The luakeys package

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

v0.6 from 2022/06/09

```
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 / LuaTEXpackage that can parse key-value options like the TEX packages keyval, kvsetkeys, kvoptions, xkeyval, pgfkeys etc. `luakeys`, however, accomplishes this task by using the Lua language and doesn't rely on TEX. Therefore this package can only be used with the TEX engine LuaTEX. Since `luakeys` uses LPeg, the parsing mechanism should be pretty robust.

The TUGboat article "Implementing key–value input: An introduction" (Volume 30 (2009), No. 1) by *Joseph Wright* and *Christian Feuersänger* gives a good overview of the available key-value packages.

This package would not be possible without the article "Parsing complex data formats in LuaTEX with LPEG" (Volume 40 (2019), No. 2).

2 How the package is loaded

2.1 Using the Lua module `luakeys.lua`

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

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

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

2.2 Using the LuaLATEX wrapper `luakeys.sty`

For example, the MiKTEX package manager downloads packages only when needed. It has been reported that this automatic download only works with this wrapper files. Probably MiKTEX is searching for an occurrence of the LATEX macro "`\usepackage {luakeys}`".

The supplied LuaLATEX file is quite small:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{luakeys}
\directlua{luakeys = require('luakeys')}
```

It loads the Lua module into the global variable `luakeys`.

```
\documentclass{article}
\usepackage{luakeys}
```

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

2.3 Using the plain LuaTEX wrapper `luakeys.tex`

Even smaller is the file `luakeys.tex`. It consists of only one line:

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

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

```
\input luakeys.tex

\directlua{
  local keys = luakeys.parse('one,two,three', { naked_as_value = true })
  tex.print(keys[1])
  tex.print(keys[2])
  tex.print(keys[3])
} % one two three
\bye
```

3 Lua interface / API

To learn more about the individual functions (local functions), please read the source code documentation, which was created with LDoc. The Lua module exports this functions and tables:

```
local luakeys = require('luakeys')
local opts = luakeys.opts
local stringify = luakeys.stringify
local define = luakeys.define
local parse = luakeys.parse
local render = luakeys.render
local debug = luakeys.debug
local save = luakeys.save
local get = luakeys.get
local is = luakeys.is
```

3.1 Lua indentifier names

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

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

These unabbreviated variable names are commonly used.

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

3.2 Function `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.

```
\newcommand{\mykeyvalcmd}[2][]{
  \directlua{
    result = luakeys.parse('#1')
    luakeys.debug(result)
  }
  #2
}
\mykeyvalcmd[one=1]{test}
```

In plain TEX:

```
\def\mykeyvalcommand#1{
  \directlua{
    result = luakeys.parse('#1')
    luakeys.debug(result)
  }
}
\mykeyvalcmd{one=1}
```

3.3 Options to configure the `parse` function

The `parse` function can be called with an options table. This options are supported:

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

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

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

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

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

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

  -- 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,
  },

  -- 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,

  -- { key = { 'value' } } -> { key = 'value' }
  unpack = false,
}
local result = luakeys.parse('key', 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 }
```

3.3.1 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.

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

If you want to convert a scale point into a unit string you can use the module lualibs-util-dim.lua.

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

3.3.2 Option "debug"

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

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

3.3.3 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.

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

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

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

3.3.4 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' }
```

3.3.5 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 }
```

3.3.6 Option "format_keys"

lower

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

snake

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

upper

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

3.3.7 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 3.5.11)
7. `keys = function(key, value, depth, current, result): key, value`
8. `result = function(result): void`

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

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

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

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

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

```lua
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.

3.3.8 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' }
```

3.3.9 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
```

3.3.10 Option "unpack"

With the help of the option `unpack`, all tables that consist of only one 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 } }
```

3.4 Function `define(defs, opts): parse`

The `define` function returns a `parse` function (see 3.2). The name of a key can be specified in three ways:

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

```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 = { some_def = 'etc' } } } },
  },
}, { 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. The code example below lists all the attributes that can be used to define key-value pairs.

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

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

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

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

    default = true,

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

    -- > \MacroName
```

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

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

    -- The name of the key, can be omitted
    name = 'key',
    opposite_keys = { [true] = 'show', [false] = 'hide' },
    process = function(value, input, result, unknown)
      return value
    end,
    required = true,
    sub_keys = { key_level_2 = { } },
  }
}
```

3.5.1 Attribute "alias"

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

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

multiple aliases by a list of strings.

