

The luakeys package

Josef Friedrich

josef@friedrich.rocks

github.com/Josef-Friedrich/luakeys

v0.14.0 from 2024/04/09

```
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	4
1.1	Pros of luakeys	4
1.2	Cons of luakeys	4
2	How the package is loaded	4
2.1	Using the Lua module luakeys.lua	4
2.2	Using the Lua ^L TeX wrapper luakeys.sty	5
2.3	Using the plain LuaTeX wrapper luakeys.tex	5
3	Lua interface / API	5
3.1	Function “parse(kv_string, opts): result, unknown, raw”	6
3.2	Options to configure the parse function	7
3.3	Table “opts”	8
3.3.1	Option “accumulated_result”	9
3.3.2	Option “assignment_operator”	9
3.3.3	Option “convert_dimensions”	10
3.3.4	Option “debug”	10
3.3.5	Option “default”	11
3.3.6	Option “defaults”	11
3.3.7	Option “defs”	11
3.3.8	Option “false_aliases”	12
3.3.9	Option “format_keys”	12
3.3.10	Option “group_begin”	13
3.3.11	Option “group_end”	13
3.3.12	Option “invert_flag”	13
3.3.13	Option “hooks”	14
3.3.14	Option “list_separator”	15
3.3.15	Option “naked_as_value”	15
3.3.16	Option “no_error”	15
3.3.17	Option “quotation_begin”	16
3.3.18	Option “quotation_end”	16
3.3.19	Option “true_aliases”	16
3.3.20	Option “unpack”	16
3.4	Function “define(defs, opts): parse”	16
3.5	Attributes to define a key-value pair	17
3.5.1	Attribute “alias”	18
3.5.2	Attribute “always_present”	19
3.5.3	Attribute “choices”	19
3.5.4	Attribute “data_type”	19
3.5.5	Attribute “default”	20
3.5.6	Attribute “description”	20
3.5.7	Attribute “exclusive_group”	20
3.5.8	Attribute “macro”	20
3.5.9	Attribute “match”	21
3.5.10	Attribute “name”	21
3.5.11	Attribute “opposite_keys”	22
3.5.12	Attribute “pick”	22
3.5.13	Attribute “process”	23

3.5.14	Attribute “required”	25
3.5.15	Attribute “sub_keys”	25
3.6	Function “render(result): string”	26
3.7	Function “debug(result): void”	26
3.8	Function “save(identifier, result): void”	26
3.9	Function “get(identifier): result”	27
3.10	Class “DefinitionManager()”	27
3.11	Table “is”	27
3.11.1	Function “is.boolean(value): boolean”	27
3.11.2	Function “is.dimension(value): boolean”	28
3.11.3	Function “is.integer(value): boolean”	28
3.11.4	Function “is.number(value): boolean”	28
3.11.5	Function “is.string(value): boolean”	28
3.11.6	Function “is.list(value): boolean”	29
3.11.7	Function “is.any(value): boolean”	29
3.12	Table “utils”	29
3.12.1	Function “utils.merge_tables(target, source, overwrite): table”	30
3.13	Table “version”	30
3.14	Table “error_messages”	30
4	Syntax of the recognized key-value format	32
4.1	An attempt to put the syntax into words	32
4.2	An (incomplete) attempt to put the syntax into the Extended Backus-Naur Form	32
4.3	Recognized data types	33
4.3.1	boolean	33
4.3.2	number	33
4.3.3	dimension	33
4.3.4	string	34
4.3.5	Naked keys	34
5	Examples	36
5.1	Extend and modify keys of existing macros	36
5.2	Process document class options	37
5.3	Process package options	37
6	Debug packages	39
6.1	For plain T _E X: luakeys-debug.tex	39
6.2	For L ^A T _E X: luakeys-debug.sty	39
7	Activity diagramm of the parsing process	40
8	Implementation	41
8.1	luakeys.lua	41
8.2	luakeys.tex	77
8.3	luakeys.sty	78
8.4	luakeys-debug.tex	79
8.5	luakeys-debug.sty	80

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 Lua^AT_EX wrapper `luakeys.sty`

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

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

```
\input luakeys.tex
\directlua{
  local 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 stringify = luakeys.stringify
local debug = luakeys.debug
local save = luakeys.save
local get = luakeys.get
local is = luakeys.is
local utils = luakeys.utils

```

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

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

These unabbreviated variable names are commonly used.

