
48 ---@return table target The modified target table.
49 local function merge_tables(target, source)
50   for key, value in pairs(source) do
51     if type(value) == 'table' then
52       if type(target[key] or false) == 'table' then
53         merge_tables(target[key] or {}, source[key] or {})
54       elseif target[key] == nil then
55         target[key] = value
56       end
57     elseif target[key] == nil then
58       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
92   ↪ or the table is empty.
93   get_array_size = function(value)
94     local count = 0
95     if type(value) == 'table' then
96       for _ in ipairs(value) do
97         count = count + 1
98       end
99     end
100    return count
101  end,
102
103  ---Get the size of a table `{ one = 'one', 'two', 'three' }` = 3.
104  ---
105  ---@param value table/any # A table or any input.
106  ---
107  ---@return number # The size of the array like table. 0 if the input is no table
108  ↪ or the table is empty.
109  get_table_size = function(value)
110    local count = 0
111    if type(value) == 'table' then
112      for _ in pairs(value) do
113        count = count + 1
114      end
115    end
116    return count
117  end,
118  merge_tables = merge_tables,

```

```

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
133   ↳ optional argument (by default '[').
134 ---@param end_delimiter? string # The character that marks the end of an optional
135   ↳ argument (by default ']').
136 ---
137 ---@return string/nil # The string that was enclosed by the delimiters. The
138   ↳ delimiters themselves are not returned.
139 scan_oarg = function(initial_delimiter, end_delimiter)
140     if initial_delimiter == nil then
141         initial_delimiter = '['
142     end
143
144     if end_delimiter == nil then
145         end_delimiter = ']'
146     end
147
148     local function convert_token(t)
149         if t.index ~= nil then
150             return utf8.char(t.index)
151         else
152             return '\\\ ' .. t.csname
153         end
154     end
155
156     local function get_next_char()
157         local t = token.get_next()
158         return convert_token(t), t
159     end
160
161     local char, t = get_next_char()
162     if char == initial_delimiter then
163         local oarg = {}
164         char = get_next_char()
165         while char ~= end_delimiter do
166             table.insert(oarg, char)
167             char = get_next_char()
168         end
169         return table.concat(oarg, '')
170     else
171         token.put_next(t)
172     end
173 end,
174 }
175
176 local namespace = {
177     opts = {
178         assignment_operator = '=',
179         convert_dimensions = false,
180         debug = false,

```

```

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

```

```

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

```

```

301     end
302   end
303   return table.concat(output)
304 end
305 return render_inner(result)
306 end
307
308 --- The function `stringify(tbl, for_tex)` converts a Lua table into a
309 --- printable string. Stringify a table means to convert the table into
310 --- a string. This function is used to realize the `debug` function.
311 --- `stringify(tbl, true)` (`for_tex = true`) generates a string which
312 --- can be embeded into TeX documents. The macro `\luakeysdebug{}` uses
313 --- this option. `stringify(tbl, false)` or `stringify(tbl)` generate a
314 --- string suitable for the terminal.
315 ---
316 ---@see https://stackoverflow.com/a/54593224/10193818
317 ---
318 ---@param result table # A table to stringify.
319 ---@param for_tex? boolean # Stringify the table into a text string that can be
320 ↪ embeded inside a TeX document via tex.print(). Curly braces and whites spaces
321 ↪ are escaped.
322 ---
323 ---@return string
324
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)

```