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

3.5.2 Attribute "always_present"

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

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

If the attribute `always_present` is set to true, the key is always included in the result. If no default value is 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:
--- 'The value "unknown" does not exist in the choices: one, two, three!'
```

3.5.4 Attribute "data_type"

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

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

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

3.5.5 Attribute "default"

Use the `default` attribute to provide a default value for each naked key individually. With the global `default` attribute (3.3.3) a default value can be specified for all naked keys.

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

3.5.6 Attribute "exclusive_group"

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

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

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

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

3.5.7 Attribute “opposite_keys”

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

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

If the key `hide` is parsed, then `false`.

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

3.5.8 Attribute “macro”

The attribute `macro` stores the value in a TEX macro.

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

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

3.5.9 Attribute “match”

The value of the key is passed to the Lua function `string.match(value, match)` ((`http://www.lua.org/manual/5.3/manual.html#pdf-string.match`)). 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:
-- '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.

```
local parse2 = luakeys.define({ year = { match = '%d%d%d%d' } })
local result2 = parse2('year=1978') -- { year = '1978' }
```

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

```
local result3 = parse2('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 parse2 = luakeys.define({
  { name = 'one', default = 1 },
  { name = 'two', default = 2 },
})
local result2 = parse2('one,two') -- { one = 1, two = 2 }
```

3.5.11 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 parse2 = 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 result2 = parse2('key,one=1,two=2') -- { key = 3 }
```

The following example demonstrates the `result` parameter:

```
local parse3 = luakeys.define({
  key = {
    process = function(value, input, result, unknown)
      result.additional_key = true
      return value
    end,
  },
})
local result3 = parse3('key=1') -- { key = 1, additional_key = true }
```

The following example demonstrates the `unknown` parameter:

```
local parse4 = luakeys.define({
  key = {
    process = function(value, input, result, unknown)
      unknown.unknown_key = true
      return value
    end,
  },
})
```

```
    parse4('key=1') -- throws error message: 'Unknown keys: unknown_key=true,'
```

3.5.12 Attribute "required"

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

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

A recursive example:

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

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

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

3.5.13 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`

3.10.1 Function `is.boolean(value): boolean`

```
-- true
equal(luakeys.is.boolean('true'), true)
equal(luakeys.is.boolean('True'), true)
equal(luakeys.is.boolean('TRUE'), true)
equal(luakeys.is.boolean('false'), true)
equal(luakeys.is.boolean('False'), true)
equal(luakeys.is.boolean('FALSE'), true)
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)
```

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)
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)
equal(luakeys.is.number(1.1), true)
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)
```

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`'
| '`nc`' | '`NC`'
| '`nd`' | '`ND`'
| '`pc`' | '`PC`'
| '`pt`' | '`PT`'
| '`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{
  num1 = 4,
  num2 = -4,
  num3 = 0.4
}
```

```
{
  ['num1'] = 4,
  ['num2'] = -4,
  ['num3'] = 0.4
}
```

4.3.3 dimension

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

```
\luakeysdebug{
  bp = 1bp,
  cc = 1cc,
  cm = 1cm,
  dd = 1dd,
  em = 1em,
  ex = 1ex,
  in = 1in,
  mm = 1mm,
  nc = 1nc,
  nd = 1nd,
  pc = 1pc,
  pt = 1pt,
  sp = 1sp,
}
```

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

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`

```
require('busted.runner')()
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

```
\directlua{luakeys.parse('\@classoptionslist')}
```

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

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

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

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

6 Debug packages

Two small debug packages are included in `luakeys`. One debug package can be used in LaTeX (luakeys-debug.sty) and one can be used in plain TeX (luakeys-debug.tex). Both packages provide only one command: `\luakeysdebug{kv-string}`

```
\luakeysdebug{one,two,three}
```

Then the following output should appear in the document:

```
{
  ['two'] = true,
  ['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 Implementation