```

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

```

It is recommended to use luakeys together with the github.com/sumneko/lu-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 $\text{T}_{\text{E}}\text{X}$:

```

\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`, `false_aliases`, `format_keys`, `group_begin`, `group_end`, `hooks`, `invert_flag`, `list_separator`, `naked_as_value`, `no_error`, `quotation_begin`, `quotation_end`, `true_aliases`, `unpack`

```

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

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

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

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

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

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

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

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

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

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

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

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

    -- Visit all key-value pairs recursively.

```

```

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

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

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

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

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

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

invert_flag = '!',

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

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

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

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

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

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

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

```

3.3 Table “opts”

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

```

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

```



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

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

3.3.1 Option “accumulated_result”

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

```
local result = {}

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

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

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

3.3.2 Option “assignment_operator”

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

The code example below demonstrates all six delimiter related options.

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

Delimiter options	Section
<code>assignment_operator</code>	3.3.2
<code>group_begin</code>	3.3.10
<code>group_end</code>	3.3.11
<code>list_separator</code>	3.3.14
<code>quotation_begin</code>	3.3.17
<code>quotation_end</code>	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 defined using defs, see [3.5.13](#))
7. `keys` = function(key, value, depth, current, result): key, value
8. `result` = function(result): void

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

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

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

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

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

```

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

```

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

3.3.14 Option “list_separator”

The option `list_separator` configures the delimiter that separates list items from each other. The default value of this option is `","`. A code example can be found in section [3.3.2](#).

3.3.15 Option “naked_as_value”

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

```

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

```

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

```

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

```

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

3.3.16 Option “no_error”

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

```

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

```

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

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

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

3.3.17 Option “quotation_begin”

The option `quotation_begin` configures the delimiter that marks the beginning of a string. The default value of this option is `''''` (double quotes). A code example can be found in section 3.3.2.

3.3.18 Option “quotation_end”

The option `quotation_end` configures the delimiter that marks the end of a string. The default value of this option is `''''` (double quotes). A code example can be found in section 3.3.2.

3.3.19 Option “true_aliases”

See section 3.3.8.

3.3.20 Option “unpack”

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

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

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

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

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

The `define` function returns a `parse` function (see 3.1). The name of a key can be specified in three ways:

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


```

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

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

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

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

```

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

```

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

```

3.5 Attributes to define a key-value pair

The definition of a key-value pair can be made with the help of various attributes. The name “*attribute*” for an option, a key, a property ... (to list just a few naming possibilities) to define keys, was deliberately chosen to distinguish them from the options of the `parse` function. These attributes are allowed: `alias`, `always_present`, `choices`, `data_type`, `default`, `description`, `exclusive_group`, `l3_t1_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.

```

---@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
    -- defined, true is taken as the value.
    always_present = false,

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

    -- Possible data types:
    -- 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$',

-- 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 defined, true is taken as the value.

```

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

```

3.5.3 Attribute “choices”

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

```

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

```

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

```

parse('key=unknown')
-- error message:
--- '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-%d%d$' },
})
local result = parse('birthday=1978-12-03') -- { birthday = '1978-12-03' }

```

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

```

parse('birthday=1978-12-XX')
-- throws error message:
-- 'luakeys error [E009]: The value "1978-12-XX" of the key "birthday"
-- does not match "~%d%d%d%d-%d%d-%d%d$"'

```

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

```

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

```

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

```

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

```

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

3.5.10 Attribute “name”

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

```

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

```

... we can write:

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

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

The following example demonstrates the `value` parameter:

```

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

```

The following example demonstrates the `input` parameter:

```

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

```

The following example demonstrates the `result` parameter:

```

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

```

The following example demonstrates the `unknown` parameter:

```

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

```

```

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

```


3.5.14 Attribute “required”

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

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

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

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

A recursive example:

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

The `important2` key on level 2 is missing.

```
parse2('important1={unimportant}')
-- throws error message: 'luakeys error [E012]: Missing required key
↪ "important2"!'
```

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

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

3.5.15 Attribute “sub_keys”

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

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

```

})
luakeys.debug(result) -- { level1 = { level2 = 42 } }
luakeys.debug(unknown) -- { level1 = { 'unknown' } }

```

3.6 Function “render(result): string”

The function `render(result)` reverses the function `parse(kv_string)`. It takes a Lua table and converts this table into a key-value string. The resulting string usually has a different order as the input table.

```

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

```

In Lua only tables with 1-based consecutive integer keys (a.k.a. array tables) can be parsed in order.

```

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

```

3.7 Function “debug(result): void”

The function `debug(result)` pretty prints a Lua table to standard output (stdout). It is a utility function that can be used to debug and inspect the resulting Lua table of the function `parse`. You have to compile your T_EX 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 Class “DefinitionManager()”

The `DefinitionManager` class makes it possible to store key-value definitions in a central location. 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,"'
```

3.11 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.

3.11.1 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)

```

3.11.2 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)

```

3.11.3 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)

```

3.11.4 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)

```

3.11.5 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)
```

3.11.6 Function “is.list(value): boolean”

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

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

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

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

3.12 Table “utils”

The `utils` table bundles some auxiliary functions.

