The luakeys package

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

v0.13.0 from 2023/01/13

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

Result:

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

Contents

1 Introduction

`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 article is based on a question asked on tex.stackexchange.com by Will Robertson: A big list of every keyval package. CTAN also provides an overview page on the subject of Key-Val: packages with key-value argument systems.

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

2.2 Using the LuaLaTeX wrapper `luakeys.sty`

For example, the MiKTeX package manager downloads packages only when needed. It has been reported that this automatic download only works with this wrapper files. Probably MiKTeX is searching for an occurrence of the LaTeX macro "`\usepackage{luakeys}`". The `luakeys.sty` file loads the Lua module into the global variable `luakeys`.

```
\documentclass{article}
\usepackage{luakeys}
\begin{document}
  \directlua{
    local lk = luakeys.new()
    local keys = lk.parse('one,two,three', { naked_as_value = true })
    tex.print(keys[1])
    tex.print(keys[2])
    tex.print(keys[3])
  } % one two three
\end{document}
```

2.3 Using the plain LuaTeX wrapper `luakeys.tex`

The file `luakeys.tex` does the same as the LuaLaTeX wrapper and loads the Lua module `luakeys.lua` into the global variable `luakeys`.

```
\input luakeys.tex
\directlua{
  local lk = luakeys.new()
  local keys = lk.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

Luakeys exports only one function that must be called to access the public API. This export function returns a table containing the public functions and additional tables:

```
local luakeys = require('luakeys')()
local new = luakeys.new
local version = luakeys.version
local parse = luakeys.parse
local define = luakeys.define
local opts = luakeys.opts
local error_messages = luakeys.error_messages
local render = luakeys.render
```

```lua
local stringify = luakeys.stringify
local debug = luakeys.debug
local save = luakeys.save
local get = luakeys.get
local is = luakeys.is
local utils = luakeys.utils
```

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

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

These unabbreviated variable names are commonly used.

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

It is recommended to use luakeys together with the github.com/sumneko/lua-language-server when developing in a text editor. luakeys supports the annotation format offered by the server. You should then get warnings if you misuse luakeys' now rather large API.

3.1 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.

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

In plain TEX:

```latex
\input luakeys.tex
\def\mykeyvalcmd#1{
  \directlua{
    local lk = luakeys.new()
```

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

3.2 Options to configure the parse function

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

```
local opts = {
  -- Result table that is filled with each call of the parse function.
  accumulated_result = accumulated_result,

  -- Configure the delimiter that assigns a value to a key.
  assignment_operator = '=',

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

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

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

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

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

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

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

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

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

  -- Listed in the order of execution
  hooks = {
    kv_string = function(kv_string)
      return kv_string
    end,

    -- Visit all key-value pairs recursively.
```

```
    keys_before_opts = function(key, value, depth, current, result)
      return key, value
    end,

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

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

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

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

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

  invert_flag = '!',

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

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

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

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

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

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

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

3.3 Table “opts”

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

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

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

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

3.3.1 Option "accumulated_result"

Strictly speaking, this is not an option. The `accumulated_result` "option" can be used to specify a result table that is filled with each call of the `parse` function.

```
  local result = {}

  luakeys.parse('key1=one', { accumulated_result = result })
  assert.are.same({ key1 = 'one' }, result)

  luakeys.parse('key2=two', { accumulated_result = result })
  assert.are.same({ key1 = 'one', key2 = 'two' }, result)

  luakeys.parse('key1=1', { accumulated_result = result })
  assert.are.same({ key1 = 1, key2 = 'two' }, result)
```

3.3.2 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	3.3.2
group_begin	3.3.10
group_end	3.3.11
list_separator	3.3.14
quotation_begin	3.3.17
quotation_end	3.3.18

3.3.3 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`.

3.3.4 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{
  lk = luakeys.new()
  lk.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.

3.3.5 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.

3.3.6 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.

3.3.7 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.

3.3.8 Option “false_aliases”

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

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

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

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

See section 3.3.19 for the corresponding option.

3.3.9 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.

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

You can also combine several types of formatting.

```lua
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`.

3.3.10 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 3.3.2.

3.3.11 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 3.3.2.

3.3.12 Option "`invert_flag`"

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

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

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

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

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

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

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

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

3.3.13 Option "hooks"

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

1. `kv_string = function(kv_string): kv_string`
2. `keys_before_opts = function(key, value, depth, current, result): key, value`
3. `result_before_opts = function(result): void`
4. `keys_before_def = function(key, value, depth, current, result): key, value`
5. `result_before_def = function(result): void`
6. (`process`) (has to be definied using defs, see 3.5.13)
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.

3.3.14 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 3.3.2.

3.3.15 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.

3.3.16 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`.

3.3.17 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 3.3.2.

3.3.18 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 3.3.2.

3.3.19 Option "`true_aliases`"

See section 3.3.8.

3.3.20 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 3.1). The name of a key can be specified in three ways:

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

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

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

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

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

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

```lua
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, description, 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.

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

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

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

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

```
  -- To provide a default value for each naked key individually.
  default = true,

  -- Can serve as a comment.
  description = 'Describe your key-value pair.',

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

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

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

  -- The name of the key, can be omitted
  name = 'key',

  -- Convert opposite (naked) keys
  -- into a boolean value and store this boolean under a target key:
  --   show -> opposite_keys = true
  --   hide -> opposite_keys = false
  -- Short form: opposite_keys = { 'show', 'hide' }
  opposite_keys = { [true] = 'show', [false] = 'hide' },

  -- Pick a value by its data type:
  -- 'any', 'string', 'number', 'dimension', 'integer', 'boolean'.
  pick = false, -- 'false' disables the picking.

  -- A function whose return value is passed to the key.
  process = function(value, input, result, unknown)
    return value
  end,

  -- To enforce that a key must be specified.
  required = false,

  -- To build nested key-value pair definitions.
  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.

```lua
-- 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.

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

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

```lua
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.

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

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

```lua
    parse('key=unknown')
    -- error message:
    --- 'luakeys error [E004]: 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'`, `'list'`. A type conversion can fail with the three data types `'dimension'`, `'integer'`, `'number'`. Then an error message is displayed.

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

```
  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 (3.3.5) 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 "description"

This attribute is currently not processed further. It can serve as a comment.

3.5.7 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!'
```

3.5.8 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%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:
    -- 'luakeys error [E009]: The value "1978-12-XX" of the key "birthday"
    --  does not match "^%d%d%d%d-%d%d%-%d%d$"!'
```

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

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

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

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

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

3.5.10 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 "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 }
```

Opposing key pairs can be specified in a short form, namely as a list: The opposite key, which represents the true value, must be specified first in this list, followed by the false value.

```
local parse = luakeys.define({
  visibility = { opposite_keys = { 'show', 'hide' } },
})
```

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

3.5.13 Attribute "process"

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

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

The following example demonstrates the `value` parameter:

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

The following example demonstrates the `input` parameter:

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

The following example demonstrates the `result` parameter:

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

The following example demonstrates the `unknown` parameter:

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

```lua
    parse('key=1') -- throws error message: 'luakeys error [E019]: Unknown keys:
    ↪  "unknown_key=true,"'
```

3.5.14 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: 'luakeys error [E012]: 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: 'luakeys error [E012]: Missing required key
      ↪  "important2"!'
```

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

```
      parse2('unimportant')
      -- throws error message: 'luakeys error [E012]: Missing required key
      ↪  "important1"!'
```

3.5.15 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. Some 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.list(value): boolean"

```
-- true
equal(luakeys.is.list({ 'one', 'two', 'three' }), true)
equal(luakeys.is.list({ [1] = 'one', [2] = 'two', [3] = 'three' }),
  true)

-- false
equal(luakeys.is.list({ one = 'one', two = 'two', three = 'three' }),
  false)
equal(luakeys.is.list('one,two,three'), false)
equal(luakeys.is.list('list'), false)
equal(luakeys.is.list(nil), false)
```

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

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

3.11 Table “utils”

The `utils` table bundles some auxiliary functions.

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

---table
local merge_tables = utils.merge_tables
local clone_table = utils.clone_table
local remove_from_table = utils.remove_from_table
local get_table_keys = utils.get_table_keys
local get_table_size = utils.get_table_size
local get_array_size = utils.get_array_size

local tex_printf = utils.tex_printf

---error
local throw_error_message = utils.throw_error_message
local throw_error_code = utils.throw_error_code

local scan_oarg = utils.scan_oarg

---ansi_color
local colorize = utils.ansi_color.colorize
local red = utils.ansi_color.red
local green = utils.ansi_color.green
local yellow = utils.ansi_color.yellow
local blue = utils.ansi_color.blue
local magenta = utils.ansi_color.magenta
local cyan = utils.ansi_color.cyan

---log
local set = utils.log.set
local get = utils.log.set
local err = utils.log.error
local warn = utils.log.warn
local info = utils.log.info
local verbose = utils.log.verbose
local debug = utils.log.debug
```

3.11.1 Function “utils.merge_tables(target, source, overwrite): table”

The function `merge_tables` merges two tables into the first specified table. It copies keys from the ‘source‘ table into the ‘target‘ table. It returns the target table.

If the `overwrite` parameter is set to `true`, values in the target table are overwritten.

```
local result = luakeys.utils.merge_tables({ key = 'target' }, {
  key = 'source',
  key2 = 'new',
}, true)
luakeys.debug(result) -- { key = 'source', key2 = 'new' }
```

Give the parameter `overwrite` the value `false` to overwrite values in the target table.

```
local result2 = luakeys.utils.merge_tables({ key = 'target' }, {
  key = 'source',
  key2 = 'new',
}, false)
luakeys.debug(result2) -- { key = 'target', key2 = 'new' }
```

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

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

```
\input luakeys.tex

\def\mycmd{\directlua{
    local lk = luakeys.new()
    local oarg = lk.utils.scan_oarg('[', ']')
    if oarg then
      local keys = lk.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.