```

361     if (type(value) == 'table') then
362         if (next(value)) then
363             add(depth, key .. ' = ' .. start_bracket)
364             add(0, stringify_inner(value, depth + 1))
365             add(depth, end_bracket .. ',');
366         else
367             add(depth,
368                 key .. ' = ' .. start_bracket .. end_bracket .. ',')
369         end
370     else
371         if (type(value) == 'string') then
372             value = string.format('\'%s\'', value)
373         else
374             value = tostring(value)
375         end
376
377         add(depth, key .. ' = ' .. value .. ',')
378     end
379 end
380 end
381
382 return table.concat(output, line_break)
383 end
384
385 return start_bracket .. line_break .. stringify_inner(result, 1) ..
386        line_break .. end_bracket
387 end
388
389 --- The function `debug(result)` pretty prints a Lua table to standard
390 --- output (stdout). It is a utility function that can be used to
391 --- debug and inspect the resulting Lua table of the function
392 --- `parse`. You have to compile your TeX document in a console to
393 --- see the terminal output.
394 ---
395 ---@param result table # A table to be printed to standard output for debugging
396 --- ↪ purposes.
397 local function debug(result)
398     print('\n' .. stringify(result, false))
399 end
400 --- Parser / Lpeg related
401 --- @section
402
403 --- Generate the PEG parser using Lpeg.
404 ---
405 --- Explanations of some LPEG notation forms:
406 ---
407 --- * `patt ^ 0` = `expression *`
408 --- * `patt ^ 1` = `expression +`
409 --- * `patt ^ -1` = `expression ?`
410 --- * `patt1 * patt2` = `expression1 expression2`: Sequence
411 --- * `patt1 + patt2` = `expression1 / expression2`: Ordered choice
412 ---
413 --- * [TUGboat article: Parsing complex data formats in LuaTeX with
414 --- ↪ LPEG](https://tug.or-g/TUGboat/tb40-2/tb125menke-Patterndf)
415 ---
416 ---@param initial_rule string # The name of the first rule of the grammar table
417 --- ↪ passed to the `lpeg.P(attern)` function (e. g. `list`, `number`).
418 ---@param opts? table # Whether the dimensions should be converted to scaled points
419 --- ↪ (by default `false`).
420 ---
421 ---@return userdata # The parser.

```

```

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

```

```

479     local index = #result + 1
480     return rawset(result, index, arg1)
481 else
482     return rawset(result, arg1, arg2)
483 end
484 end
485
486 -- LuaFormatter off
487 return Pattern({
488     [1] = initial_rule,
489
490     -- list_item*
491     list = CaptureFolding(
492         CaptureTable('') * Variable('list_item')^0,
493         add_to_table
494     ),
495
496     -- '{ list }'
497     list_container =
498         ws(opts.group_begin) * Variable('list') * ws(opts.group_end),
499
500     -- ( list_container / key_value_pair / value ) ', '?
501     list_item =
502         CaptureGroup(
503             Variable('list_container') +
504             Variable('key_value_pair') +
505             Variable('value')
506         ) * ws(opts.list_separator)^-1,
507
508     -- key '=' (list_container / value)
509     key_value_pair =
510         (Variable('key') * ws(opts.assignment_operator)) *
511         ↪ (Variable('list_container') + Variable('value')),
512
513     -- number / string_quoted / string_unquoted
514     key =
515         Variable('number') +
516         Variable('string_quoted') +
517         Variable('string_unquoted'),
518
519     -- boolean !value / dimension !value / number !value / string_quoted !value /
520     ↪ string_unquoted
521     -- !value -> Not-predicate -> * -Variable('value')
522     value =
523         Variable('boolean') * -Variable('value') +
524         Variable('dimension') * -Variable('value') +
525         Variable('number') * -Variable('value') +
526         Variable('string_quoted') * -Variable('value') +
527         Variable('string_unquoted'),
528
529     -- for is.boolean()
530     boolean_only = Variable('boolean') * -1,
531
532     -- boolean_true / boolean_false
533     boolean =
534         (
535             Variable('boolean_true') * CaptureConstant(true) +
536             Variable('boolean_false') * CaptureConstant(false)
537         ),
538
539     boolean_true =
540         Pattern('true') +

```

```

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

```

```

598
599     string_unquoted =
600         white_space^0 *
601         CaptureSimple(
602             Variable('word_unquoted')^1 *
603             (Set('\t')^1 * Variable('word_unquoted')^1)^0 *
604             white_space^0,
605
606         word_unquoted = (1 - white_space - Set(
607             opts.group_begin ..
608             opts.group_end ..
609             opts.assignment_operator ..
610             opts.list_separator))^1
611     })
612     -- LuaFormatter on
613 end
614
615 local function visit_tree(tree, callback_func)
616     if type(tree) ~= 'table' then
617         throw_error('Parameter "tree" has to be a table, got: ' ..
618             tostring(tree))
619     end
620     local function visit_tree_recursive(tree,
621         current,
622         result,
623         depth,
624         callback_func)
625         for key, value in pairs(current) do
626             if type(value) == 'table' then
627                 value = visit_tree_recursive(tree, value, {}, depth + 1,
628                     callback_func)
629             end
630
631             key, value = callback_func(key, value, depth, current, tree)
632
633             if key ~= nil and value ~= nil then
634                 result[key] = value
635             end
636         end
637         if next(result) ~= nil then
638             return result
639         end
640     end
641
642     local result = visit_tree_recursive(tree, tree, {}, 1, callback_func)
643
644     if result == nil then
645         return {}
646     end
647     return result
648 end
649
650 local is = {
651     boolean = function(value)
652         if value == nil then
653             return false
654         end
655         if type(value) == 'boolean' then
656             return true
657         end
658         local parser = generate_parser('boolean_only')
659         local result = parser:match(tostring(value))

```