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

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

local tex_printf = utils.tex_printf

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

---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

```

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

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

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

```

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

```

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

```

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

```

3.13 Table “version”

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

```

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

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

```

3.14 Table “error_messages”

```

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

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

$$\langle list \rangle ::= \{ \langle list-item \rangle \}$$
$$\langle list-container \rangle ::= \{ \langle list \rangle \}$$
$$\langle list-item \rangle ::= (\langle list-container \rangle | \langle key-value-pair \rangle | \langle value \rangle) [', ']$$
$$\langle key-value-pair \rangle ::= \langle value \rangle '=' (\langle list-container \rangle | \langle value \rangle)$$
$$\begin{aligned} \langle value \rangle ::= & \langle boolean \rangle \\ & | \langle dimension \rangle \\ & | \langle number \rangle \\ & | \langle string-quoted \rangle \\ & | \langle string-unquoted \rangle \end{aligned}$$
$$\langle dimension \rangle ::= \langle number \rangle \langle unit \rangle$$
$$\langle number \rangle ::= \langle sign \rangle (\langle integer \rangle [\langle fractional \rangle] | \langle fractional \rangle)$$
$$\langle fractional \rangle ::= '.' \langle integer \rangle$$
$$\langle sign \rangle ::= '-' | '+'$$
$$\langle integer \rangle ::= \langle digit \rangle \{ \langle digit \rangle \}$$
$$\langle digit \rangle ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'$$
$$\begin{aligned} \langle unit \rangle ::= & \text{'bp'} | \text{'BP'} \\ & | \text{'cc'} | \text{'CC'} \\ & | \text{'cm'} | \text{'CM'} \\ & | \text{'dd'} | \text{'DD'} \\ & | \text{'em'} | \text{'EM'} \\ & | \text{'ex'} | \text{'EX'} \\ & | \text{'in'} | \text{'IN'} \\ & | \text{'mm'} | \text{'MM'} \\ & | \text{'mu'} | \text{'MU'} \\ & | \text{'nc'} | \text{'NC'} \\ & | \text{'nd'} | \text{'ND'} \\ & | \text{'pc'} | \text{'PC'} \\ & | \text{'pt'} | \text{'PT'} \end{aligned}$$


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

$\langle \text{boolean} \rangle ::= \langle \text{boolean-true} \rangle | \langle \text{boolean-false} \rangle$

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

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

... 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 Lua \TeX source code ([source/texk/web2c/luatexdir/luatexlib.c](#)) and the dimension module of the `lualibs` library ([lualibs-util-dim.lua](#)), all units should be recognized.

	Description
bp	big point
cc	cicero
cm	centimeter
dd	didot
em	horizontal measure of M
ex	vertical measure of x
in	inch
mm	millimeter
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{\myincludegraphics}[2][]{
  \directlua{
    print_image_macro = require('extend-keys.lua')
    print_image_macro('#2', '#1')
  }
}

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

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

5.2 Process document class options

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

```
\luaescapestring{\@optionlist{\@currname.\@current}}.
```

```
\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{\@optionlist{\@currname.\@current}}'))
}

```

```

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

```

6 Debug packages

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

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

Then the following output should appear in the document:

