The luakeys package

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

v0.17.0 from 2025/07/19

```
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
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.

```
\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:

```
\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`

```lua
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 `new()` function (3.6) 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.

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

You can also combine several types of formatting.

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

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

3.3.10 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`.

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

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

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

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

```
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.

```
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 definies a new `parse` function. The `define` function returns a `parse` function (see 3.1). This created `parse` function is configured with the specified key-value defintions.

The name of a key can be specified in three ways:

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

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

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

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

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

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

```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.

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

3.5.2 Attribute "always_present"

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

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

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

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

3.5.3 Attribute "choices"

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

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

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

```
  parse('key=unknown')
  -- error message:
  --- '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.

```
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$' },
})
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$"!'
```

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:

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

The following example demonstrates the `input` parameter:

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

The following example demonstrates the `result` parameter:

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

```
    end,
  },
})
local result = parse('key=1') -- { key = 1, additional_key = true }
```

The following example demonstrates the `unknown` parameter:

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

```
    parse('key=1') -- throws error message: '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 "new()"

Calling the function `luakeys.new()` of the global table `luakeys` has the same effect as this expression: `require('luakeys')()`. The Lua module from `luakeys` exports exactly one function, namely the `new()` function.

3.7 Function "render(value, config): string"

The `render(value)` function renders or serialises a Lua value of any kind into a string. This function can be used to reverse the function `parse(kv_string)`. The keys of the resulting serialized table are sorted alpabetically.

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

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)
```

3.7.1 High level configuration options

`style` Render the input as a `'lua'` table or in the `'tex'` style, default `'tex'`.

`inline` Render the input on one line without line breaks, default `true`.

3.7.2 Low level configuration options

`line_break` The character for a line break, for example, use `\n` for terminal output or `\par` for TeX rendering, default `''`.

`begin_table` The starting delimiter for a table, default `{`.

`end_table` The final delimiter for a table, default `}`.

`table_delimiters_first_depth` Whether table delimiters of the 1st level should be displayed. Instead of `'{key1,key2={value2}}'` render `'key1,key2={value2}'`, default 'false'.

`indent` Characters used for indentation, default `''`.

`begin_key` The starting delimiter for a key, default `'['`.

`end_key` The final delimiter for a key, default `']'`.

`assignment` The symbol for the assignment operator, default `'='`.

`separator` The separator for the individual table elements, default `','`.

`separator_last` Append a separator after the last element, default `false`.

`quotation` The symbol that delimits a string, default `"'"`.

`format_key` A function that formats the key.

`format_value` A function that formats the value.

3.8 Function "debug(result, config): void"

The function `debug(result, config)` pretty prints a Lua value 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. The configuration for the embedded render function can be specified as the second argument (see 3.7).

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

The output should look like this:

```
{
  level1 = {
    level2 = {
      key = value
    }
  }
}
```

3.9 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.

3.10 Function "get(identifier): result"

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

3.11 Class "`DefinitionManager()`"

The `DefinitionManager` class allows you to store key-value definitions in an object. This class provides a `parse` method that is configured with these definitions. New subsets of definitions can be formed based on the saved definitions using the `include` and `exclude` methods.

```
local DefinitionManager = luakeys.DefinitionManager

local manager = DefinitionManager({
  key1 = { default = 1 },
  key2 = { default = 2 },
  key3 = { default = 3 },
})

local def = manager:get('key1')
luakeys.debug(def) -- { default = 1 }

local defs1 = manager:include({ 'key2' })
luakeys.debug(defs1) -- { key2 = { default = 2 } }

local defs2 = manager:exclude({ 'key2' })
luakeys.debug(defs2) -- { key1 = { default = 1 }, key3 = { default = 3 } }

manager:parse('key3', { 'key3' }) -- { key3 = 3 }
manager:parse('new3', { key3 = 'new3' }) -- { new3 = 3 }
--manager:parse('key1', { 'key3' }) -- 'Unknown keys: "key1,"'
```

3.12 Field "`.defs`"

The `defs` field can be used to access the collection of key-value pair definitions on an instance.

```
  local manager = DefinitionManager({
    key1 = { default = 1 },
    key2 = { default = 2 },
    key3 = { default = 3 },
  })
  manager.defs.key1.default = 2
  assert.are.same(manager.defs, {
    key1 = { default = 2 },
    key2 = { default = 2 },
    key3 = { default = 3 },
  })
```

3.13 Method "`:set(key, def)`"

The `DefinitionManager:set(key, def)` method adds a key-value pair definition under a specified key name to the definition collection.

```
  manager:set('key4', { default = 4 })
  assert.is.equal(manager:get('key4').default, 4)
```