```lua
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
```

3.13 Table “error_messages”

```lua
local parse = luakeys.define({ key = { required = true } })

it('Default error', function()
  assert.has_error(function()
    parse('unknown')
  end, 'luakeys error [E012]: Missing required key "key"!')
end)

it('Custom error', function()
  luakeys.error_messages.E012 = 'The key @key is missing!'
  assert.has_error(function()
    parse('unknown')
  end, 'luakeys error [E012]: The key "key" is missing!')
end)
```

E001 : Unknown parse option: @unknown!

E002 : Unknown hook: @unknown!

E003 : Duplicate aliases @alias1 and @alias2 for key @key!

E004 : The value @value does not exist in the choices: @choices

E005 : Unknown data type: @unknown

E006 : The value @value of the key @key could not be converted into the data type @data_type!

E007 : The key @key belongs to the mutually exclusive group @exclusive_group and another key of the group named @another_key is already present!

E008 : def.match has to be a string

E009 : The value @value of the key @key does not match @match!

E010 : Usage: opposite_keys = "true_key", "false_key" or [true] = "true_key", [false] = "false_key"

E011 : Wrong data type in the "pick" attribute: @unknown. Allowed are: @data_types.

E012 : Missing required key @key!

`E013` : `The key definition must be a table! Got @data_type for key @key.`

`E014` : `Unknown definition attribute: @unknown`

`E015` : `Key name couldn't be detected!`

`E017` : `Unknown style to format keys: @unknown! Allowed styles are: @styles`

`E018` : `The option "format_keys" has to be a table not @data_type`

`E019` : `Unknown keys: @unknown`

`E020` : `Both opposite keys were given: @true and @false!`

`E021` : `Opposite key was specified more than once: @key!`

`E023` : `Don't use this function from the global luakeys table. Create a new instance using e. g.: local lk = luakeys.new()`

4 Syntax of the recognized key-value format

4.1 An attempt to put the syntax into words

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

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

⟨*list*⟩ ::= { ⟨*list-item*⟩ }

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

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

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

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

⟨*dimension*⟩ ::= ⟨*number*⟩ ⟨*unit*⟩

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

⟨*fractional*⟩ ::= '`.`' ⟨*integer*⟩

⟨*sign*⟩ ::= '`-`' | '`+`'

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

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

⟨*unit*⟩ ::= 'bp' | 'BP'
| 'cc' | 'CC'
| 'cm' | 'CM'
| 'dd' | 'DD'
| 'em' | 'EM'
| 'ex' | 'EX'
| 'in' | 'IN'
| 'mm' | 'MM'
| 'mu' | 'MU'
| 'nc' | 'NC'
| 'nd' | 'ND'
| 'pc' | 'PC'
| 'pt' | 'PT'
| 'px' | 'PX'
| 'sp' | 'SP'

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

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

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

... to be continued

4.3 Recognized data types

4.3.1 boolean

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

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

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

4.3.2 number

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

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

4.3.3 dimension

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

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

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

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

The next example illustrates the different notations of the dimensions.

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

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

4.3.4 string

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

```
local kv_string = [[
  without double quotes = no commas and equal signs are allowed,
  with double quotes = ", and = are allowed",
  escape quotes = "a quote \" sign",
  curly braces = "curly { } braces are allowed",
```

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

4.3.5 Naked keys

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

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

All recognized data types can be used as standalone values.

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

5 Examples

5.1 Extend and modify keys of existing macros

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

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

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

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

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

return print_image_macro
```

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

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

\myincludegrahics{test.png}

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

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

5.2 Process document class options

The options of a LaTeX document class can be accessed via the `\LuakeysGetClassOptions` macro. `\LuakeysGetClassOptions` is an alias for

`\luaescapestring{\@raw@classoptionslist}`.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{test-class}[2022/05/26 Test class to access the class options]
\DeclareOption*{} % suppresses the warning: LaTeX Warning: Unused global option(s):
\ProcessOptions\relax
\RequirePackage{luakeys}

\directlua{
  lk = luakeys.new()
}

% Using the macro \LuakeysGetClassOptions
\directlua{
  lk.debug(lk.parse('\LuakeysGetClassOptions'))
}

% Low level approach
\directlua{
  lk.debug(lk.parse('\luaescapestring{\@raw@classoptionslist}'))
}

\LoadClass{article}
```

```
\documentclass[test={key1,key2}]{test-class}

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

5.3 Process package options

The options of a LaTeX package can be accessed via the `\LuakeysGetPackageOptions` macro. `\LuakeysGetPackageOptions` is an alias for

`\luaescapestring{\@ptionlist{\@currname.\@currext}}`.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{test-package}[2022/11/27 Test package to access the package
↪  options]
\DeclareOption*{} % suppresses the error message: ! LaTeX Error: Unknown option
```

```
\ProcessOptions\relax
\RequirePackage{luakeys}

\directlua{
  lk = luakeys.new()
}

% Using the macro \LuakeysGetPackageOptions
\directlua{
  lk.debug(lk.parse('\LuakeysGetPackageOptions'))
}

% Low level approach
\directlua{
  lk.debug(lk.parse('\luaescapestring{\@ptionlist{\@currname.\@currext}}'))
}
```

```
\documentclass{article}
\usepackage[test={key1,key2}]{test-package}
\begin{document}
This document uses the package "test-package".
\end{document}
```

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

6.1 For plain TeX: luakeys-debug.tex

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

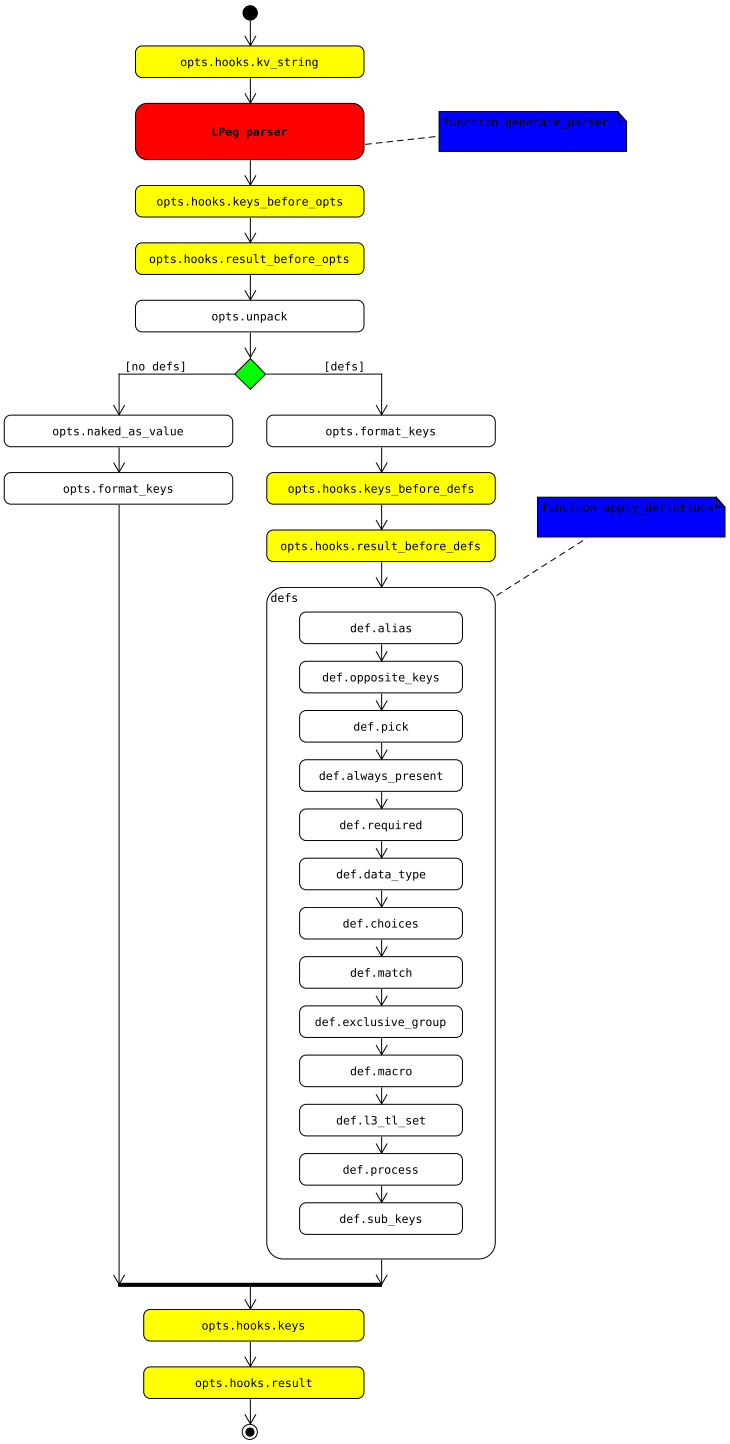
```
\input luakeys-debug.tex
\luakeysdebug{one,two,three}
\bye
```

6.2 For LaTeX: luakeys-debug.sty

An example of how to use the command in LaTeX:

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

7 Activity diagramm of the parsing process



8 Implementation

8.1 luakeys.lua