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

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

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

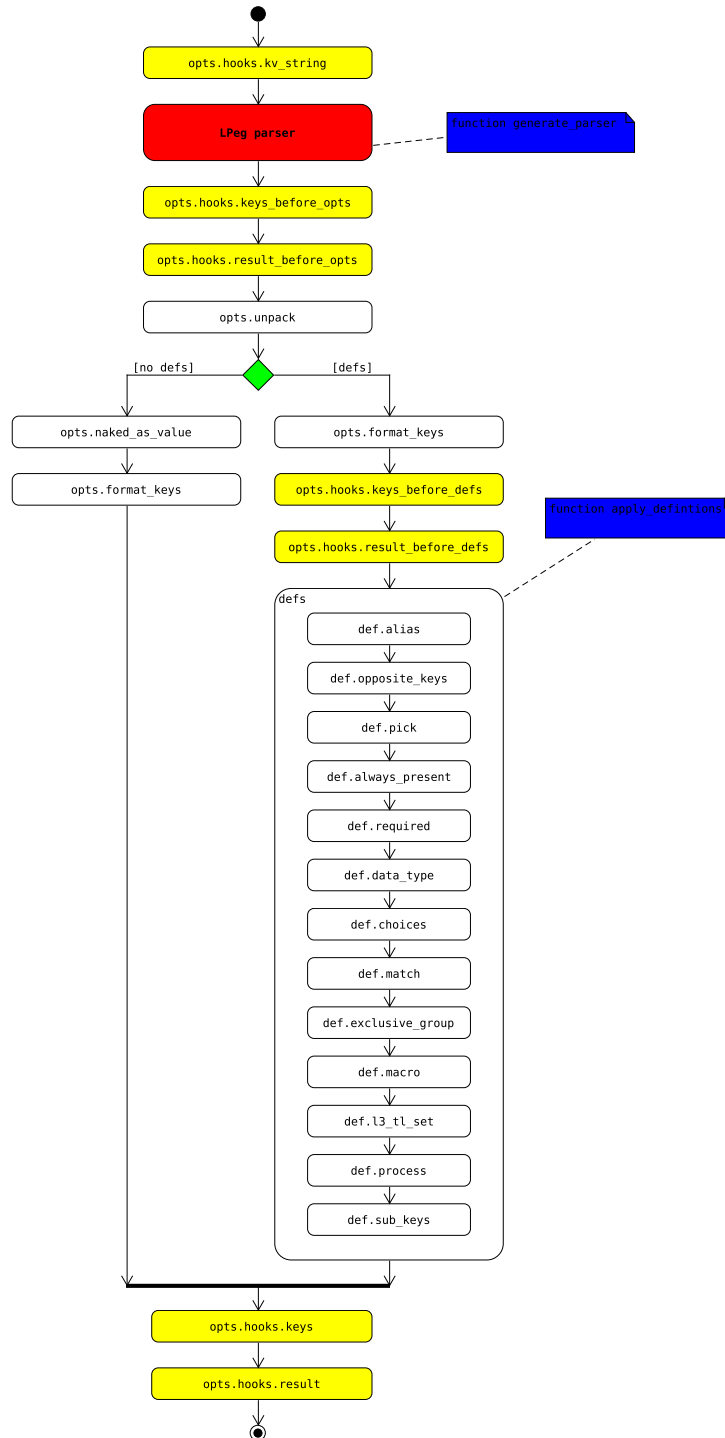
```
\input luakeys-debug.tex
\uakeysdebug{one,two,three}
\bye
```

6.2 For \LaTeX : `luakeys-debug.sty`

An example of how to use the command in \LaTeX :

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

7 Activity diagramm of the parsing process



8 Implementation

8.1 luakeys.lua

```
1  ---luakeys.lua
2  ---Copyright 2021-2024 Josef Friedrich
3  ---
4  ---This work may be distributed and/or modified under the
5  ---conditions of the LaTeX Project Public License, either version 1.3c
6  ---of this license or (at your option) any later version.
7  ---The latest version of this license is in
8  ---http://www.latex-project.org/lppl.txt
9  ---and version 1.3c or later is part of all distributions of LaTeX
10 ---version 2008/05/04 or later.
11 ---
12 ---This work has the LPPL maintenance status `maintained'.
13 ---
14 ---The Current Maintainer of this work is Josef Friedrich.
15 ---
16 ---This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
17 ---luakeys-debug.sty and luakeys-debug.tex.
18 ----A key-value parser written with Lpeg.
19 ---
20 local lpeg = require('lpeg')
21
22 if not tex then
23   ---Dummy functions for the tests.
24   tex = {
25     sp = function(input)
26       return 1234567
27     end,
28   }
29
30   token = {
31     set_macro = function(csname, content, global)
32     end,
33   }
34 end
35
36 ---
37 local utils = (function()
38   ---
39   ---Merge two tables into the first specified table.
40   ---The `merge_tables` function copies keys from the `source` table
41   ---to the `target` table. It returns the target table.
42   ---
43   ---https://stackoverflow.com/a/1283608/10193818
44   ---
45   ---@param target table # The target table where all values are copied.
46   ---@param source table # The source table from which all values are copied.
47   ---@param overwrite? boolean # Overwrite the values in the target table if they
48   ---↔ are present (default true).
49   ---
50   ---@return table target The modified target table.
51   local function merge_tables(target, source, overwrite)
52     if overwrite == nil then
53       overwrite = true
54     end
55     for key, value in pairs(source) do
56       if type(value) == 'table' and type(target[key] or false) ==
57         'table' then
58         merge_tables(target[key] or {}, source[key] or {}, overwrite)
59       end
60     end
61     return target
62   end
63 end)
```

```

58         elseif (not overwrite and target[key] == nil) or
59             (overwrite and target[key] ~= value) then
60             target[key] = value
61         end
62     end
63     return target
64 end
65
66 ---
67 ---Clone a table, i.e. make a deep copy of the source table.
68 ---
69 ---http://lua-users.org/wiki/CopyTable
70 ---
71 ---@param source table # The source table to be cloned.
72 ---
73 ---@return table # A deep copy of the source table.
74 local function clone_table(source)
75     local copy
76     if type(source) == 'table' then
77         copy = {}
78         for orig_key, orig_value in next, source, nil do
79             copy[clone_table(orig_key)] = clone_table(orig_value)
80         end
81         setmetatable(copy, clone_table(getmetatable(source)))
82     else ---number, string, boolean, etc
83         copy = source
84     end
85     return copy
86 end
87
88 ---
89 ---Remove an element from a table.
90 ---
91 ---@param source table # The source table.
92 ---@param value any # The value to be removed from the table.
93 ---
94 ---@return any/nil # If the value was found, then this value, otherwise nil.
95 local function remove_from_table(source, value)
96     for index, v in pairs(source) do
97         if value == v then
98             source[index] = nil
99             return value
100        end
101    end
102 end
103
104 ---
105 ---Return the keys of a table as a sorted list (array like table).
106 ---
107 ---@param source table # The source table.
108 ---
109 ---@return table # An array table with the sorted key names.
110 local function get_table_keys(source)
111     local keys = {}
112     for key in pairs(source) do
113         table.insert(keys, key)
114     end
115     table.sort(keys)
116     return keys
117 end
118
119 ---