3.14 Method "`:get(key): Definition`"

The `DefinitionManager:get(key)` method retrieves a key-value pair definition based on its key name.

```
  assert.is.equal(manager:get('key3').default, 3)
```

3.15 Method ":key_names(): string[]"

The `DefinitionManager:key_names()` method returns all key names of the corresponding definitions as an array.

```
assert.is.same(manager:key_names(), {
  'key1',
  'key2',
  'key3' })
```

3.16 Method ":include(key_spec, clone): Defs"

The `DefinitionManager:include(key_spec, clone)` method includes a subset of the key-value definitions in the return collection of definitions or, if `key_spec` is not specified, all definitions.

```
-- argument: clone = nil
local defs1 = manager:include({ 'key3' })
assert.is.equal(defs1.key3.default, 3)
assert.is.equal(defs1.key3, manager.defs.key3)

-- argument: clone = true
local defs2 = manager:include({ 'key3' }, true)
assert.is_not.equal(defs2.key3, manager.defs.key3)

-- argument: key_spec = nil -> all definitions are returned'
local defs3 = manager:include()
assert.are.same(defs3, {
  key1 = { default = 1 },
  key2 = { default = 2 },
  key3 = { default = 3 },
})
```

3.17 Method ":exclude(key_spec, clone): Defs"

This method excludes a subset of the key-value definitions or, if `key_spec` is not specified, returns all definitions.

```
-- argument: clone = nil
local defs1 = manager:exclude({ 'key3' })
assert.is.equal(defs1.key1.default, 1)
assert.is.equal(defs1.key2.default, 2)
assert.is.equal(defs1.key3, nil)
assert.is.equal(defs1.key1, manager.defs.key1)

-- argument: clone = true
local defs2 = manager:exclude({ 'key3' }, true)
assert.is_not.equal(defs2.key1, manager.defs.key1)

-- argument: key_spec = nil -> all definitions are returned'
local defs3 = manager:exclude()
assert.are.same(defs3, {
  key1 = { default = 1 },
  key2 = { default = 2 },
  key3 = { default = 3 },
})
```

3.18 Method ":clone(opts): DefinitionManager"

This method creates a new instance of the `DefinitionManager` class and adds a deep copy of the key-value pair definitions to it. An options table can be used to exclude or include key-value pair definitions. If no option or `nil` is passed, all definitions are cloned.

```
local manager = DefinitionManager({
  key1 = { default = 1 },
  key2 = { default = 2 },
  key3 = { default = 3 },
})

local clone1 = manager:clone()
assert.is_not.equal(clone1, manager)
assert.are.same(clone1:key_names(), { 'key1', 'key2', 'key3' })

-- option include
local clone2 = manager:clone({ include = { 'key1' } })
assert.are.same(clone2:key_names(), { 'key1' })

-- option exclude
local clone3 = manager:clone({ exclude = { 'key1' } })
assert.are.same(clone3:key_names(), { 'key2', 'key3' })
```

3.19 Method ":define(key_selection): parse"

The `DefinitionManager:define(key_selection)` method creates a new `parse` function. This created `parse` function is configured with key-value defintions of this instance or a subset, if a key selection is specified. This method is related to the `define` function (see 3.4), but with the difference that the `define` method is already configured with the key-value pair definitions of the parent `DefintionManager` class.

```
local parse = manager:define({
  key1 = 'new1'
})

local result = parse('new1')
assert.are.same(result, {
  new1 = 1 }
)

-- exception
assert.has_error(function()
  parse('key1')
end, 'luakeys error [E019]: Unknown keys: "key1"')
```

3.20 Method ":parse(kv_string, key_selection): result, unknown, raw"

The `DefinitionManager:parse(kv_string, key_selection)` method translates a key-value string in LaTeX/TeX style into a Lua table using all definitions of this manager or a subset of the definitions. This method is related to the `parse` function (see 3.1), but with the difference that the `parse` method is already configured with the key-value pair definitions of the parent `DefintionManager` class.

```
-- key_selection=nil: use all defintions
local result = manager:parse('key1,key2,key3')
```

```
  assert.are.same(result, {
    key1 = 1, key2 = 2, key3 = 3 }
  )

  -- key_selection=key3
  local result = manager:parse('key3', { 'key3' })
  assert.is.equal(result.key3, 3)

  -- rename key
  local result = manager:parse('new3', { key3 = 'new3' })
  assert.is.equal(result.new3, 3)

  -- exception
  assert.has_error(function()
    manager:parse('key1', { 'key3' })
  end, 'luakeys error [E019]: Unknown keys: "key1"')
```

3.21 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.