```
---luakeys.lua
---Copyright 2021-2023 Josef Friedrich
---
---This work may be distributed and/or modified under the
---conditions of the LaTeX Project Public License, either version 1.3c
---of this license or (at your option) any later version.
---The latest version of this license is in
---http://www.latex-project.org/lppl.txt
---and version 1.3c or later is part of all distributions of LaTeX
---version 2008/05/04 or later.
---
---This work has the LPPL maintenance status `maintained'.
---
---The Current Maintainer of this work is Josef Friedrich.
---
---This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
---luakeys-debug.sty and luakeys-debug.tex.
----A key-value parser written with Lpeg.
---
local lpeg = require('lpeg')

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

  token = {
    set_macro = function(csname, content, global)
    end,
  }
end

---
local utils = (function()
  ---
  ---Merge two tables into the first specified table.
  ---The `merge_tables` function copies keys from the `source` table
  ---to the `target` table. It returns the target table.
  ---
  ---https://stackoverflow.com/a/1283608/10193818
  ---
  ---@param target table # The target table where all values are copied.
  ---@param source table # The source table from which all values are copied.
  ---@param overwrite? boolean # Overwrite the values in the target table if they
  ↪  are present (default true).
  ---
  ---@return table target The modified target table.
  local function merge_tables(target, source, overwrite)
    if overwrite == nil then
      overwrite = true
    end
    for key, value in pairs(source) do
      if type(value) == 'table' and type(target[key] or false) ==
        'table' then
        merge_tables(target[key] or {}, source[key] or {}, overwrite)
```

```lua
      elseif (not overwrite and target[key] == nil) or
        (overwrite and target[key] ~= value) then
        target[key] = value
      end
    end
    return target
  end

  ---
  ---Clone a table, i.e. make a deep copy of the source table.
  ---
  ---http://lua-users.org/wiki/CopyTable
  ---
  ---@param source table # The source table to be cloned.
  ---
  ---@return table # A deep copy of the source table.
  local function clone_table(source)
    local copy
    if type(source) == 'table' then
      copy = {}
      for orig_key, orig_value in next, source, nil do
        copy[clone_table(orig_key)] = clone_table(orig_value)
      end
      setmetatable(copy, clone_table(getmetatable(source)))
    else ---number, string, boolean, etc
      copy = source
    end
    return copy
  end

  ---
  ---Remove an element from a table.
  ---
  ---@param source table # The source table.
  ---@param value any # The value to be removed from the table.
  ---
  ---@return any|nil # If the value was found, then this value, otherwise nil.
  local function remove_from_table(source, value)
    for index, v in pairs(source) do
      if value == v then
        source[index] = nil
        return value
      end
    end
  end

  ---
  ---Return the keys of a table as a sorted list (array like table).
  ---
  ---@param source table # The source table.
  ---
  ---@return table # An array table with the sorted key names.
  local function get_table_keys(source)
    local keys = {}
    for key in pairs(source) do
      table.insert(keys, key)
    end
    table.sort(keys)
    return keys
  end

  ---
```

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

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

---
---Print a formatted string.
---
---* `%d` or `%i`: Signed decimal integer
---* `%u`: Unsigned decimal integer
---* `%o`: Unsigned octal
---* `%x`: Unsigned hexadecimal integer
---* `%X`: Unsigned hexadecimal integer (uppercase)
---* `%f`: Decimal floating point, lowercase
---* `%e`: Scientific notation (mantissa/exponent), lowercase
---* `%E`: Scientific notation (mantissa/exponent), uppercase
---* `%g`: Use the shortest representation: %e or %f
---* `%G`: Use the shortest representation: %E or %F
---* `%a`: Hexadecimal floating point, lowercase
---* `%A`: Hexadecimal floating point, uppercase
---* `%c`: Character
---* `%s`: String of characters
---* `%p`: Pointer address        b8000000
---* `%%`: A `%` followed by another `%` character will write a single `%` to the stream.
---* `%q`: formats `booleans`, `nil`, `numbers`, and `strings` in a way that the result is a valid constant in Lua source code.
---
---http://www.lua.org/source/5.3/lstrlib.c.html#str_format
---
---@param format string # A string in the `printf` format
---@param ... any # A sequence of additional arguments, each containing a value to be used to replace a format specifier in the format string.
```

```
177     local function tex_printf(format, ...)
178       tex.print(string.format(format, ...))
179     end
180
181     ---
182     ---Throw a single error message.
183     ---
184     ---@param message string
185     ---@param help? table
186     local function throw_error_message(message, help)
187       if type(tex.error) == 'function' then
188         tex.error(message, help)
189       else
190         error(message)
191       end
192     end
193
194     ---
195     ---Throw an error by specifying an error code.
196     ---
197     ---@param error_messages table
198     ---@param error_code string
199     ---@param args? table
200     local function throw_error_code(error_messages,
201       error_code,
202       args)
203       local template = error_messages[error_code]
204
205       ---
206       ---@param message string
207       ---@param a table
208       ---
209       ---@return string
210       local function replace_args(message, a)
211         for key, value in pairs(a) do
212           if type(value) == 'table' then
213             value = table.concat(value, ', ')
214           end
215           message = message:gsub('@' .. key,
216             '"' .. tostring(value) .. '"')
217         end
218         return message
219       end
220
221       ---
222       ---@param list table
223       ---@param a table
224       ---
225       ---@return table
226       local function replace_args_in_list(list, a)
227         for index, message in ipairs(list) do
228           list[index] = replace_args(message, a)
229         end
230         return list
231       end
232
233       ---
234       ---@type string
235       local message
236       ---@type table
237       local help = {}
238
```

```
239       if type(template) == 'table' then
240         message = template[1]
241         if args ~= nil then
242           help = replace_args_in_list(template[2], args)
243         else
244           help = template[2]
245         end
246       else
247         message = template
248       end
249
250       if args ~= nil then
251         message = replace_args(message, args)
252       end
253
254       message = 'luakeys error [' .. error_code .. ']: ' .. message
255
256       for _, help_message in ipairs({
257         'You may be able to find more help in the documentation:',
258         'http://mirrors.ctan.org/macros/luatex/generic/luakeys/luakeys-doc.pdf',
259         'Or ask a question in the issue tracker on Github:',
260         'https://github.com/Josef-Friedrich/luakeys/issues',
261       }) do
262         table.insert(help, help_message)
263       end
264
265       throw_error_message(message, help)
266     end
267
268     ---
269     ---Scan for an optional argument.
270     ---
271     ---@param initial_delimiter? string # The character that marks the beginning of an
      ↪  optional argument (by default `[`).
272     ---@param end_delimiter? string # The character that marks the end of an optional
      ↪  argument (by default `]`).
273     ---
274     ---@return string|nil # The string that was enclosed by the delimiters. The
      ↪  delimiters themselves are not returned.
275     local function scan_oarg(initial_delimiter,
276       end_delimiter)
277       if initial_delimiter == nil then
278         initial_delimiter = '['
279       end
280
281       if end_delimiter == nil then
282         end_delimiter = ']'
283       end
284
285       ---
286       ---@param t Token
287       ---
288       ---@return string
289       local function convert_token(t)
290         if t.index ~= nil then
291           return utf8.char(t.index)
292         else
293           return '\\' .. t.csname
294         end
295       end
296
297       local function get_next_char()
```

```
298         local t = token.get_next()
299         return convert_token(t), t
300       end
301
302       local char, t = get_next_char()
303       if char == initial_delimiter then
304         local oarg = {}
305         char = get_next_char()
306         while char ~= end_delimiter do
307           table.insert(oarg, char)
308           char = get_next_char()
309         end
310         return table.concat(oarg, '')
311       else
312         token.put_next(t)
313       end
314     end
315
316     local function visit_tree(tree, callback_func)
317       if type(tree) ~= 'table' then
318         throw_error_message(
319           'Parameter "tree" has to be a table, got: ' ..
320             tostring(tree))
321       end
322       local function visit_tree_recursive(tree,
323         current,
324         result,
325         depth,
326         callback_func)
327         for key, value in pairs(current) do
328           if type(value) == 'table' then
329             value = visit_tree_recursive(tree, value, {}, depth + 1,
330               callback_func)
331           end
332
333           key, value = callback_func(key, value, depth, current, tree)
334
335           if key ~= nil and value ~= nil then
336             result[key] = value
337           end
338         end
339         if next(result) ~= nil then
340           return result
341         end
342       end
343
344       local result =
345         visit_tree_recursive(tree, tree, {}, 1, callback_func)
346
347       if result == nil then
348         return {}
349       end
350       return result
351     end
352
353     ---@alias ColorName
    ↪  'black'|'red'|'green'|'yellow'|'blue'|'magenta'|'cyan'|'white'|'reset'
354     ---@alias ColorMode 'bright'|'dim'
355
356     ---
357     ---Small library to surround strings with ANSI color codes.
358     --
```

```lua
---[SGR (Select Graphic Rendition)
↪  Parameters](https://en.wikipedia.org/wiki/ANSI_escape_code#SGR_(Select_Graphic_Rendition)_parameters)
---
---__attributes__
---
---| color      |code|
---|------------|----|
---| reset      |  0 |
---| clear      |  0 |
---| bright     |  1 |
---| dim        |  2 |
---| underscore |  4 |
---| blink      |  5 |
---| reverse    |  7 |
---| hidden     |  8 |
---
---__foreground__
---
---| color      |code|
---|------------|----|
---| black      | 30 |
---| red        | 31 |
---| green      | 32 |
---| yellow     | 33 |
---| blue       | 34 |
---| magenta    | 35 |
---| cyan       | 36 |
---| white      | 37 |
---
---__background__
---
---| color      |code|
---|------------|----|
---| onblack    | 40 |
---| onred      | 41 |
---| ongreen    | 42 |
---| onyellow   | 43 |
---| onblue     | 44 |
---| onmagenta  | 45 |
---| oncyan     | 46 |
---| onwhite    | 47 |
local ansi_color = (function()

  ---
  ---@param code integer
  ---
  ---@return string
  local function format_color_code(code)
    return string.char(27) .. '[' .. tostring(code) .. 'm'
  end

  ---
  ---@private
  ---
  ---@param color ColorName # A color name.
  ---@param mode? ColorMode
  ---@param background? boolean # Colorize the background not the text.
  ---
  ---@return string
  local function get_color_code(color, mode, background)
    local output = ''
    local code
```

```
420
421         if mode == 'bright' then
422           output = format_color_code(1)
423         elseif mode == 'dim' then
424           output = format_color_code(2)
425         end
426
427         if not background then
428           if color == 'reset' then
429             code = 0
430           elseif color == 'black' then
431             code = 30
432           elseif color == 'red' then
433             code = 31
434           elseif color == 'green' then
435             code = 32
436           elseif color == 'yellow' then
437             code = 33
438           elseif color == 'blue' then
439             code = 34
440           elseif color == 'magenta' then
441             code = 35
442           elseif color == 'cyan' then
443             code = 36
444           elseif color == 'white' then
445             code = 37
446           else
447             code = 37
448           end
449         else
450           if color == 'black' then
451             code = 40
452           elseif color == 'red' then
453             code = 41
454           elseif color == 'green' then
455             code = 42
456           elseif color == 'yellow' then
457             code = 43
458           elseif color == 'blue' then
459             code = 44
460           elseif color == 'magenta' then
461             code = 45
462           elseif color == 'cyan' then
463             code = 46
464           elseif color == 'white' then
465             code = 47
466           else
467             code = 40
468           end
469         end
470         return output .. format_color_code(code)
471       end
472
473       ---
474       ---@param text any
475       ---@param color ColorName # A color name.
476       ---@param mode? ColorMode
477       ---@param background? boolean # Colorize the background not the text.
478       ---
479       ---@return string
480       local function colorize(text, color, mode, background)
481         return string.format('%s%s%s',
```

```lua
          get_color_code(color, mode, background), text,
          get_color_code('reset'))
      end

      return {
        colorize = colorize,

        ---
        ---@param text any
        ---
        ---@return string
        red = function(text)
          return colorize(text, 'red')
        end,

        ---
        ---@param text any
        ---
        ---@return string
        green = function(text)
          return colorize(text, 'green')
        end,

        ---@return string
        yellow = function(text)
          return colorize(text, 'yellow')
        end,

        ---
        ---@param text any
        ---
        ---@return string
        blue = function(text)
          return colorize(text, 'blue')
        end,

        ---
        ---@param text any
        ---
        ---@return string
        magenta = function(text)
          return colorize(text, 'magenta')
        end,

        ---
        ---@param text any
        ---
        ---@return string
        cyan = function(text)
          return colorize(text, 'cyan')
        end,
      }
    end)()

    ---
    ---A small logging library.
    ---
    ---Log levels:
    ---
    ---* 0: silent
    ---* 1: error (red)
    ---* 2: warn (yellow)
```

```
544     ---* 3: info (green)
545     ---* 4: verbose (blue)
546     ---* 5: debug (magenta)
547     ---
548     local log = (function()
549       ---@private
550       local opts = { level = 0 }
551
552       local function colorize_not(s)
553         return s
554       end
555
556       local colorize = colorize_not
557
558       ---@private
559       local function print_message(message, ...)
560         local args = {...}
561         for index, value in ipairs(args) do
562           args[index] = colorize(value)
563         end
564         print(string.format(message, table.unpack(args)))
565       end
566
567       ---
568       ---Set the log level.
569       ---
570       ---@param level 0|'silent'|1|'error'|2|'warn'|3|'info'|4|'verbose'|5|'debug'
571       local function set_log_level(level)
572         if type(level) == 'string' then
573           if level == 'silent' then
574             opts.level = 0
575           elseif level == 'error' then
576             opts.level = 1
577           elseif level == 'warn' then
578             opts.level = 2
579           elseif level == 'info' then
580             opts.level = 3
581           elseif level == 'verbose' then
582             opts.level = 4
583           elseif level == 'debug' then
584             opts.level = 5
585           else
586             throw_error_message(string.format('Unknown log level: %s',
587               level))
588           end
589         else
590           if level > 5 or level < 0 then
591             throw_error_message(string.format(
592               'Log level out of range 0-5: %s', level))
593           end
594           opts.level = level
595         end
596       end
597
598       ---
599       ---@return integer
600       local function get_log_level()
601         return opts.level
602       end
603
604       ---
605       ---Log at level 1 (error).
```

```lua
    ---
    ---The other log levels are: 0 (silent), 1 (error), 2 (warn), 3 (info), 4
    ↪  (verbose), 5 (debug).
    ---
    ---@param message string
    ---@param ... any
    local function error(message, ...)
      if opts.level >= 1 then
        colorize = ansi_color.red
        print_message(message, ...)
        colorize = colorize_not
      end
    end

    ---
    ---Log at level 2 (warn).
    ---
    ---The other log levels are: 0 (silent), 1 (error), 2 (warn), 3 (info), 4
    ↪  (verbose), 5 (debug).
    ---
    ---@param message string
    ---@param ... any
    local function warn(message, ...)
      if opts.level >= 2 then
        colorize = ansi_color.yellow
        print_message(message, ...)
        colorize = colorize_not
      end
    end

    ---
    ---Log at level 3 (info).
    ---
    ---The other log levels are: 0 (silent), 1 (error), 2 (warn), 3 (info), 4
    ↪  (verbose), 5 (debug).
    ---
    ---@param message string
    ---@param ... any
    local function info(message, ...)
      if opts.level >= 3 then
        colorize = ansi_color.green
        print_message(message, ...)
        colorize = colorize_not
      end
    end

    ---
    ---Log at level 4 (verbose).
    ---
    ---The other log levels are: 0 (silent), 1 (error), 2 (warn), 3 (info), 4
    ↪  (verbose), 5 (debug).
    ---
    ---@param message string
    ---@param ... any
    local function verbose(message, ...)
      if opts.level >= 4 then
        colorize = ansi_color.blue
        print_message(message, ...)
        colorize = colorize_not
      end
    end