```

```

120 ---Get the size of a table `{ one = 'one', 'two', 'three' }` = 3.
121 ---
122 ---@param value any # A table or any input.
123 ---
124 ---@return number # The size of the array like table. 0 if the input is no table
↪ or the table is empty.
125 local function get_table_size(value)
126     local count = 0
127     if type(value) == 'table' then
128         for _ in pairs(value) do
129             count = count + 1
130         end
131     end
132     return count
133 end
134
135 ---
136 ---Get the size of an array like table, for example `{ 'one', 'two',
137 ---'three' }` = 3.
138 ---
139 ---@param value any # A table or any input.
140 ---
141 ---@return number # The size of the array like table. 0 if the input is no table
↪ or the table is empty.
142 local function get_array_size(value)
143     local count = 0
144     if type(value) == 'table' then
145         for _ in ipairs(value) do
146             count = count + 1
147         end
148     end
149     return count
150 end
151
152 ---
153 ---Print a formatted string.
154 ---
155 ---* `%d` or `%i`: Signed decimal integer
156 ---* `%u`: Unsigned decimal integer
157 ---* `%o`: Unsigned octal
158 ---* `%x`: Unsigned hexadecimal integer
159 ---* `%X`: Unsigned hexadecimal integer (uppercase)
160 ---* `%f`: Decimal floating point, lowercase
161 ---* `%e`: Scientific notation (mantissa/exponent), lowercase
162 ---* `%E`: Scientific notation (mantissa/exponent), uppercase
163 ---* `%g`: Use the shortest representation: %e or %f
164 ---* `%G`: Use the shortest representation: %E or %F
165 ---* `%a`: Hexadecimal floating point, lowercase
166 ---* `%A`: Hexadecimal floating point, uppercase
167 ---* `%c`: Character
168 ---* `%s`: String of characters
169 ---* `%p`: Pointer address      b8000000
170 ---* `%%`: A `%` followed by another `%` character will write a single `%` to the
↪ stream.
171 ---* `%q`: formats `booleans`, `nil`, `numbers`, and `strings` in a way that the
↪ result is a valid constant in Lua source code.
172 ---
173 ---http://www.lua.org/source/5.3/lstrlib.c.html#str_format
174 ---
175 ---@param format string # A string in the `printf` format
176 ---@param ... any # A sequence of additional arguments, each containing a value to
↪ be used to replace a format specifier in the format string.