7.1 luakeys.lua

```lua
-- luakeys.lua
-- Copyright 2021-2022 Josef Friedrich
--
-- This work may be distributed and/or modified under the
-- conditions of the LaTeX Project Public License, either version 1.3c
-- of this license or (at your option) any later version.
-- The latest version of this license is in
--   http://www.latex-project.org/lppl.txt
-- and version 1.3c or later is part of all distributions of LaTeX
-- version 2008/05/04 or later.
--
-- This work has the LPPL maintenance status `maintained'.
--
-- The Current Maintainer of this work is Josef Friedrich.
--
-- This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
-- luakeys-debug.sty and luakeys-debug.tex.
--- A key-value parser written with Lpeg.
--
-- @module luakeys
local lpeg = require('lpeg')

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

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

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

  --- Get the size of a table `{ one = 'one', 'two', 'three' }` = 3.
  --
  -- @tparam table value A table or any input.
```

```lua
    --
    -- @treturn number The size of the array like table. 0 if the input is
    -- no table or the table is empty.
    get_table_size = function(value)
      local count = 0
      if type(value) == 'table' then
        for _ in pairs(value) do
          count = count + 1
        end
      end
      return count
    end,

    remove_from_array = function(array, element)
      for index, value in pairs(array) do
        if element == value then
          array[index] = nil
          return value
        end
      end
    end,
  }

  --- https://stackoverflow.com/a/1283608/10193818
  local function merge_tables(target, t2)
    for k, v in pairs(t2) do
      if type(v) == 'table' then
        if type(target[k] or false) == 'table' then
          merge_tables(target[k] or {}, t2[k] or {})
        elseif target[k] == nil then
          target[k] = v
        end
      elseif target[k] == nil then
        target[k] = v
      end
    end
    return target
  end

  --- http://lua-users.org/wiki/CopyTable
  local function clone_table(orig)
    local orig_type = type(orig)
    local copy
    if orig_type == 'table' then
      copy = {}
      for orig_key, orig_value in next, orig, nil do
        copy[clone_table(orig_key)] = clone_table(orig_value)
      end
      setmetatable(copy, clone_table(getmetatable(orig)))
    else -- number, string, boolean, etc
      copy = orig
    end
    return copy
  end

  --- This table stores all allowed option keys.
  local all_options = {
    convert_dimensions = false,
    debug = false,
    default = true,
    defaults = false,
    defs = false,
```

```
  format_keys = false,
  hooks = {},
  naked_as_value = false,
  no_error = false,
  postprocess = false,
  preprocess = false,
  unpack = true,
}

--- The default options.
local default_options = clone_table(all_options)

local function throw_error(message)
  if type(tex.error) == 'function' then
    tex.error(message)
  else
    error(message)
  end
end

local l3_code_cctab = 10

--- Convert back to strings
-- @section

--- The function `render(tbl)` reverses the function
--  `parse(kv_string)`. It takes a Lua table and converts this table
--  into a key-value string. The resulting string usually has a
--  different order as the input table. In Lua only tables with
--  1-based consecutive integer keys (a.k.a. array tables) can be
--  parsed in order.
--
-- @tparam table result A table to be converted into a key-value string.
--
-- @treturn string A key-value string that can be passed to a TeX
-- macro.
local function render(result)
  local function render_inner(result)
    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)
```

```lua
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.
--
-- @tparam table result A table to stringify.
--
-- @tparam boolean for_tex Stringify the table into a text string that
--   can be embeded inside a TeX document via tex.print(). Curly braces
--   and whites spaces are escaped.
--
-- https://stackoverflow.com/a/54593224/10193818
local function stringify(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

    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
```

```
245               add(depth, key .. ' = ' .. start_bracket .. end_bracket .. ',')
246             end
247           else
248             if (type(value) == 'string') then
249               value = string.format('\'%s\'', value)
250             else
251               value = tostring(value)
252             end
253
254             add(depth, key .. ' = ' .. value .. ',')
255           end
256         end
257       end
258
259       return table.concat(output, line_break)
260     end
261
262     return
263       start_bracket .. line_break .. stringify_inner(result, 1) .. line_break ..
264         end_bracket
265   end
266
267   --- The function `debug(tbl)` pretty prints a Lua table to standard
268   --   output (stdout). It is a utility function that can be used to
269   --   debug and inspect the resulting Lua table of the function
270   --   `parse`. You have to compile your TeX document in a console to
271   --   see the terminal output.
272   --
273   -- @tparam table result A table to be printed to standard output for
274   -- debugging purposes.
275   local function debug(result)
276     print('\n' .. stringify(result, false))
277   end
278
279   --- Parser / Lpeg related
280   -- @section
281
282   --- Generate the PEG parser using Lpeg.
283   --
284   -- Explanations of some LPeg notation forms:
285   --
286   -- * `patt ^ 0` = `expression *`
287   -- * `patt ^ 1` = `expression +`
288   -- * `patt ^ -1` = `expression ?`
289   -- * `patt1 * patt2` = `expression1 expression2`: Sequence
290   -- * `patt1 + patt2` = `expression1 / expression2`: Ordered choice
291   --
292   -- * [TUGboat article: Parsing complex data formats in LuaTEX with
      ↪  LPEG](https://tug.org/TUGboat/tb40-2/tb125menke-Patterndf)
293   --
294   -- @treturn userdata The parser.
295   local function generate_parser(initial_rule, convert_dimensions)
296     if convert_dimensions == nil then
297       convert_dimensions = false
298     end
299
300     local Variable = lpeg.V
301     local Pattern = lpeg.P
302     local Set = lpeg.S
303     local Range = lpeg.R
304     local CaptureGroup = lpeg.Cg
305     local CaptureFolding = lpeg.Cf
```

```
306     local CaptureTable = lpeg.Ct
307     local CaptureConstant = lpeg.Cc
308     local CaptureSimple = lpeg.C
309
310     -- Optional whitespace
311     local white_space = Set(' \t\n\r')
312
313     --- Match literal string surrounded by whitespace
314     local ws = function(match)
315       return white_space ^ 0 * Pattern(match) * white_space ^ 0
316     end
317
318     local capture_dimension = function(input)
319       if convert_dimensions then
320         return tex.sp(input)
321       else
322         return input
323       end
324     end
325
326     --- Add values to a table in two modes:
327     --
328     -- # Key value pair
329     --
330     -- If arg1 and arg2 are not nil, then arg1 is the key and arg2 is the
331     -- value of a new table entry.
332     --
333     -- # Index value
334     --
335     -- If arg2 is nil, then arg1 is the value and is added as an indexed
336     -- (by an integer) value.
337     --
338     -- @tparam table table
339     -- @tparam mixed arg1
340     -- @tparam mixed arg2
341     --
342     -- @treturn table
343     local add_to_table = function(table, arg1, arg2)
344       if arg2 == nil then
345         local index = #table + 1
346         return rawset(table, index, arg1)