```

```
664       ---
665       ---Log at level 5 (debug).
666       ---
667       ---The other log levels are: 0 (silent), 1 (error), 2 (warn), 3 (info), 4
          ↪  (verbose), 5 (debug).
668       ---
669       ---@param message string
670       ---@param ... any
671       local function debug(message, ...)
672         if opts.level >= 5 then
673           colorize = ansi_color.magenta
674           print_message(message, ...)
675           colorize = colorize_not
676         end
677       end
678
679       return {
680         set = set_log_level,
681         get = get_log_level,
682         error = error,
683         warn = warn,
684         info = info,
685         verbose = verbose,
686         debug = debug,
687       }
688     end)()
689
690     return {
691       merge_tables = merge_tables,
692       clone_table = clone_table,
693       remove_from_table = remove_from_table,
694       get_table_keys = get_table_keys,
695       get_table_size = get_table_size,
696       get_array_size = get_array_size,
697       visit_tree = visit_tree,
698       tex_printf = tex_printf,
699       throw_error_message = throw_error_message,
700       throw_error_code = throw_error_code,
701       scan_oarg = scan_oarg,
702       ansi_color = ansi_color,
703       log = log,
704     }
705   end)()
706
707   ---
708   ---Convert back to strings
709   ---@section
710   local visualizers = (function()
711
712     ---
713     ---Reverse the function
714     ---`parse(kv_string)`. It takes a Lua table and converts this table
715     ---into a key-value string. The resulting string usually has a
716     ---different order as the input table. In Lua only tables with
717     ---1-based consecutive integer keys (a.k.a. array tables) can be
718     ---parsed in order.
719     ---
720     ---@param result table # A table to be converted into a key-value string.
721     ---
722     ---@return string # A key-value string that can be passed to a TeX macro.
723     local function render(result)
724       local function render_inner(result)
```

```lua
    local output = {}
    local function add(text)
      table.insert(output, text)
    end
    for key, value in pairs(result) do
      if (key and type(key) == 'string') then
        if (type(value) == 'table') then
          if (next(value)) then
            add(key .. '={')
            add(render_inner(value))
            add('},')
          else
            add(key .. '={},')
          end
        else
          add(key .. '=' .. tostring(value) .. ',')
        end
      else
        add(tostring(value) .. ',')
      end
    end
    return table.concat(output)
  end
  return render_inner(result)
end

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

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

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

```

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

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

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

      for key, value in pairs(input) do
        if (key and type(key) == 'number' or type(key) == 'string') then
          key = format_key(key)

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

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

      return table.concat(output, line_break)
    end

    return start_bracket .. line_break .. stringify_inner(result, 1) ..
             line_break .. end_bracket
  end

  ---
  ---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.
  ---
  ---@param result table # A table to be printed to standard output for debugging
  ↪  purposes.
  local function debug(result)
    print('\n' .. stringify(result, false))
  end

  return { render = render, stringify = stringify, debug = debug }
```

```
end)()

---@class OptionCollection
---@field accumulated_result? table
---@field assignment_operator? string # default `=`
---@field convert_dimensions? boolean # default `false`
---@field debug? boolean # default `false`
---@field default? boolean # default `true`
---@field defaults? table
---@field defs? DefinitionCollection
---@field false_aliases? table default `{ 'false', 'FALSE', 'False' }`,
---@field format_keys? boolean # default `false`,
---@field group_begin? string default `{`,
---@field group_end? string default `}`,
---@field hooks? HookCollection
---@field invert_flag? string default `!`
---@field list_separator? string default `,`
---@field naked_as_value? boolean # default `false`
---@field no_error? boolean # default `false`
---@field quotation_begin? string `"`
---@field quotation_end? string `"`
---@field true_aliases? table `{ 'true', 'TRUE', 'True' }`
---@field unpack? boolean # default `true`