```

```

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

```

```

239     if type(template) == 'table' then
240         message = template[1]
241         if args ~= nil then
242             help = replace_args_in_list(template[2], args)
243         else
244             help = template[2]
245         end
246     else
247         message = template
248     end
249
250     if args ~= nil then
251         message = replace_args(message, args)
252     end
253
254     message = 'luakeys error [' .. error_code .. ']: ' .. message
255
256     for _, help_message in ipairs({
257         'You may be able to find more help in the documentation:',
258         'http://mirrors.ctan.org/macros/latex/generic/luakeys/luakeys-doc.pdf',
259         'Or ask a question in the issue tracker on Github:',
260         'https://github.com/Josef-Friedrich/luakeys/issues',
261     }) do
262         table.insert(help, help_message)
263     end
264
265     throw_error_message(message, help)
266 end
267
268 local function visit_tree(tree, callback_func)
269     if type(tree) ~= 'table' then
270         throw_error_message(
271             'Parameter "tree" has to be a table, got: ' ..
272             tostring(tree))
273     end
274     local function visit_tree_recursive(tree,
275         current,
276         result,
277         depth,
278         callback_func)
279         for key, value in pairs(current) do
280             if type(value) == 'table' then
281                 value = visit_tree_recursive(tree, value, {}, depth + 1,
282                     callback_func)
283             end
284
285             key, value = callback_func(key, value, depth, current, tree)
286
287             if key ~= nil and value ~= nil then
288                 result[key] = value
289             end
290         end
291         if next(result) ~= nil then
292             return result
293         end
294     end
295
296     local result =
297         visit_tree_recursive(tree, tree, {}, 1, callback_func)
298
299     if result == nil then
300         return {}

```

```

301     end
302     return result
303 end
304
305 ---@alias ColorName
306 ↪ 'black'/'red'/'green'/'yellow'/'blue'/'magenta'/'cyan'/'white'/'reset'
307 ---@alias ColorMode 'bright'/'dim'
308
309 ---
310 ---Small library to surround strings with ANSI color codes.
311 ---
312 ---[SGR (Select Graphic Rendition)
313 ↪ Parameters](https://en.wikipedia.org/wiki/ANSI_escape_code#SGR_(Select_Graphic_Rendition)_parameters)
314 ---
315 ---__attributes__
316 ---
317 ---| color      |code|
318 ---|-----|----|
319 ---| reset     | 0 |
320 ---| clear      | 0 |
321 ---| bright     | 1 |
322 ---| dim        | 2 |
323 ---| underscore | 4 |
324 ---| blink      | 5 |
325 ---| reverse    | 7 |
326 ---| hidden     | 8 |
327 ---
328 ---__foreground__
329 ---
330 ---| color      |code|
331 ---|-----|----|
332 ---| black      | 30 |
333 ---| red        | 31 |
334 ---| green      | 32 |
335 ---| yellow     | 33 |
336 ---| blue       | 34 |
337 ---| magenta    | 35 |
338 ---| cyan       | 36 |
339 ---| white      | 37 |
340 ---
341 ---__background__
342 ---
343 ---| color      |code|
344 ---|-----|----|
345 ---| onblack    | 40 |
346 ---| onred      | 41 |
347 ---| ongreen    | 42 |
348 ---| onyellow   | 43 |
349 ---| onblue     | 44 |
350 ---| onmagenta  | 45 |
351 ---| oncyan     | 46 |
352 ---| onwhite    | 47 |
353 ---
354 ---local ansi_color = (function()
355 ---
356 ---@param code integer
357 ---
358 ---@return string
359 ---local function format_color_code(code)
360     return string.char(27) .. '[' .. tostring(code) .. 'm'
361 end

```

```

361 ---
362 ---@private
363 ---
364 ---@param color ColorName # A color name.
365 ---@param mode? ColorMode
366 ---@param background? boolean # Colorize the background not the text.
367 ---
368 ---@return string
369 local function get_color_code(color, mode, background)
370     local output = ''
371     local code
372
373     if mode == 'bright' then
374         output = format_color_code(1)
375     elseif mode == 'dim' then
376         output = format_color_code(2)
377     end
378
379     if not background then
380         if color == 'reset' then
381             code = 0
382         elseif color == 'black' then
383             code = 30
384         elseif color == 'red' then
385             code = 31
386         elseif color == 'green' then
387             code = 32
388         elseif color == 'yellow' then
389             code = 33
390         elseif color == 'blue' then
391             code = 34
392         elseif color == 'magenta' then
393             code = 35
394         elseif color == 'cyan' then
395             code = 36
396         elseif color == 'white' then
397             code = 37
398         else
399             code = 37
400         end
401     else
402         if color == 'black' then
403             code = 40
404         elseif color == 'red' then
405             code = 41
406         elseif color == 'green' then
407             code = 42
408         elseif color == 'yellow' then
409             code = 43
410         elseif color == 'blue' then
411             code = 44
412         elseif color == 'magenta' then
413             code = 45
414         elseif color == 'cyan' then
415             code = 46
416         elseif color == 'white' then
417             code = 47
418         else
419             code = 40
420         end
421     end
422     return output .. format_color_code(code)