```

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

```

```

722
723     for index, value in ipairs(key_path) do
724         new_key_path[index] = value
725     end
726
727     table.insert(new_key_path, key)
728     return new_key_path
729 end
730
731 local function get_default_value(def)
732     if def.default ~= nil then
733         return def.default
734     elseif opts ~= nil and opts.default ~= nil then
735         return opts.default
736     end
737     return true
738 end
739
740 local function find_value(search_key, def)
741     if input[search_key] ~= nil then
742         local value = input[search_key]
743         input[search_key] = nil
744         return value
745         -- naked keys: values with integer keys
746     elseif utils.remove_from_array(input, search_key) ~= nil then
747         return get_default_value(def)
748     end
749 end
750
751 local apply = {
752     alias = function(value, key, def)
753         if type(def.alias) == 'string' then
754             def.alias = { def.alias }
755         end
756         local alias_value
757         local used_alias_key
758         -- To get an error if the key and an alias is present
759         if value ~= nil then
760             alias_value = value
761             used_alias_key = key
762         end
763         for _, alias in ipairs(def.alias) do
764             local v = find_value(alias, def)
765             if v ~= nil then
766                 if alias_value ~= nil then
767                     throw_error(string.format(
768                         'Duplicate aliases "%s" and "%s" for key "%s"!',
769                         used_alias_key, alias, key))
770                 end
771                 used_alias_key = alias
772                 alias_value = v
773             end
774         end
775         if alias_value ~= nil then
776             return alias_value
777         end
778     end,
779
780     always_present = function(value, key, def)
781         if value == nil and def.always_present then
782             return get_default_value(def)
783         end

```

```

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

```

```

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

```

```

908
909 opposite_keys = function(value, key, def)
910   if def.opposite_keys ~= nil then
911     local true_value = def.opposite_keys[true]
912     local false_value = def.opposite_keys[false]
913     if true_value == nil or false_value == nil then
914       throw_error(
915         'Usage opposite_keys = { [true] = "...", [false] = "..." }')
916     end
917     if utils.remove_from_array(input, true_value) ~= nil then
918       return true
919     elseif utils.remove_from_array(input, false_value) ~= nil then
920       return false
921     end
922   end
923 end,
924
925 process = function(value, key, def)
926   if value == nil then
927     return
928   end
929   if def.process ~= nil and type(def.process) == 'function' then
930     return def.process(value, input_root, output, unknown)
931   end
932 end,
933
934 pick = function(value, key, def)
935   if def.pick then
936     local pick_types
937
938     -- Allow old deprecated attribut pick = true
939     if def.pick == true then
940       pick_types = { 'any' }
941     elseif type(def.pick) == 'table' then
942       pick_types = def.pick
943     else
944       pick_types = { def.pick }
945     end
946
947     -- Check if the pick attribute is valid
948     for _, pick_type in ipairs(pick_types) do
949       if type(pick_type) == 'string' and is[pick_type] == nil then
950         throw_error(
951           'Wrong data type in the "pick" attribute: ' ..
952             tostring(pick_type) ..
953           '. Allowed are: any, boolean, dimension, integer, number, string.')
954       end
955     end
956
957     -- The key has already a value. We leave the function at this
958     -- point to be able to check the pick attribute for errors
959     -- beforehand.
960     if value ~= nil then
961       return value
962     end
963
964     for _, pick_type in ipairs(pick_types) do
965       for i, v in pairs(input) do
966         -- We can not use ipairs here. `ipairs(t)` iterates up to the
967         -- first absent index. Values are deleted from the `input`
968         -- table.
969         if type(i) == 'number' then

```