---@alias KeysHook fun(key: string, value: any, depth: integer, current: table,
↪  result: table): string, any
---@alias ResultHook fun(result: table): nil

---@class HookCollection
---@field kv_string? fun(kv_string: string): string
---@field keys_before_opts? KeysHook
---@field result_before_opts? ResultHook
---@field keys_before_def? KeysHook
---@field result_before_def? ResultHook
---@field keys? KeysHook
---@field result? ResultHook

---@alias ProcessFunction fun(value: any, input: table, result: table, unknown:
↪  table): any

---@alias PickDataType 'string'|'number'|'dimension'|'integer'|'boolean'|'any'

---@class Definition
---@field alias? string|table
---@field always_present? boolean
---@field choices? table
---@field data_type? 'boolean'|'dimension'|'integer'|'number'|'string'|'list'
---@field default? any
---@field description? string
---@field exclusive_group? string
---@field l3_tl_set? string
---@field macro? string
---@field match? string
---@field name? string
---@field opposite_keys? table
---@field pick? PickDataType|PickDataType[]|false
---@field process? ProcessFunction
---@field required? boolean
---@field sub_keys? table<string, Definition>

---@alias DefinitionCollection table<string|number, Definition>

```

```lua
local namespace = {
  opts = {
    accumulated_result = false,
    assignment_operator = '=',
    convert_dimensions = false,
    debug = false,
    default = true,
    defaults = false,
    defs = false,
    false_aliases = { 'false', 'FALSE', 'False' },
    format_keys = false,
    group_begin = '{',
    group_end = '}',
    hooks = {},
    invert_flag = '!',
    list_separator = ',',
    naked_as_value = false,
    no_error = false,
    quotation_begin = '"',
    quotation_end = '"',
    true_aliases = { 'true', 'TRUE', 'True' },
    unpack = true,
  },

  hooks = {
    kv_string = true,
    keys_before_opts = true,
    result_before_opts = true,
    keys_before_def = true,
    result_before_def = true,
    keys = true,
    result = true,
  },

  attrs = {
    alias = true,
    always_present = true,
    choices = true,
    data_type = true,
    default = true,
    description = true,
    exclusive_group = true,
    l3_tl_set = true,
    macro = true,
    match = true,
    name = true,
    opposite_keys = true,
    pick = true,
    process = true,
    required = true,
    sub_keys = true,
  },

  error_messages = {
    E001 = {
      'Unknown parse option: @unknown!',
      { 'The available options are:', '@opt_names' },
    },
    E002 = {
      'Unknown hook: @unknown!',
      { 'The available hooks are:', '@hook_names' },
    },
```

```
968       E003 = 'Duplicate aliases @alias1 and @alias2 for key @key!',
969       E004 = 'The value @value does not exist in the choices: @choices',
970       E005 = 'Unknown data type: @unknown',
971       E006 = 'The value @value of the key @key could not be converted into the data
          ↪  type @data_type!',
972       E007 = 'The key @key belongs to the mutually exclusive group @exclusive_group
          ↪  and another key of the group named @another_key is already present!',
973       E008 = 'def.match has to be a string',
974       E009 = 'The value @value of the key @key does not match @match!',
975
976       E011 = 'Wrong data type in the "pick" attribute: @unknown. Allowed are:
          ↪  @data_types.',
977       E012 = 'Missing required key @key!',
978       E013 = 'The key definition must be a table! Got @data_type for key @key.',
979       E014 = {
980         'Unknown definition attribute: @unknown',
981         { 'The available attributes are:', '@attr_names' },
982       },
983       E015 = 'Key name couldn't be detected!',
984       E017 = 'Unknown style to format keys: @unknown! Allowed styles are: @styles',
985       E018 = 'The option "format_keys" has to be a table not @data_type',
986       E019 = 'Unknown keys: @unknown',
987
988       ---Input / parsing error
989       E021 = 'Opposite key was specified more than once: @key!',
990       E020 = 'Both opposite keys were given: @true and @false!',
991       ---Config error (wrong configuration of luakeys)
992       E010 = 'Usage: opposite_keys = { "true_key", "false_key" } or { [true] =
          ↪  "true_key", [false] = "false_key" } ',
993       E023 = {
994         'Don't use this function from the global luakeys table. Create a new instance
            ↪  using e. g.: local lk = luakeys.new()',
995         {
996           'This functions should not be used from the global luakeys table:',
997           'parse()',
998           'save()',
999           'get()',
1000        },
1001      },
1002    },
1003  }
1004
1005  ---
1006  ---Main entry point of the module.
1007  ---
1008  ---The return value is intentional not documented so the Lua language server can
      ↪  figure out the types.
1009  local function main()
1010
1011    ---The default options.
1012    ---@type OptionCollection
1013    local default_opts = utils.clone_table(namespace.opts)
1014
1015    local error_messages = utils.clone_table(namespace.error_messages)
1016
1017    ---
1018    ---@param error_code string
1019    ---@param args? table
1020    local function throw_error(error_code, args)
1021      utils.throw_error_code(error_messages, error_code, args)
1022    end
1023
```

```
1024    ---
1025    ---Normalize the parse options.
1026    ---
1027    ---@param opts? OptionCollection|unknown # Options in a raw format. The table may
        ↪  be empty or some keys are not set.
1028    ---
1029    ---@return OptionCollection
1030    local function normalize_opts(opts)
1031      if type(opts) ~= 'table' then
1032        opts = {}
1033      end
1034      for key, _ in pairs(opts) do
1035        if namespace.opts[key] == nil then
1036          throw_error('E001', {
1037            unknown = key,
1038            opt_names = utils.get_table_keys(namespace.opts),
1039          })
1040        end
1041      end
1042      local old_opts = opts
1043      opts = {}
1044      for name, _ in pairs(namespace.opts) do
1045        if old_opts[name] ~= nil then
1046          opts[name] = old_opts[name]
1047        else
1048          opts[name] = default_opts[name]
1049        end
1050      end
1051
1052      for hook in pairs(opts.hooks) do
1053        if namespace.hooks[hook] == nil then
1054          throw_error('E002', {
1055            unknown = hook,
1056            hook_names = utils.get_table_keys(namespace.hooks),
1057          })
1058        end
1059      end
1060      return opts
1061    end
1062
1063    local l3_code_cctab = 10
1064
1065    ---
1066    ---Parser / Lpeg related
1067    ---@section
1068
1069    ---Generate the PEG parser using Lpeg.
1070    ---
1071    ---Explanations of some LPeg notation forms:
1072    ---
1073    ---* `patt ^ 0` = `expression *`
1074    ---* `patt ^ 1` = `expression +`
1075    ---* `patt ^ -1` = `expression ?`
1076    ---* `patt1 * patt2` = `expression1 expression2`: Sequence
1077    ---* `patt1 + patt2` = `expression1 / expression2`: Ordered choice
1078    ---
1079    ---* [TUGboat article: Parsing complex data formats in LuaTEX with
        ↪  LPEG](https://tug.or-g/TUGboat/tb40-2/tb125menke-Patterndf)
1080    ---
1081    ---@param initial_rule string # The name of the first rule of the grammar table
        ↪  passed to the `lpeg.P(attern)` function (e. g. `list`, `number`).
1082    ---@param opts? table # Whether the dimensions should be converted to scaled
        ↪  points (by default `false`).
```

```
1083      ---
1084      ---@return userdata # The parser.
1085      local function generate_parser(initial_rule, opts)
1086        if type(opts) ~= 'table' then
1087          opts = normalize_opts(opts)
1088        end
1089
1090        local Variable = lpeg.V
1091        local Pattern = lpeg.P
1092        local Set = lpeg.S
1093        local Range = lpeg.R
1094        local CaptureGroup = lpeg.Cg
1095        local CaptureFolding = lpeg.Cf
1096        local CaptureTable = lpeg.Ct
1097        local CaptureConstant = lpeg.Cc
1098        local CaptureSimple = lpeg.C
1099
1100        ---Optional whitespace
1101        local white_space = Set(' \t\n\r')
1102
1103        ---Match literal string surrounded by whitespace
1104        local ws = function(match)
1105          return white_space ^ 0 * Pattern(match) * white_space ^ 0
1106        end
1107
1108        local line_up_pattern = function(patterns)
1109          local result
1110          for _, pattern in ipairs(patterns) do
1111            if result == nil then
1112              result = Pattern(pattern)
1113            else
1114              result = result + Pattern(pattern)
1115            end
1116          end
1117          return result
1118        end
1119
1120        ---
1121        ---Convert a dimension to an normalized dimension string or an
1122        ---integer in the scaled points format.
1123        ---
1124        ---@param input string
1125        ---
1126        ---@return integer|string # A dimension as an integer or a dimension string.
1127        local capture_dimension = function(input)
1128          ---Remove all whitespaces
1129          input = input:gsub('%s+', '')
1130          ---Convert the unit string into lowercase.
1131          input = input:lower()
1132          if opts.convert_dimensions then
1133            return tex.sp(input)
1134          else
1135            return input
1136          end
1137        end
1138
1139        ---
1140        ---Add values to a table in two modes:
1141        ---
1142        ---Key-value pair:
1143        ---
1144        ---If `arg1` and `arg2` are not nil, then `arg1` is the key and `arg2` is the
```