```

```

423     end
424
425     ---
426     ---@param text any
427     ---@param color ColorName # A color name.
428     ---@param mode? ColorMode
429     ---@param background? boolean # Colorize the background not the text.
430     ---
431     ---@return string
432     local function colorize(text, color, mode, background)
433         return string.format('%s%s%s',
434             get_color_code(color, mode, background), text,
435             get_color_code('reset'))
436     end
437
438     return {
439         colorize = colorize,
440
441         ---
442         ---@param text any
443         ---
444         ---@return string
445         red = function(text)
446             return colorize(text, 'red')
447         end,
448
449         ---
450         ---@param text any
451         ---
452         ---@return string
453         green = function(text)
454             return colorize(text, 'green')
455         end,
456
457         ---@return string
458         yellow = function(text)
459             return colorize(text, 'yellow')
460         end,
461
462         ---
463         ---@param text any
464         ---
465         ---@return string
466         blue = function(text)
467             return colorize(text, 'blue')
468         end,
469
470         ---
471         ---@param text any
472         ---
473         ---@return string
474         magenta = function(text)
475             return colorize(text, 'magenta')
476         end,
477
478         ---
479         ---@param text any
480         ---
481         ---@return string
482         cyan = function(text)
483             return colorize(text, 'cyan')
484         end,

```



```

485     }
486 end()
487
488 ---
489 ---A small logging library.
490 ---
491 ---Log levels:
492 ---
493 ----* 0: silent
494 ----* 1: error (red)
495 ----* 2: warn (yellow)
496 ----* 3: info (green)
497 ----* 4: verbose (blue)
498 ----* 5: debug (magenta)
499 ---
500 local log = (function()
501     ---@private
502     local opts = { level = 0 }
503
504     local function colorize_not(s)
505         return s
506     end
507
508     local colorize = colorize_not
509
510     ---@private
511     local function print_message(message, ...)
512         local args = { ... }
513         for index, value in ipairs(args) do
514             args[index] = colorize(value)
515         end
516         print(string.format(message, table.unpack(args)))
517     end
518
519     ---
520     ---Set the log level.
521     ---
522     ---@param level 0/'silent'/1/'error'/2/'warn'/3/'info'/4/'verbose'/5/'debug'
523     local function set_log_level(level)
524         if type(level) == 'string' then
525             if level == 'silent' then
526                 opts.level = 0
527             elseif level == 'error' then
528                 opts.level = 1
529             elseif level == 'warn' then
530                 opts.level = 2
531             elseif level == 'info' then
532                 opts.level = 3
533             elseif level == 'verbose' then
534                 opts.level = 4
535             elseif level == 'debug' then
536                 opts.level = 5
537             else
538                 throw_error_message(string.format('Unknown log level: %s',
539                     level))
540             end
541         else
542             if level > 5 or level < 0 then
543                 throw_error_message(string.format(
544                     'Log level out of range 0-5: %s', level))
545             end
546             opts.level = level

```

```

547     end
548 end
549
550 ---
551 ---@return integer
552 local function get_log_level()
553     return opts.level
554 end
555
556 ---
557 ---Log at level 1 (error).
558 ---
559 ---The other log levels are: 0 (silent), 1 (error), 2 (warn), 3 (info), 4
560 ↪ (verbose), 5 (debug).
561 ---
562 ---@param message string
563 ---@param ... any
564 local function error(message, ...)
565     if opts.level >= 1 then
566         colorize = ansi_color.red
567         print_message(message, ...)
568         colorize = colorize_not
569     end
570 end
571
572 ---
573 ---Log at level 2 (warn).
574 ---
575 ---The other log levels are: 0 (silent), 1 (error), 2 (warn), 3 (info), 4
576 ↪ (verbose), 5 (debug).
577 ---
578 ---@param message string
579 ---@param ... any
580 local function warn(message, ...)
581     if opts.level >= 2 then
582         colorize = ansi_color.yellow
583         print_message(message, ...)
584         colorize = colorize_not
585     end
586 end
587
588 ---
589 ---Log at level 3 (info).
590 ---
591 ---The other log levels are: 0 (silent), 1 (error), 2 (warn), 3 (info), 4
592 ↪ (verbose), 5 (debug).
593 ---
594 ---@param message string
595 ---@param ... any
596 local function info(message, ...)
597     if opts.level >= 3 then
598         colorize = ansi_color.green
599         print_message(message, ...)
600         colorize = colorize_not
601     end
602 end
603
604 ---
605 ---Log at level 4 (verbose).
606 ---
607 ---The other log levels are: 0 (silent), 1 (error), 2 (warn), 3 (info), 4
608 ↪ (verbose), 5 (debug).