```

970         local picked_value = nil
971         if is[pick_type](v) then
972             picked_value = v
973         end
974
975         if picked_value ~= nil then
976             input[i] = nil
977             return picked_value
978         end
979     end
980 end
981 end
982 end
983 end,
984
985 required = function(value, key, def)
986     if def.required ~= nil and def.required and value == nil then
987         throw_error(string.format('Missing required key "%s"', key))
988     end
989 end,
990
991 sub_keys = function(value, key, def)
992     if def.sub_keys ~= nil then
993         local v
994         -- To get keys defined with always_present
995         if value == nil then
996             v = {}
997         elseif type(value) == 'string' then
998             v = { value }
999         elseif type(value) == 'table' then
1000             v = value
1001         end
1002         v = apply_definitions(def.sub_keys, opts, v, output[key],
1003             unknown, add_to_key_path(key_path, key), input_root)
1004         if utils.get_table_size(v) > 0 then
1005             return v
1006         end
1007     end
1008 end,
1009 }
1010
1011 --- standalone values are removed.
1012 -- For some callbacks and the third return value of parse, we
1013 -- need an unchanged raw result from the parse function.
1014 input = clone_table(input)
1015 if output == nil then
1016     output = {}
1017 end
1018 if unknown == nil then
1019     unknown = {}
1020 end
1021 if key_path == nil then
1022     key_path = {}
1023 end
1024
1025 for index, def in pairs(defs) do
1026     -- Find key and def
1027     local key
1028     -- `{ key1 = { }, key2 = { } }`
1029     if type(def) == 'table' and def.name == nil and type(index) ==
1030         'string' then
1031         key = index

```

```

1032     -- `{ { name = 'key1' }, { name = 'key2' } }`
1033 elseif type(def) == 'table' and def.name ~= nil then
1034     key = def.name
1035     -- Definitions as strings in an array: `{ 'key1', 'key2' }`
1036 elseif type(index) == 'number' and type(def) == 'string' then
1037     key = def
1038     def = { default = get_default_value({}) }
1039 end
1040
1041 if type(def) ~= 'table' then
1042     throw_error('Key definition must be a table!')
1043 end
1044
1045 for attr, _ in pairs(def) do
1046     if namespace.attrs[attr] == nil then
1047         throw_error('Unknown definition attribute: ' .. tostring(attr))
1048     end
1049 end
1050
1051 if key == nil then
1052     throw_error('Key name couldn't be detected!')
1053 end
1054
1055 local value = find_value(key, def)
1056
1057 for _, def_opt in ipairs({
1058     'alias',
1059     'opposite_keys',
1060     'pick',
1061     'always_present',
1062     'required',
1063     'data_type',
1064     'choices',
1065     'match',
1066     'exclusive_group',
1067     'macro',
1068     'l3_t1_set',
1069     'process',
1070     'sub_keys',
1071 }) do
1072     if def[def_opt] ~= nil then
1073         local tmp_value = apply[def_opt](value, key, def)
1074         if tmp_value ~= nil then
1075             value = tmp_value
1076         end
1077     end
1078 end
1079
1080 output[key] = value
1081 end
1082
1083 if utils.get_table_size(input) > 0 then
1084     -- Move to the current unknown table.
1085     local current_unknown = unknown
1086     for _, key in ipairs(key_path) do
1087         if current_unknown[key] == nil then
1088             current_unknown[key] = {}
1089         end
1090         current_unknown = current_unknown[key]
1091     end
1092
1093     -- Copy all unknown key-value-pairs to the current unknown table.

```