347       else
348         return rawset(table, arg1, arg2)
349       end
350     end
351
352     -- LuaFormatter off
353     return Pattern({
354       [1] = initial_rule,
355
356       -- list_item*
357       list = CaptureFolding(
358         CaptureTable('') * Variable('list_item')^0,
359         add_to_table
360       ),
361
362       -- '{' list '}'
363       list_container =
364         ws('{') * Variable('list') * ws('}'),
365
366       -- ( list_container / key_value_pair / value ) ','?
367       list_item =
```

```
368         CaptureGroup(
369           Variable('list_container') +
370           Variable('key_value_pair') +
371           Variable('value')
372         ) * ws(',')^-1,
373
374       -- key '=' (list_container / value)
375       key_value_pair =
376         (Variable('key') * ws('=')) * (Variable('list_container') +
          ↪  Variable('value')),
377
378       -- number / string_quoted / string_unquoted
379       key =
380         Variable('number') +
381         Variable('string_quoted') +
382         Variable('string_unquoted'),
383
384       -- boolean !value / dimension !value / number !value / string_quoted !value /
          ↪  string_unquoted
385       -- !value -> Not-predicate -> * -Variable('value')
386       value =
387         Variable('boolean') * -Variable('value') +
388         Variable('dimension') * -Variable('value') +
389         Variable('number') * -Variable('value')  +
390         Variable('string_quoted') * -Variable('value') +
391         Variable('string_unquoted'),
392
393       -- for is.boolean()
394       boolean_only = Variable('boolean') * -1,
395
396       -- boolean_true / boolean_false
397       boolean =
398         (
399           Variable('boolean_true') * CaptureConstant(true) +
400           Variable('boolean_false') * CaptureConstant(false)
401         ),
402
403       boolean_true =
404         Pattern('true') +
405         Pattern('TRUE') +
406         Pattern('True'),
407
408       boolean_false =
409         Pattern('false') +
410         Pattern('FALSE') +
411         Pattern('False'),
412
413       -- for is.dimension()
414       dimension_only = Variable('dimension') * -1,
415
416       dimension = (
417         Variable('tex_number') * white_space^0 *
418         Variable('unit')
419       ) / capture_dimension,
420
421       -- for is.number()
422       number_only = Variable('number') * -1,
423
424       -- capture number
425       number = Variable('tex_number') / tonumber,
426
427       -- 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
    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('nc') + Pattern('NC') +
      Pattern('nd') + Pattern('ND') +
      Pattern('pc') + Pattern('PC') +
      Pattern('pt') + Pattern('PT') +
      Pattern('sp') + Pattern('SP'),

    -- '"' ('\"' / !'"')* '"'
    string_quoted =
      white_space^0 * Pattern('"') *
      CaptureSimple((Pattern('\\"') + 1 - Pattern('"'))^0) *
      Pattern('"') * 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('{},='))^1
  })
-- LuaFormatter on
end

local function visit_tree(tree, callback_func)
  if type(tree) ~= 'table' then
    throw_error('Parameter "tree" has to be a table, got: ' ..
                  tostring(tree))
  end
  local function visit_tree_recursive(tree,
    current,
    result,
    depth,
    callback_func)
    for key, value in pairs(current) do
      if type(value) == 'table' then
        value = visit_tree_recursive(tree, value, {}, depth + 1, callback_func)
      end

      key, value = callback_func(key, value, depth, current, tree)
```

```
490
491         if key ~= nil and value ~= nil then
492           result[key] = value
493         end
494       end
495       if next(result) ~= nil then
496         return result
497       end
498     end
499
500     local result = visit_tree_recursive(tree, tree, {}, 1, callback_func)
501
502     if result == nil then
503       return {}
504     end
505     return result
506   end
507
508   local is = {
509     boolean = function(value)
510       if value == nil then
511         return false
512       end
513       if type(value) == 'boolean' then
514         return true
515       end
516       local parser = generate_parser('boolean_only', false)
517       local result = parser:match(value)
518       return result ~= nil
519     end,
520
521     dimension = function(value)
522       if value == nil then
523         return false
524       end
525       local parser = generate_parser('dimension_only', false)
526       local result = parser:match(value)
527       return result ~= nil
528     end,
529
530     integer = function(value)
531       local n = tonumber(value)
532       if n == nil then
533         return false
534       end
535       return n == math.floor(n)
536     end,
537
538     number = function(value)
539       if value == nil then
540         return false
541       end
542       if type(value) == 'number' then
543         return true
544       end
545       local parser = generate_parser('number_only', false)
546       local result = parser:match(value)
547       return result ~= nil
548     end,
549
550     string = function(value)
551       return type(value) == 'string'
```

```
552     end,
553   }
554
555   --- Apply the key-value-pair definitions (defs) on an input table in a
556   --- recursive fashion.
557   ---
558   ---@param defs table A table containing all definitions.
559   ---@param opts table The parse options table.
560   ---@param input table The current input table.
561   ---@param output table The current output table.
562   ---@param unknown table Always the root unknown table.
563   ---@param key_path table An array of key names leading to the current
564   ---@param input_root table The root input table
565   ---  input and output table.
566   local function apply_definitions(defs,
567     opts,
568     input,
569     output,
570     unknown,
571     key_path,
572     input_root)
573     local exclusive_groups = {}
574
575     local function add_to_key_path(key_path, key)
576       local new_key_path = {}
577
578       for index, value in ipairs(key_path) do
579         new_key_path[index] = value
580       end
581
582       table.insert(new_key_path, key)
583       return new_key_path
584     end
585
586     local function set_default_value(def)
587       if def.default ~= nil then
588         return def.default
589       elseif opts ~= nil and opts.default ~= nil then
590         return opts.default
591       end
592       return true
593     end
594
595     local function find_value(search_key, def)
596       if input[search_key] ~= nil then
597         local value = input[search_key]
598         input[search_key] = nil
599         return value
600         -- naked keys: values with integer keys
601       elseif utils.remove_from_array(input, search_key) ~= nil then
602         return set_default_value(def)
603       end
604     end
605
606     local apply = {
607       alias = function(value, key, def)
608         if type(def.alias) == 'string' then
609           def.alias = { def.alias }
610         end
611         local alias_value
612         local used_alias_key
613         -- To get an error if the key and an alias is present
```

```
614         if value ~= nil then
615           alias_value = value
616           used_alias_key = key
617         end
618         for _, alias in ipairs(def.alias) do
619           local v = find_value(alias, def)
620           if v ~= nil then
621             if alias_value ~= nil then
622               throw_error(string.format(
623                 'Duplicate aliases "%s" and "%s" for key "%s"!',