3.21.1 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)
```

3.21.2 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)
```

3.21.3 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)
```

3.21.4 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)
```

3.21.5 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)
```

3.21.6 Function "is.list(value): boolean"

Check if the given value is a list or an array. As we all know, there is no such thing as a list or array in Lua. A list is nothing more than a table that uses an ascending sequence of integers as its keys.

```
-- 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)
```

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

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

3.22 Table "utils"

The `utils` table bundles some auxiliary functions.

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

---table
```

```lua
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

---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
```

3.22.1 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.

```lua
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.

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

3.23 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
```

3.24 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:

```
{
  one = true,
  three = true,
  two = true
}
```

6.1 For plain TeX: luakeys-debug.tex

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

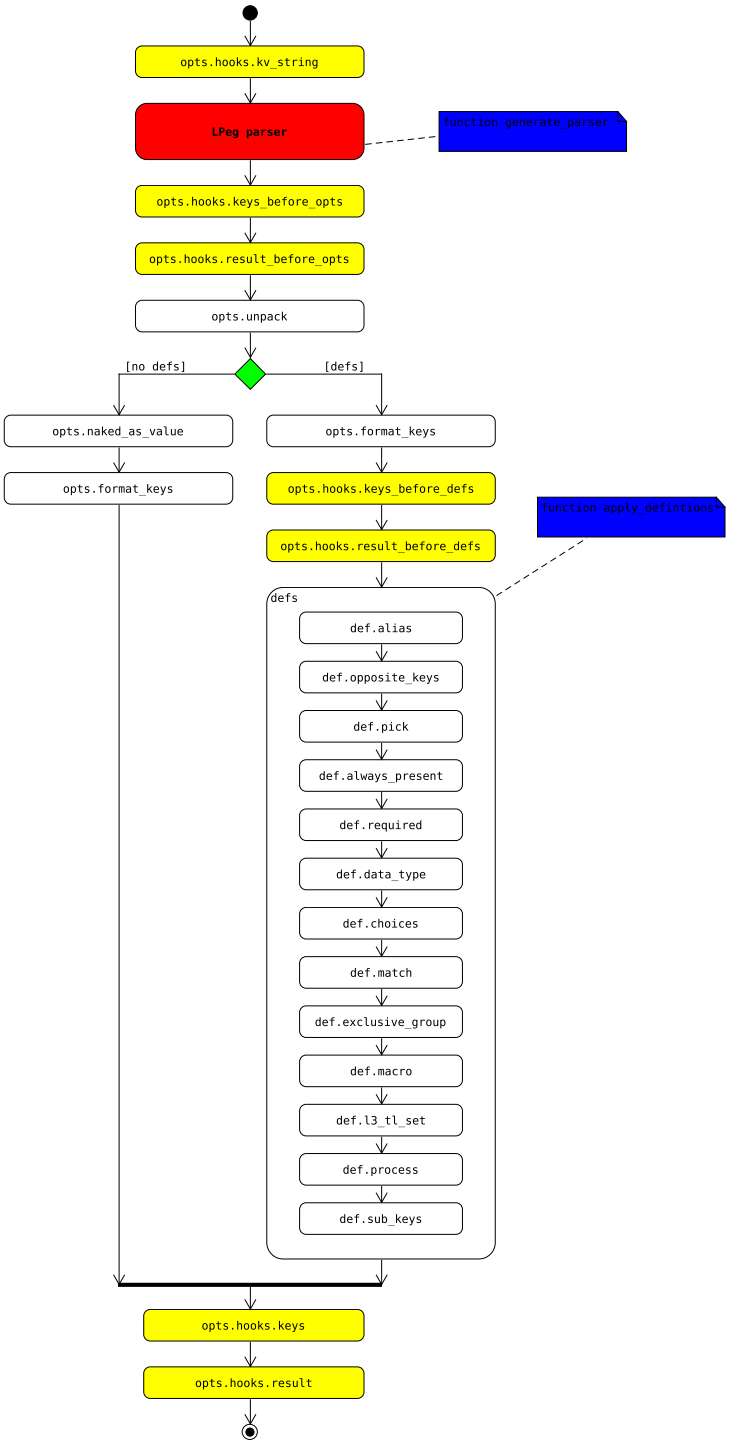
```
\input luakeys-debug.tex
\luakeysdebug{one,two,three}
\bye
```

6.2 For LaTeX: luakeys-debug.sty

An example of how to use the command in LaTeX:

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

7 Activity diagramm of the parsing process



8 Implementation

The source code is hosted on Github. The following links will take you to the individual files:

- luakeys.lua
- luakeys.tex
- luakeys.sty
- luakeys-debug.tex
- luakeys-debug.sty