```
1145        ---value of a new table entry.
1146        ---
1147        ---Indexed value:
1148        ---
1149        ---If `arg2` is nil, then `arg1` is the value and is added as an indexed
1150        ---(by an integer) value.
1151        ---
1152        ---@param result table # The result table to which an additional key-value pair
            ↪  or value should to be added
1153        ---@param arg1 any # The key or the value.
1154        ---@param arg2? any # Always the value.
1155        ---
1156        ---@return table # The result table to which an additional key-value pair or
            ↪  value has been added.
1157        local add_to_table = function(result, arg1, arg2)
1158          if arg2 == nil then
1159            local index = #result + 1
1160            return rawset(result, index, arg1)
1161          else
1162            return rawset(result, arg1, arg2)
1163          end
1164        end
1165
1166        -- LuaFormatter off
1167        return Pattern({
1168          [1] = initial_rule,
1169
1170          ---list_item*
1171          list = CaptureFolding(
1172            CaptureTable('') * Variable('list_item')^0,
1173            add_to_table
1174          ),
1175
1176          ---'{' list '}'
1177          list_container =
1178            ws(opts.group_begin) * Variable('list') * ws(opts.group_end),
1179
1180          ---( list_container / key_value_pair / value ) ','?
1181          list_item =
1182            CaptureGroup(
1183              Variable('list_container') +
1184              Variable('key_value_pair') +
1185              Variable('value')
1186            ) * ws(opts.list_separator)^-1,
1187
1188          ---key '=' (list_container / value)
1189          key_value_pair =
1190            (Variable('key') * ws(opts.assignment_operator)) *
                ↪  (Variable('list_container') + Variable('value')),
1191
1192          ---number / string_quoted / string_unquoted
1193          key =
1194            Variable('number') +
1195            Variable('string_quoted') +
1196            Variable('string_unquoted'),
1197
1198          ---boolean !value / dimension !value / number !value / string_quoted !value /
            ↪  string_unquoted
1199          ---!value -> Not-predicate -> * -Variable('value')
1200          value =
1201            Variable('boolean') * -Variable('value') +
1202            Variable('dimension') * -Variable('value') +
```

```
      Variable('number') * -Variable('value')  +
      Variable('string_quoted') * -Variable('value') +
      Variable('string_unquoted'),

    ---for is.boolean()
    boolean_only = Variable('boolean') * -1,

    ---boolean_true / boolean_false
    boolean =
      (
        Variable('boolean_true') * CaptureConstant(true) +
        Variable('boolean_false') * CaptureConstant(false)
      ),

    boolean_true = line_up_pattern(opts.true_aliases),

    boolean_false = line_up_pattern(opts.false_aliases),

    ---for is.dimension()
    dimension_only = Variable('dimension') * -1,

    dimension = (
      Variable('tex_number') * white_space^0 *
      Variable('unit')
    ) / capture_dimension,

    ---for is.number()
    number_only = Variable('number') * -1,

    ---capture number
    number = Variable('tex_number') / tonumber,

    ---sign? white_space? (integer+ fractional? / fractional)
    tex_number =
      Variable('sign')^0 * white_space^0 *
      (Variable('integer')^1 * Variable('fractional')^0) +
      Variable('fractional'),

    sign = Set('-+'),

    fractional = Pattern('.') * Variable('integer')^1,

    integer = Range('09')^1,

    ---'bp' / 'BP' / 'cc' / etc.

    ↪  ---https://raw.githubusercontent.com/latex3/lualibs/master/lualibs-util-dim.lua

    ↪  ---https://github.com/TeX-Live/luatex/blob/51db1985f5500dafd2393aa2e403fefa57d3cb76/source/texk/we
    unit =
      Pattern('bp') + Pattern('BP') +
      Pattern('cc') + Pattern('CC') +
      Pattern('cm') + Pattern('CM') +
      Pattern('dd') + Pattern('DD') +
      Pattern('em') + Pattern('EM') +
      Pattern('ex') + Pattern('EX') +
      Pattern('in') + Pattern('IN') +
      Pattern('mm') + Pattern('MM') +
      Pattern('mu') + Pattern('MU') +
      Pattern('nc') + Pattern('NC') +
      Pattern('nd') + Pattern('ND') +
      Pattern('pc') + Pattern('PC') +
```

```
          Pattern('pt') + Pattern('PT') +
          Pattern('px') + Pattern('PX') +
          Pattern('sp') + Pattern('SP'),

        ---'"' ('\"' / !'"')* '"'
        string_quoted =
          white_space^0 * Pattern(opts.quotation_begin) *
          CaptureSimple((Pattern('\\' .. opts.quotation_end) + 1 -
          ↪  Pattern(opts.quotation_end))^0) *
          Pattern(opts.quotation_end) * white_space^0,

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

        word_unquoted = (1 - white_space - Set(
          opts.group_begin ..
          opts.group_end ..
          opts.assignment_operator  ..
          opts.list_separator))^1
      })
  -- LuaFormatter on
    end

    local is = {
      boolean = function(value)
        if value == nil then
          return false
        end
        if type(value) == 'boolean' then
          return true
        end
        local parser = generate_parser('boolean_only')
        local result = parser:match(tostring(value))
        return result ~= nil
      end,

      dimension = function(value)
        if value == nil then
          return false
        end
        local parser = generate_parser('dimension_only')
        local result = parser:match(tostring(value))
        return result ~= nil
      end,

      integer = function(value)
        local n = tonumber(value)
        if n == nil then
          return false
        end
        return n == math.floor(n)
      end,

      number = function(value)
        if value == nil then
          return false
        end
        if type(value) == 'number' then
```

```
          return true
        end
        local parser = generate_parser('number_only')
        local result = parser:match(tostring(value))
        return result ~= nil
      end,

      string = function(value)
        return type(value) == 'string'
      end,

      list = function(value)
        if type(value) ~= 'table' then
          return false
        end

        for k, _ in pairs(value) do
          if type(k) ~= 'number' then
            return false
          end
        end
        return true
      end,

      any = function(value)
        return true
      end,
    }

    ---
    ---Apply the key-value-pair definitions (defs) on an input table in a
    ---recursive fashion.
    ---
    ---@param defs table # A table containing all definitions.
    ---@param opts table # The parse options table.
    ---@param input table # The current input table.
    ---@param output table # The current output table.
    ---@param unknown table # Always the root unknown table.
    ---@param key_path table # An array of key names leading to the current
    ---@param input_root table # The root input table input and output table.
    local function apply_definitions(defs,
      opts,
      input,
      output,
      unknown,
      key_path,
      input_root)
      local exclusive_groups = {}

      local function add_to_key_path(key_path, key)
        local new_key_path = {}

        for index, value in ipairs(key_path) do
          new_key_path[index] = value
        end

        table.insert(new_key_path, key)
        return new_key_path
      end

      local function get_default_value(def)
        if def.default ~= nil then
```

```
1386            return def.default
1387          elseif opts ~= nil and opts.default ~= nil then
1388            return opts.default
1389          end
1390          return true
1391        end
1392
1393        local function find_value(search_key, def)
1394          if input[search_key] ~= nil then
1395            local value = input[search_key]
1396            input[search_key] = nil
1397            return value
1398            ---naked keys: values with integer keys
1399          elseif utils.remove_from_table(input, search_key) ~= nil then
1400            return get_default_value(def)
1401          end
1402        end
1403
1404        local apply = {
1405          alias = function(value, key, def)
1406            if type(def.alias) == 'string' then
1407              def.alias = { def.alias }
1408            end
1409            local alias_value
1410            local used_alias_key
1411            ---To get an error if the key and an alias is present
1412            if value ~= nil then
1413              alias_value = value
1414              used_alias_key = key
1415            end
1416            for _, alias in ipairs(def.alias) do
1417              local v = find_value(alias, def)
1418              if v ~= nil then
1419                if alias_value ~= nil then
1420                  throw_error('E003', {
1421                    alias1 = used_alias_key,
1422                    alias2 = alias,
1423                    key = key,
1424                  })
1425                end
1426                used_alias_key = alias
1427                alias_value = v
1428              end
1429            end
1430            if alias_value ~= nil then
1431              return alias_value
1432            end
1433          end,
1434
1435          always_present = function(value, key, def)
1436            if value == nil and def.always_present then
1437              return get_default_value(def)
1438            end
1439          end,
1440
1441          choices = function(value, key, def)
1442            if value == nil then
1443              return
1444            end
1445            if def.choices ~= nil and type(def.choices) == 'table' then
1446              local is_in_choices = false
1447              for _, choice in ipairs(def.choices) do
```

```
            if value == choice then
              is_in_choices = true
            end
          end
          if not is_in_choices then
            throw_error('E004', { value = value, choices = def.choices })
          end
        end
      end,

      data_type = function(value, key, def)
        if value == nil then
          return
        end
        if def.data_type ~= nil then
          local converted
          ---boolean
          if def.data_type == 'boolean' then
            if value == 0 or value == '' or not value then
              converted = false
            else
              converted = true
            end
            ---dimension
          elseif def.data_type == 'dimension' then
            if is.dimension(value) then
              converted = value
            end
            ---integer
          elseif def.data_type == 'integer' then
            if is.number(value) then
              local n = tonumber(value)
              if type(n) == 'number' and n ~= nil then
                converted = math.floor(n)
              end
            end
            ---number
          elseif def.data_type == 'number' then
            if is.number(value) then
              converted = tonumber(value)
            end
            ---string
          elseif def.data_type == 'string' then
            converted = tostring(value)
            ---list
          elseif def.data_type == 'list' then
            if is.list(value) then
              converted = value
            end
          else
            throw_error('E005', { data_type = def.data_type })
          end
          if converted == nil then
            throw_error('E006', {
              value = value,
              key = key,
              data_type = def.data_type,
            })
          else
            return converted
          end
        end
```