624                 used_alias_key, alias, key))
625             end
626             used_alias_key = alias
627             alias_value = v
628           end
629         end
630         if alias_value ~= nil then
631           return alias_value
632         end
633       end,
634
635       always_present = function(value, key, def)
636         if value == nil and def.always_present then
637           return set_default_value(def)
638         end
639       end,
640
641       choices = function(value, key, def)
642         if value == nil then
643           return
644         end
645         if def.choices ~= nil and type(def.choices) == 'table' then
646           local is_in_choices = false
647           for _, choice in ipairs(def.choices) do
648             if value == choice then
649               is_in_choices = true
650             end
651           end
652           if not is_in_choices then
653             throw_error('The value "' .. value ..
654                         '" does not exist in the choices: ' ..
655                         table.concat(def.choices, ', ') .. '!')
656           end
657         end
658       end,
659
660       data_type = function(value, key, def)
661         if value == nil then
662           return
663         end
664         if def.data_type ~= nil then
665           local converted
666           -- boolean
667           if def.data_type == 'boolean' then
668             if value == 0 or value == '' or not value then
669               converted = false
670             else
671               converted = true
672             end
673             -- dimension
674           elseif def.data_type == 'dimension' then
675             if is.dimension(value) then
```

```
676             converted = value
677           end
678           -- integer
679         elseif def.data_type == 'integer' then
680           if is.number(value) then
681             converted = math.floor(tonumber(value))
682           end
683           -- number
684         elseif def.data_type == 'number' then
685           if is.number(value) then
686             converted = tonumber(value)
687           end
688           -- string
689         elseif def.data_type == 'string' then
690           converted = tostring(value)
691         else
692           throw_error('Unknown data type: ' .. def.data_type)
693         end
694         if converted == nil then
695           throw_error('The value "' .. value .. '" of the key "' .. key ..
696                       '" could not be converted into the data type "' ..
697                       def.data_type .. '"!')
698         else
699           return converted
700         end
701       end
702     end,
703
704     exclusive_group = function(value, key, def)
705       if value == nil then
706         return
707       end
708       if def.exclusive_group ~= nil then
709         if exclusive_groups[def.exclusive_group] ~= nil then
710           throw_error('The key "' .. key ..
711                       '" belongs to a mutually exclusive group "' ..
712                       def.exclusive_group .. '" and the key "' ..
713                       exclusive_groups[def.exclusive_group] ..
714                       '" is already present!')
715         else
716           exclusive_groups[def.exclusive_group] = key
717         end
718       end
719     end,
720
721     l3_tl_set = function(value, key, def)
722       if value == nil then
723         return
724       end
725       if def.l3_tl_set ~= nil then
726         tex.print(l3_code_cctab, '\\tl_set:Nn \\g_' .. def.l3_tl_set .. '_tl')
727         tex.print('{' .. value .. '}')
728       end
729     end,
730
731     macro = function(value, key, def)
732       if value == nil then
733         return
734       end
735       if def.macro ~= nil then
736         token.set_macro(def.macro, value, 'global')
737       end
```

```
738       end,
739
740       match = function(value, key, def)
741         if value == nil then
742           return
743         end
744         if def.match ~= nil then
745           if type(def.match) ~= 'string' then
746             throw_error('def.match has to be a string')
747           end
748           local match = string.match(value, def.match)
749           if match == nil then
750             throw_error('The value "' .. value .. '" of the key "' .. key ..
751                         '" does not match "' .. def.match .. '"!')
752           else
753             return match
754           end
755         end
756       end,
757
758       opposite_keys = function(value, key, def)
759         if def.opposite_keys ~= nil then
760           local true_value = def.opposite_keys[true]
761           local false_value = def.opposite_keys[false]
762           if true_value == nil or false_value == nil then
763             throw_error(
764               'Usage opposite_keys = { [true] = "...", [false] = "..." }')
765           end
766           if utils.remove_from_array(input, true_value) ~= nil then
767             return true
768           elseif utils.remove_from_array(input, false_value) ~= nil then
769             return false
770           end
771         end
772       end,
773
774       process = function(value, key, def)
775         if value == nil then
776           return
777         end
778         if def.process ~= nil and type(def.process) == 'function' then
779           return def.process(value, input_root, output, unknown)
780         end
781       end,
782
783       required = function(value, key, def)
784         if def.required ~= nil and def.required and value == nil then
785           throw_error(string.format('Missing required key "%s"!', key))
786         end
787       end,
788
789       sub_keys = function(value, key, def)
790         if def.sub_keys ~= nil then
791           local v
792           -- To get keys defined with always_present
793           if value == nil then
794             v = {}
795           elseif type(value) == 'string' then
796             v = { value }
797           elseif type(value) == 'table' then
798             v = value
799           end
```

```
800           v = apply_definitions(def.sub_keys, opts, v, output[key], unknown,
801             add_to_key_path(key_path, key), input_root)
802           if utils.get_table_size(v) > 0 then
803             return v
804           end
805         end
806       end,
807     }
808
809     --- standalone values are removed.
810     -- For some callbacks and the third return value of parse, we
811     -- need an unchanged raw result from the parse function.
812     input = clone_table(input)
813     if output == nil then
814       output = {}
815     end
816     if unknown == nil then
817       unknown = {}
818     end
819     if key_path == nil then
820       key_path = {}
821     end
822
823     for index, def in pairs(defs) do
824       --- Find key and def
825       local key
826       if type(def) == 'table' and def.name == nil and type(index) == 'string' then
827         key = index
828       elseif type(def) == 'table' and def.name ~= nil then
829         key = def.name
830       elseif type(index) == 'number' and type(def) == 'string' then
831         key = def
832         def = { default = true }
833       end
834
835       if type(def) ~= 'table' then
836         throw_error('Key definition must be a table')
837       end
838
839       if key == nil then
840         throw_error('key name couldn't be detected!')
841       end
842
843       local value = find_value(key, def)
844
845       for _, def_opt in ipairs({
846         'alias',
847         'opposite_keys',
848         'always_present',