```

1094     for key, value in pairs(input) do
1095         current_unknown[key] = value
1096     end
1097 end
1098
1099 return output, unknown
1100 end
1101
1102 --- Parse a LaTeX/TeX style key-value string into a Lua table.
1103 ---
1104 ---@param kv_string string # A string in the TeX/LaTeX style key-value format as
1105 ↪ described above.
1106 ---@param opts? table # A table containing the settings:
1107 ---   `convert_dimensions`, `unpack`, `naked_as_value`, `converter`,
1108 ---   `debug`, `preprocess`, `postprocess`.
1109 ---
1110 ---@return table result # The final result of all individual parsing and
1111 ↪ normalization steps.
1112 ---@return table unknown # A table with unknown, undefined key-value pairs.
1113 ---@return table raw # The unprocessed, raw result of the LPeg parser.
1114 local function parse(kv_string, opts)
1115     if kv_string == nil then
1116         return {}, {}, {}
1117     end
1118     opts = normalize_opts(opts)
1119     if type(opts.hooks.kv_string) == 'function' then
1120         kv_string = opts.hooks.kv_string(kv_string)
1121     end
1122     local result = generate_parser('list', opts):match(kv_string)
1123     local raw = clone_table(result)
1124
1125     local function apply_hook(name)
1126         if type(opts.hooks[name]) == 'function' then
1127             if name:match('^keys') then
1128                 result = visit_tree(result, opts.hooks[name])
1129             else
1130                 opts.hooks[name](result)
1131             end
1132         end
1133
1134         if opts.debug then
1135             print('After the execution of the hook: ' .. name)
1136             debug(result)
1137         end
1138     end
1139 end
1140
1141 local function apply_hooks(at)
1142     if at ~= nil then
1143         at = '_' .. at
1144     else
1145         at = ''
1146     end
1147     apply_hook('keys' .. at)
1148     apply_hook('result' .. at)
1149 end
1150
1151 apply_hooks('before_opts')
1152
1153 --- Normalize the result table of the LPeg parser. This normalization

```

```

1154 -- tasks are performed on the raw input table coming directly from
1155 -- the PEG parser:
1156 --
1157 ---@param result table # The raw input table coming directly from the PEG parser
1158 ---@param opts table # Some options.
1159 local function apply_opts(result, opts)
1160     local callbacks = {
1161         unpack = function(key, value)
1162             if type(value) == 'table' and utils.get_array_size(value) == 1 and
1163                 utils.get_table_size(value) == 1 and type(value[1]) ~= 'table' then
1164                 return key, value[1]
1165             end
1166             return key, value
1167         end,
1168
1169         process_naked = function(key, value)
1170             if type(key) == 'number' and type(value) == 'string' then
1171                 return value, opts.default
1172             end
1173             return key, value
1174         end,
1175
1176         format_key = function(key, value)
1177             if type(key) == 'string' then
1178                 for _, style in ipairs(opts.format_keys) do
1179                     if style == 'lower' then
1180                         key = key:lower()
1181                     elseif style == 'snake' then
1182                         key = key:gsub('[^%w]+', '_')
1183                     elseif style == 'upper' then
1184                         key = key:upper()
1185                     else
1186                         throw_error('Unknown style to format keys: ' ..
1187                             tostring(style) ..
1188                             ' Allowed styles are: lower, snake, upper')
1189                     end
1190                 end
1191                 return key, value
1192             end,
1193         }
1194     }
1195
1196     if opts.unpack then
1197         result = visit_tree(result, callbacks.unpack)
1198     end
1199
1200     if not opts.naked_as_value and opts.defs == false then
1201         result = visit_tree(result, callbacks.process_naked)
1202     end
1203
1204     if opts.format_keys then
1205         if type(opts.format_keys) ~= 'table' then
1206             throw_error(
1207                 'The option "format_keys" has to be a table not ' ..
1208                 type(opts.format_keys))
1209         end
1210         result = visit_tree(result, callbacks.format_key)
1211     end
1212
1213     return result
1214 end
1215 result = apply_opts(result, opts)

```