```
1510        end,
1511
1512        exclusive_group = function(value, key, def)
1513          if value == nil then
1514            return
1515          end
1516          if def.exclusive_group ~= nil then
1517            if exclusive_groups[def.exclusive_group] ~= nil then
1518              throw_error('E007', {
1519                key = key,
1520                exclusive_group = def.exclusive_group,
1521                another_key = exclusive_groups[def.exclusive_group],
1522              })
1523            else
1524              exclusive_groups[def.exclusive_group] = key
1525            end
1526          end
1527        end,
1528
1529        l3_tl_set = function(value, key, def)
1530          if value == nil then
1531            return
1532          end
1533          if def.l3_tl_set ~= nil then
1534            tex.print(l3_code_cctab,
1535              '\\tl_set:Nn \\g_' .. def.l3_tl_set .. '_tl')
1536            tex.print('{' .. value .. '}')
1537          end
1538        end,
1539
1540        macro = function(value, key, def)
1541          if value == nil then
1542            return
1543          end
1544          if def.macro ~= nil then
1545            token.set_macro(def.macro, value, 'global')
1546          end
1547        end,
1548
1549        match = function(value, key, def)
1550          if value == nil then
1551            return
1552          end
1553          if def.match ~= nil then
1554            if type(def.match) ~= 'string' then
1555              throw_error('E008')
1556            end
1557            local match = string.match(value, def.match)
1558            if match == nil then
1559              throw_error('E009', {
1560                value = value,
1561                key = key,
1562                match = def.match:gsub('%%', '%%%%'),
1563              })
1564            else
1565              return match
1566            end
1567          end
1568        end,
1569
1570        opposite_keys = function(value, key, def)
1571          if def.opposite_keys ~= nil then
```

```
1572            local function get_value(key1, key2)
1573              local opposite_name
1574              if def.opposite_keys[key1] ~= nil then
1575                opposite_name = def.opposite_keys[key1]
1576              elseif def.opposite_keys[key2] ~= nil then
1577                opposite_name = def.opposite_keys[key2]
1578              end
1579              return opposite_name
1580            end
1581            local true_key = get_value(true, 1)
1582            local false_key = get_value(false, 2)
1583            if true_key == nil or false_key == nil then
1584              throw_error('E010')
1585            end
1586
1587            ---@param v string
1588            local function remove_values(v)
1589              local count = 0
1590              while utils.remove_from_table(input, v) do
1591                count = count + 1
1592              end
1593              return count
1594            end
1595
1596            local true_count = remove_values(true_key)
1597            local false_count = remove_values(false_key)
1598
1599            if true_count > 1 then
1600              throw_error('E021', { key = true_key })
1601            end
1602
1603            if false_count > 1 then
1604              throw_error('E021', { key = false_key })
1605            end
1606
1607            if true_count > 0 and false_count > 0 then
1608              throw_error('E020',
1609                { ['true'] = true_key, ['false'] = false_key })
1610            end
1611            if true_count == 0 and false_count == 0 then
1612              return
1613            end
1614            return true_count == 1 or false_count == 0
1615          end
1616        end,
1617
1618        process = function(value, key, def)
1619          if value == nil then
1620            return
1621          end
1622          if def.process ~= nil and type(def.process) == 'function' then
1623            return def.process(value, input_root, output, unknown)
1624          end
1625        end,
1626
1627        pick = function(value, key, def)
1628          if def.pick then
1629            local pick_types
1630
1631            ---Allow old deprecated attribut pick = true
1632            if def.pick == true then
1633              pick_types = { 'any' }
```

```
1634            elseif type(def.pick) == 'table' then
1635              pick_types = def.pick
1636            else
1637              pick_types = { def.pick }
1638            end
1639
1640            ---Check if the pick attribute is valid
1641            for _, pick_type in ipairs(pick_types) do
1642              if type(pick_type) == 'string' and is[pick_type] == nil then
1643                throw_error('E011', {
1644                  unknown = tostring(pick_type),
1645                  data_types = {
1646                    'any',
1647                    'boolean',
1648                    'dimension',
1649                    'integer',
1650                    'number',
1651                    'string',
1652                  },
1653                })
1654              end
1655            end
1656
1657            ---The key has already a value. We leave the function at this
1658            ---point to be able to check the pick attribute for errors
1659            ---beforehand.
1660            if value ~= nil then
1661              return value
1662            end
1663
1664            for _, pick_type in ipairs(pick_types) do
1665              for i, v in pairs(input) do
1666                ---We can not use ipairs here. `ipairs(t)` iterates up to the
1667                ---first absent index. Values are deleted from the `input`
1668                ---table.
1669                if type(i) == 'number' then
1670                  local picked_value = nil
1671                  if is[pick_type](v) then
1672                    picked_value = v
1673                  end
1674
1675                  if picked_value ~= nil then
1676                    input[i] = nil
1677                    return picked_value
1678                  end
1679                end
1680              end
1681            end
1682          end
1683        end,
1684
1685        required = function(value, key, def)
1686          if def.required ~= nil and def.required and value == nil then
1687            throw_error('E012', { key = key })
1688          end
1689        end,
1690
1691        sub_keys = function(value, key, def)
1692          if def.sub_keys ~= nil then
1693            local v
1694            ---To get keys defined with always_present
1695            if value == nil then
```

```
            v = {}
          elseif type(value) == 'string' then
            v = { value }
          elseif type(value) == 'table' then
            v = value
          end
          v = apply_definitions(def.sub_keys, opts, v, output[key],
            unknown, add_to_key_path(key_path, key), input_root)
          if utils.get_table_size(v) > 0 then
            return v
          end
        end
      end,
    }

    ---standalone values are removed.
    ---For some callbacks and the third return value of parse, we
    ---need an unchanged raw result from the parse function.
    input = utils.clone_table(input)
    if output == nil then
      output = {}
    end
    if unknown == nil then
      unknown = {}
    end
    if key_path == nil then
      key_path = {}
    end

    for index, def in pairs(defs) do
      ---Find key and def
      local key
      ---`{ key1 = { }, key2 = { } }`
      if type(def) == 'table' and def.name == nil and type(index) ==
        'string' then
        key = index
        ---`{ { name = 'key1' }, { name = 'key2' } }`
      elseif type(def) == 'table' and def.name ~= nil then
        key = def.name
        ---Definitions as strings in an array: `{ 'key1', 'key2' }`
      elseif type(index) == 'number' and type(def) == 'string' then
        key = def
        def = { default = get_default_value({}) }
      end

      if type(def) ~= 'table' then
        throw_error('E013', { data_type = tostring(def), key = index }) ---key is
        ↪  nil
      end

      for attr, _ in pairs(def) do
        if namespace.attrs[attr] == nil then
          throw_error('E014', {
            unknown = attr,
            attr_names = utils.get_table_keys(namespace.attrs),
          })
        end
      end

      if key == nil then
        throw_error('E015')
      end
```

```
1757
1758        local value = find_value(key, def)
1759
1760        for _, def_opt in ipairs({
1761          'alias',
1762          'opposite_keys',
1763          'pick',
1764          'always_present',
1765          'required',
1766          'data_type',
1767          'choices',
1768          'match',
1769          'exclusive_group',
1770          'macro',
1771          'l3_tl_set',
1772          'process',
1773          'sub_keys',
1774        }) do
1775          if def[def_opt] ~= nil then
1776            local tmp_value = apply[def_opt](value, key, def)
1777            if tmp_value ~= nil then
1778              value = tmp_value
1779            end
1780          end
1781        end
1782
1783        output[key] = value
1784      end
1785
1786      if utils.get_table_size(input) > 0 then
1787        ---Move to the current unknown table.
1788        local current_unknown = unknown
1789        for _, key in ipairs(key_path) do
1790          if current_unknown[key] == nil then
1791            current_unknown[key] = {}
1792          end
1793          current_unknown = current_unknown[key]
1794        end
1795
1796        ---Copy all unknown key-value-pairs to the current unknown table.
1797        for key, value in pairs(input) do
1798          current_unknown[key] = value
1799        end
1800      end
1801
1802      return output, unknown
1803    end
1804
1805    ---
1806    ---Parse a LaTeX/TeX style key-value string into a Lua table.
1807    ---
1808    ---@param kv_string string # A string in the TeX/LaTeX style key-value format as
       ↪  described above.
1809    ---@param opts? OptionCollection # A table containing options.
1810    ---
1811    ---@return table result # The final result of all individual parsing and
       ↪  normalization steps.
1812    ---@return table unknown # A table with unknown, undefinied key-value pairs.
1813    ---@return table raw # The unprocessed, raw result of the LPeg parser.
1814    local function parse(kv_string, opts)
1815      opts = normalize_opts(opts)
1816
```

```lua
    local function log_result(caption, result)
      utils.log
        .debug('%s: \n%s', caption, visualizers.stringify(result))
    end

    if kv_string == nil then
      return {}, {}, {}
    end

    if opts.debug then
      utils.log.set('debug')
    end

    utils.log.debug('kv_string: "%s"', kv_string)

    if type(opts.hooks.kv_string) == 'function' then
      kv_string = opts.hooks.kv_string(kv_string)
    end

    local result = generate_parser('list', opts):match(kv_string)
    local raw = utils.clone_table(result)

    log_result('result after Lpeg Parsing', result)

    local function apply_hook(name)
      if type(opts.hooks[name]) == 'function' then
        if name:match('^keys') then
          result = utils.visit_tree(result, opts.hooks[name])
        else
          opts.hooks[name](result)
        end

        if opts.debug then
          print('After the execution of the hook: ' .. name)
          visualizers.debug(result)
        end
      end
    end

    local function apply_hooks(at)
      if at ~= nil then
        at = '_' .. at
      else
        at = ''
      end
      apply_hook('keys' .. at)
      apply_hook('result' .. at)
    end

    apply_hooks('before_opts')

    log_result('after hooks before_opts', result)