849         'required',
850         'data_type',
851         'choices',
852         'match',
853         'exclusive_group',
854         'macro',
855         'l3_tl_set',
856         'process',
857         'sub_keys',
858       }) do
859         if def[def_opt] ~= nil then
860           local tmp_value = apply[def_opt](value, key, def)
861           if tmp_value ~= nil then
```

```
862             value = tmp_value
863           end
864         end
865       end
866
867       output[key] = value
868     end
869
870     if utils.get_table_size(input) > 0 then
871       -- Move to the current unknown table.
872       local current_unknown = unknown
873       for _, key in ipairs(key_path) do
874         if current_unknown[key] == nil then
875           current_unknown[key] = {}
876         end
877         current_unknown = current_unknown[key]
878       end
879
880       -- Copy all unknown key-value-pairs to the current unknown table.
881       for key, value in pairs(input) do
882         current_unknown[key] = value
883       end
884     end
885
886     return output, unknown
887   end
888
889   --- Parse a LaTeX/TeX style key-value string into a Lua table.
890   ---
891   ---@param kv_string string A string in the TeX/LaTeX style key-value format as
      ↪  described above.
892   ---@param opts table A table containing the settings:
893   ---   `convert_dimensions`, `unpack`, `naked_as_value`, `converter`,
894   ---   `debug`, `preprocess`, `postprocess`.
895   --
896   ---@return table result The final result of all individual parsing and normalization
      ↪  steps.
897   ---@return table unknown A table with unknown, undefinied key-value pairs.
898   ---@return table raw The unprocessed, raw result of the LPeg parser.
899   local function parse(kv_string, opts)
900     if kv_string == nil then
901       return {}
902     end
903
904     --- Normalize the parse options.
905     ---
906     --- @param opts table Options in a raw format. The table may be empty or some keys
      ↪  are not set.
907     ---
908     --- @return table
909     local function normalize_opts(opts)
910       if type(opts) ~= 'table' then
911         opts = {}
912       end
913       for key, _ in pairs(opts) do
914         if all_options[key] == nil then
915           throw_error('Unknown parse option: ' .. tostring(key) .. '!')
916         end
917       end
918       local old_opts = opts
919       opts = {}
920       for name, _ in pairs(all_options) do
```

```
921         if old_opts[name] ~= nil then
922           opts[name] = old_opts[name]
923         else
924           opts[name] = default_options[name]
925         end
926       end
927
928       local hooks = {
929         kv_string = true,
930         keys_before_opts = true,
931         result_before_opts = true,
932         keys_before_def = true,
933         result_before_def = true,
934         keys = true,
935         result = true,
936       }
937
938       for hook in pairs(opts.hooks) do
939         if hooks[hook] == nil then
940           throw_error('Unknown hook: ' .. tostring(hook) .. '!')
941         end
942       end
943       return opts
944     end
945     opts = normalize_opts(opts)
946
947     if type(opts.hooks.kv_string) == 'function' then
948       kv_string = opts.hooks.kv_string(kv_string)
949     end
950
951     local result = generate_parser('list', opts.convert_dimensions):match(
952       kv_string)
953     local raw = clone_table(result)
954
955     local function apply_hook(name)
956       if type(opts.hooks[name]) == 'function' then
957         if name:match('^keys') then
958           result = visit_tree(result, opts.hooks[name])
959         else
960           opts.hooks[name](result)
961         end
962
963         if opts.debug then
964           print('After the execution of the hook: ' .. name)
965           debug(result)
966         end
967       end
968     end
969
970     local function apply_hooks(at)
971       if at ~= nil then
972         at = '_' .. at
973       else
974         at = ''
975       end
976       apply_hook('keys' .. at)
977       apply_hook('result' .. at)
978     end
979
980     apply_hooks('before_opts')
981
982     --- Normalize the result table of the LPeg parser. This normalization
```

```lua
    --  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('Unknown style to format keys: ' .. tostring(style) ..
                            ' Allowed styles are: lower, snake, upper')
              end
            end
          end
          return key, value
        end,
      }

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

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

      if opts.format_keys then
        if type(opts.format_keys) ~= 'table' then
          throw_error('The option "format_keys" has to be a table not ' ..
                      type(opts.format_keys))
        end
        result = visit_tree(result, callbacks.format_key)
      end

      return result
    end
    result = apply_opts(result, opts)

    -- All unknown keys are stored in this table
```

```lua
  local unknown = nil
  if type(opts.defs) == 'table' then
    apply_hooks('before_defs')
    result, unknown = apply_definitions(opts.defs, opts, result, {}, {}, {},
      clone_table(result))
  end

  apply_hooks()

  if opts.defaults ~= nil and type(opts.defaults) == 'table' then
    merge_tables(result, opts.defaults)
  end

  if opts.debug then
    debug(result)
  end

  -- no_error
  if not opts.no_error and type(unknown) == 'table' and
    utils.get_table_size(unknown) > 0 then
    throw_error('Unknown keys: ' .. render(unknown))
  end
  return result, unknown, raw
end

--- Store results
-- @section

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

--- Exports
-- @section

local export = {
  --- @see default_options
  opts = default_options,

  --- @see stringify
  stringify = stringify,

  --- @see parse
  parse = parse,

  define = function(defs, opts)
    return function(kv_string, inner_opts)
      local options
      if inner_opts ~= nil then
        options = inner_opts
      elseif opts ~= nil then
        options = opts
      end

      if options == nil then
        options = {}
      end

      options.defs = defs

      return parse(kv_string, options)
    end
  end,
```

```
1107
1108      --- @see render
1109      render = render,
1110
1111      --- @see debug
1112      debug = debug,
1113
1114      --- The function `save(identifier, result): void` saves a result (a
1115      --  table from a previous run of `parse`) under an identifier.
1116      --  Therefore, it is not necessary to pollute the global namespace to
1117      --  store results for the later usage.
1118      --
1119      -- @tparam string identifier The identifier under which the result is
1120      --   saved.
1121      --
1122      -- @tparam table result A result to be stored and that was created by
1123      --   the key-value parser.
1124      save = function(identifier, result)
1125        result_store[identifier] = result
1126      end,
1127
1128      --- The function `get(identifier): table` retrieves a saved result
1129      --  from the result store.
1130      --
1131      -- @tparam string identifier The identifier under which the result was
1132      --   saved.
1133      get = function(identifier)
1134        -- if result_store[identifier] == nil then
1135        --   throw_error('No stored result was found for the identifier \'' ..
          ↪  identifier .. '\'')
1136        -- end
1137        return result_store[identifier]
1138      end,
1139
1140      is = is,
1141    }
1142
1143    return export
```