```

1216
1217     -- All unknown keys are stored in this table
1218     local unknown = nil
1219     if type(opts.defs) == 'table' then
1220         apply_hooks('before_defs')
1221         result, unknown = apply_definitions(opts.defs, opts, result, {}, {},
1222             {}, clone_table(result))
1223     end
1224
1225     apply_hooks()
1226
1227     if opts.defaults ~= nil and type(opts.defaults) == 'table' then
1228         merge_tables(result, opts.defaults)
1229     end
1230
1231     if opts.debug then
1232         debug(result)
1233     end
1234
1235     -- no_error
1236     if not opts.no_error and type(unknown) == 'table' and
1237         utils.get_table_size(unknown) > 0 then
1238         throw_error('Unknown keys: ' .. render(unknown))
1239     end
1240     return result, unknown, raw
1241 end
1242
1243 --- Store results
1244 -- @section
1245
1246 --- A table to store parsed key-value results.
1247 local result_store = {}
1248
1249 --- Exports
1250 -- @section
1251
1252 local export = {
1253     version = { 0, 9, 0 },
1254
1255     namespace = namespace,
1256
1257     ---This function is used in the documentation.
1258     ---
1259     ---@param from string # A key in the namespace table, either `opts`, `hook` or
1260     ↔ `attrs`.
1261     print_names = function(from)
1262         local names = {}
1263         for name in pairs(namespace[from]) do
1264             table.insert(names, name)
1265         end
1266         table.sort(names)
1267         tex.print(table.concat(names, ', '))
1268     end,
1269
1270     print_default = function(from, name)
1271         tex.print(tostring(namespace[from][name]))
1272     end,
1273
1274     --- @see default_options
1275     opts = default_options,
1276
1277     --- @see stringify

```

```

1277     stringify = stringify,
1278
1279     --- @see parse
1280     parse = parse,
1281
1282     define = function(defs, opts)
1283         return function(kv_string, inner_opts)
1284             local options
1285
1286             if inner_opts ~= nil and opts ~= nil then
1287                 options = merge_tables(opts, inner_opts)
1288             elseif inner_opts ~= nil then
1289                 options = inner_opts
1290             elseif opts ~= nil then
1291                 options = opts
1292             end
1293
1294             if options == nil then
1295                 options = {}
1296             end
1297
1298             options.defs = defs
1299
1300             return parse(kv_string, options)
1301         end
1302     end,
1303
1304     --- @see render
1305     render = render,
1306
1307     --- @see debug
1308     debug = debug,
1309
1310     --- The function `save(identifier, result): void` saves a result (a
1311     -- table from a previous run of `parse`) under an identifier.
1312     -- Therefore, it is not necessary to pollute the global namespace to
1313     -- store results for the later usage.
1314     --
1315     ---@param identifier string # The identifier under which the result is saved.
1316     --
1317     ---@param result table # A result to be stored and that was created by the
1318     ↪ key-value parser.
1319     save = function(identifier, result)
1320         result_store[identifier] = result
1321     end,
1322
1323     --- The function `get(identifier): table` retrieves a saved result
1324     -- from the result store.
1325     --
1326     ---@param identifier string # The identifier under which the result was saved.
1327     ---
1328     ---@return table
1329     get = function(identifier)
1330         -- if result_store[identifier] == nil then
1331         --     throw_error('No stored result was found for the identifier \'' ..
1332         ↪ identifier .. '\')
1333         -- end
1334         return result_store[identifier]
1335     end,
1336
1337     is = is,

```

```
1337     utils = utils,  
1338 }  
1339  
1340 return export
```

7.2 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 LPPPL 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')}
```

7.3 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 LPL 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}[2022/11/21 0.9.0 Parsing key-value options using Lua.]
21 \directlua{luakeys = require('luakeys')}
```

7.4 luakeys-debug.tex

```
1 %% luakeys-debug.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 LPL 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
20 {
21   luakeys = require('luakeys')
22 }
23
24 \def\luakeysdebug%
25 {%
26   \directlua%
27   {
28     local oarg = luakeys.utils.scan_oarg()
29     local marg = token.scan_argument(false)
30     local opts
31     if oarg then
32       opts = luakeys.parse(oarg, { format_keys = { 'snake', 'lower' } })
33     end
34     local result = luakeys.parse(marg, opts)
35     luakeys.debug(result)
36     tex.print(
37       '{' ..
38         '\string\\tt' ..
39         '\string\\parindent=0pt' ..
40         luakeys.stringify(result, true) ..
41       '}'
42     )
43   }%
44 }
```

7.5 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 LPL 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}[2022/11/21 0.9.0 Debug package for luakeys.]
21
22 \input luakeys-debug.tex
```

Change History

0.1.0	General: Initial release	59	0.7.0	case_insensitive_keys.	59
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).	59		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. . .	59
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”.	59	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”.	59
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.	59	0.9.0	General: * The definition attribute “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. . .	59
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:				