    ---
    ---Normalize the result table of the LPeg parser. This normalization
    ---tasks are performed on the raw input table coming directly from
    ---the PEG parser:
    --
    ---@param result table # The raw input table coming directly from the PEG parser
    ---@param opts table # Some options.
    local function apply_opts(result, opts)
      local callbacks = {
```

```
      unpack = function(key, value)
        if type(value) == 'table' and utils.get_array_size(value) == 1 and
          utils.get_table_size(value) == 1 and type(value[1]) ~=
          'table' then
          return key, value[1]
        end
        return key, value
      end,

      process_naked = function(key, value)
        if type(key) == 'number' and type(value) == 'string' then
          return value, opts.default
        end
        return key, value
      end,

      format_key = function(key, value)
        if type(key) == 'string' then
          for _, style in ipairs(opts.format_keys) do
            if style == 'lower' then
              key = key:lower()
            elseif style == 'snake' then
              key = key:gsub('[^%w]+', '_')
            elseif style == 'upper' then
              key = key:upper()
            else
              throw_error('E017', {
                unknown = style,
                styles = { 'lower', 'snake', 'upper' },
              })
            end
          end
        end
        return key, value
      end,

      apply_invert_flag = function(key, value)
        if type(key) == 'string' and key:find(opts.invert_flag) then
          return key:gsub(opts.invert_flag, ''), not value
        end
        return key, value
      end,
    }

    if opts.unpack then
      result = utils.visit_tree(result, callbacks.unpack)
    end

    if not opts.naked_as_value and opts.defs == false then
      result = utils.visit_tree(result, callbacks.process_naked)
    end

    if opts.format_keys then
      if type(opts.format_keys) ~= 'table' then
        throw_error('E018', { data_type = type(opts.format_keys) })
      end
      result = utils.visit_tree(result, callbacks.format_key)
    end

    if opts.invert_flag then
      result = utils.visit_tree(result, callbacks.apply_invert_flag)
    end
```

```
1941
1942        return result
1943      end
1944      result = apply_opts(result, opts)
1945
1946      log_result('after apply opts', result)
1947
1948      ---All unknown keys are stored in this table
1949      local unknown = nil
1950      if type(opts.defs) == 'table' then
1951        apply_hooks('before_defs')
1952        result, unknown = apply_definitions(opts.defs, opts, result, {},
1953          {}, {}, utils.clone_table(result))
1954      end
1955
1956      log_result('after apply_definitions', result)
1957
1958      apply_hooks()
1959
1960      if opts.defaults ~= nil and type(opts.defaults) == 'table' then
1961        utils.merge_tables(result, opts.defaults, false)
1962      end
1963
1964      log_result('End result', result)
1965
1966      if opts.accumulated_result ~= nil and type(opts.accumulated_result) ==
1967        'table' then
1968        utils.merge_tables(opts.accumulated_result, result, true)
1969      end
1970
1971      ---no_error
1972      if not opts.no_error and type(unknown) == 'table' and
1973        utils.get_table_size(unknown) > 0 then
1974        throw_error('E019', { unknown = visualizers.render(unknown) })
1975      end
1976      return result, unknown, raw
1977    end
1978
1979    ---
1980    ---A table to store parsed key-value results.
1981    local result_store = {}
1982
1983    return {
1984      new = main,
1985
1986      version = { 0, 13, 0 },
1987
1988      ---@see parse
1989      parse = parse,
1990
1991      ---
1992      ---@param defs DefinitionCollection
1993      ---@param opts? OptionCollection
1994      define = function(defs, opts)
1995        return function(kv_string, inner_opts)
1996          local options
1997
1998          if inner_opts ~= nil and opts ~= nil then
1999            options = utils.merge_tables(opts, inner_opts)
2000          elseif inner_opts ~= nil then
2001            options = inner_opts
2002          elseif opts ~= nil then
```

```
2003            options = opts
2004          end
2005
2006          if options == nil then
2007            options = {}
2008          end
2009
2010          options.defs = defs
2011
2012          return parse(kv_string, options)
2013        end
2014      end,
2015
2016      ---@see default_opts
2017      opts = default_opts,
2018
2019      error_messages = error_messages,
2020
2021      ---@see visualizers.render
2022      render = visualizers.render,
2023
2024      ---@see visualizers.stringify
2025      stringify = visualizers.stringify,
2026
2027      ---@see visualizers.debug
2028      debug = visualizers.debug,
2029
2030      ---
2031      ---Save a result (a
2032      ---table from a previous run of `parse`) under an identifier.
2033      ---Therefore, it is not necessary to pollute the global namespace to
2034      ---store results for the later usage.
2035      ---
2036      ---@param identifier string # The identifier under which the result is saved.
2037      ---
2038      ---@param result table|any # A result to be stored and that was created by the
         ↪  key-value parser.
2039      save = function(identifier, result)
2040        result_store[identifier] = result
2041      end,
2042
2043      ---
2044      ---The function `get(identifier): table` retrieves a saved result
2045      ---from the result store.
2046      ---
2047      ---@param identifier string # The identifier under which the result was saved.
2048      ---
2049      ---@return table|any
2050      get = function(identifier)
2051        ---if result_store[identifier] == nil then
2052        ---  throw_error('No stored result was found for the identifier \'' ..
           ↪  identifier .. '\'')
2053        ---end
2054        return result_store[identifier]
2055      end,
2056
2057      is = is,
2058
2059      utils = utils,
2060
2061      ---
2062      ---Exported but intentionally undocumented functions
```

```
2063        ---
2064
2065        namespace = utils.clone_table(namespace),
2066
2067        ---
2068        ---This function is used in the documentation.
2069        ---
2070        ---@param from string # A key in the namespace table, either `opts`, `hook` or
            ↪  `attrs`.
2071        print_names = function(from)
2072          local names = utils.get_table_keys(namespace[from])
2073          tex.print(table.concat(names, ', '))
2074        end,
2075
2076        print_default = function(from, name)
2077          tex.print(tostring(namespace[from][name]))
2078        end,
2079
2080        print_error_messages = function()
2081          local msgs = namespace.error_messages
2082          local keys = utils.get_table_keys(namespace.error_messages)
2083          for _, key in ipairs(keys) do
2084            local msg = msgs[key]
2085            ---@type string
2086            local msg_text
2087            if type(msg) == 'table' then
2088              msg_text = msg[1]
2089            else
2090              msg_text = msg
2091            end
2092            utils.tex_printf('\\item[\\texttt{%s}]: \\texttt{%s}', key,
2093              msg_text)
2094          end
2095        end,
2096
2097        ---
2098        ---@param exported_table table
2099        depublish_functions = function(exported_table)
2100          local function warn_global_import()
2101            throw_error('E023')
2102          end
2103
2104          exported_table.parse = warn_global_import
2105          exported_table.define = warn_global_import
2106          exported_table.save = warn_global_import
2107          exported_table.get = warn_global_import
2108        end,
2109      }
2110
2111    end
2112
2113    return main
```

8.2 luakeys.tex

```
%% luakeys.tex
%% Copyright 2021-2023 Josef Friedrich
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3c
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3c or later is part of all distributions of LaTeX
% version 2008/05/04 or later.
%
% This work has the LPPL maintenance status `maintained'.
%
% The Current Maintainer of this work is Josef Friedrich.
%
% This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
% luakeys-debug.sty and luakeys-debug.tex.

\directlua{
  if luakeys == nil then
    luakeys = require('luakeys')()
    luakeys.depublish_functions(luakeys)
  end
}
```

8.3 luakeys.sty

```
%% luakeys.sty
%% Copyright 2021-2023 Josef Friedrich
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3c
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3c or later is part of all distributions of LaTeX
% version 2008/05/04 or later.
%
% This work has the LPPL maintenance status `maintained'.
%
% The Current Maintainer of this work is Josef Friedrich.
%
% This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
% luakeys-debug.sty and luakeys-debug.tex.

\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{luakeys}[2023/01/13 v0.13.0 Parsing key-value options using Lua.]
\directlua{
  if luakeys == nil then
    luakeys = require('luakeys')()
    luakeys.depublish_functions(luakeys)
  end
}

\def\LuakeysGetPackageOptions{\luaescapestring{\@ptionlist{\@currname.\@currext}}}

\def\LuakeysGetClassOptions{\luaescapestring{\@raw@classoptionslist}}
```

8.4 luakeys-debug.tex

```
%% luakeys-debug.tex
%% Copyright 2021-2023 Josef Friedrich
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3c
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3c or later is part of all distributions of LaTeX
% version 2008/05/04 or later.
%
% This work has the LPPL maintenance status `maintained'.
%
% The Current Maintainer of this work is Josef Friedrich.
%
% This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
% luakeys-debug.sty and luakeys-debug.tex.

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

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

8.5 luakeys-debug.sty

```
%% luakeys-debug.sty
%% Copyright 2021-2023 Josef Friedrich
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3c
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3c or later is part of all distributions of LaTeX
% version 2008/05/04 or later.
%
% This work has the LPPL maintenance status `maintained'.
%
% The Current Maintainer of this work is Josef Friedrich.
%
% This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
% luakeys-debug.sty and luakeys-debug.tex.

\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{luakeys-debug}[2023/01/13 v0.13.0 Debug package for luakeys.]

\input luakeys-debug.tex
```