7.2 luakeys.tex

```
%% luakeys.tex
%% Copyright 2021-2022 Josef Friedrich
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3c
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3c or later is part of all distributions of LaTeX
% version 2008/05/04 or later.
%
% This work has the LPPL maintenance status `maintained'.
%
% The Current Maintainer of this work is Josef Friedrich.
%
% This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
% luakeys-debug.sty and luakeys-debug.tex.

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

7.3 luakeys.sty

```
%% luakeys.sty
%% Copyright 2021-2022 Josef Friedrich
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3c
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3c or later is part of all distributions of LaTeX
% version 2008/05/04 or later.
%
% This work has the LPPL maintenance status `maintained'.
%
% The Current Maintainer of this work is Josef Friedrich.
%
% This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
% luakeys-debug.sty and luakeys-debug.tex.

\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{luakeys}[2022/06/09 v0.6 Parsing key-value options using Lua.]
\directlua{luakeys = require('luakeys')}
```

7.4 luakeys-debug.tex

```
%% luakeys-debug.tex
%% Copyright 2021-2022 Josef Friedrich
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3c
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3c or later is part of all distributions of LaTeX
% version 2008/05/04 or later.
%
% This work has the LPPL maintenance status `maintained'.
%
% The Current Maintainer of this work is Josef Friedrich.
%
% This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
% luakeys-debug.sty and luakeys-debug.tex.

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

% https://tex.stackexchange.com/a/418401/42311
\catcode`\@=11
\long\def\LuaKeysIfNextChar#1#2#3{%
  \let\@tmpa=#1%
  \def\@tmpb{#2}%
  \def\@tmpc{#3}%
  \futurelet\@future\LuaKeysIfNextChar@i%
}%
\def\LuaKeysIfNextChar@i{%
  \ifx\@tmpa\@future%
    \expandafter\@tmpb
  \else
    \expandafter\@tmpc
  \fi
}%
\def\luakeysdebug@parse@options#1{
  \directlua{
    luakeys.save(
      'debug_options',
      luakeys.parse('#1', { format_keys = { 'snake', 'lower' } })
    )
  }
}%
\def\luakeysdebug@output#1{
  {
    \tt
    \parindent=0pt
    \directlua{
      local result = luakeys.parse('\luaescapestring{\unexpanded{#1}}',
      ↪  luakeys.get('debug_options'))
      tex.print(luakeys.stringify(result, true))
      luakeys.debug(result)
    }
  }
}%
\def\luakeysdebug@oarg[#1]#2{%
  \luakeysdebug@parse@options{#1}%
  \luakeysdebug@output{#2}%
}%
```

```
\def\luakeysdebug@marg#1{%
  \luakeysdebug@output{#1}%
}%
\def\luakeysdebug{\LuaKeysIfNextChar[{\luakeysdebug@oarg}{\luakeysdebug@marg}}%
\catcode`\@=12
```

7.5 luakeys-debug.sty

```
%% luakeys-debug.sty
%% Copyright 2021-2022 Josef Friedrich
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3c
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3c or later is part of all distributions of LaTeX
% version 2008/05/04 or later.
%
% This work has the LPPL maintenance status `maintained'.
%
% The Current Maintainer of this work is Josef Friedrich.
%
% This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
% luakeys-debug.sty and luakeys-debug.tex.

\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{luakeys-debug}[2022/06/09 v0.6 Debug package for luakeys.]

\input luakeys-debug.tex
```

Change History