```

```

605     ---
606     ---@param message string
607     ---@param ... any
608     local function verbose(message, ...)
609         if opts.level >= 4 then
610             colorize = ansi_color.blue
611             print_message(message, ...)
612             colorize = colorize_not
613         end
614     end
615
616     ---
617     ---Log at level 5 (debug).
618     ---
619     ---The other log levels are: 0 (silent), 1 (error), 2 (warn), 3 (info), 4
620     ↪ (verbose), 5 (debug).
621     ---
622     ---@param message string
623     ---@param ... any
624     local function debug(message, ...)
625         if opts.level >= 5 then
626             colorize = ansi_color.magenta
627             print_message(message, ...)
628             colorize = colorize_not
629         end
630     end
631
632     return {
633         set = set_log_level,
634         get = get_log_level,
635         error = error,
636         warn = warn,
637         info = info,
638         verbose = verbose,
639         debug = debug,
640     }
641 end()()
642
643 return {
644     merge_tables = merge_tables,
645     clone_table = clone_table,
646     remove_from_table = remove_from_table,
647     get_table_keys = get_table_keys,
648     get_table_size = get_table_size,
649     get_array_size = get_array_size,
650     visit_tree = visit_tree,
651     tex_printf = tex_printf,
652     throw_error_message = throw_error_message,
653     throw_error_code = throw_error_code,
654     ansi_color = ansi_color,
655     log = log,
656 }
657 end()()
658
659 ---
660 ---Convert back to strings
661 ---@section
662 local visualizers = (function()
663     ---
664     ---Reverse the function
665     ---`parse(kv_string)`. It takes a Lua table and converts this table

```

```

666 ---into a key-value string. The resulting string usually has a
667 ---different order as the input table. In Lua only tables with
668 ---1-based consecutive integer keys (a.k.a. array tables) can be
669 ---parsed in order.
670 ---
671 ---@param result table # A table to be converted into a key-value string.
672 ---
673 ---@return string # A key-value string that can be passed to a TeX macro.
674 local function render(result)
675     local function render_inner(result)
676         local output = {}
677         local function add(text)
678             table.insert(output, text)
679         end
680         for key, value in pairs(result) do
681             if (key and type(key) == 'string') then
682                 if (type(value) == 'table') then
683                     if (next(value)) then
684                         add(key .. '=' ..)
685                         add(render_inner(value))
686                         add('},')
687                     else
688                         add(key .. '={},')
689                     end
690                 else
691                     add(key .. '=' .. tostring(value) .. ',')
692                 end
693             else
694                 add(tostring(value) .. ',')
695             end
696         end
697         return table.concat(output)
698     end
699     return render_inner(result)
700 end
701 ---
702 ---
703 ---The function `stringify(tbl, for_tex)` converts a Lua table into a
704 ---printable string. Stringify a table means to convert the table into
705 ---a string. This function is used to realize the `debug` function.
706 ---`stringify(tbl, true)` (`for_tex = true`) generates a string which
707 ---can be embeded into TeX documents. The macro `\\luakeysdebug{}` uses
708 ---this option. `stringify(tbl, false)` or `stringify(tbl)` generate a
709 ---string suitable for the terminal.
710 ---
711 ---@see https://stackoverflow.com/a/54593224/10193818
712 ---
713 ---@param result table # A table to stringify.
714 ---@param for_tex? boolean # Stringify the table into a text string that can be
715 ↪ embeded inside a TeX document via tex.print(). Curly braces and whites spaces
716 ↪ are escaped.
717 ---
718 ---@return string
719 local function stringify(result, for_tex)
720     local line_break, start_bracket, end_bracket, indent
721
722     if for_tex then
723         line_break = '\\par'
724         start_bracket = '$\\{ '$
725         end_bracket = '$\\}$'
726         indent = '\\ \\ '
727     else

```

```

726     line_break = '\n'
727     start_bracket = '{'
728     end_bracket = '}'
729     indent = '  '
730 end
731
732 local function stringify_inner(input, depth)
733     local output = {}
734     depth = depth or 0
735
736     local function add(depth, text)
737         table.insert(output, string.rep(indent, depth) .. text)
738     end
739
740     local function format_key(key)
741         if (type(key) == 'number') then
742             return string.format('[%s]', key)
743         else
744             return string.format('[\'%s\'', key)
745         end
746     end
747
748     if type(input) ~= 'table' then
749         return tostring(input)
750     end
751
752     for key, value in pairs(input) do
753         if (key and type(key) == 'number' or type(key) == 'string') then
754             key = format_key(key)
755
756             if (type(value) == 'table') then
757                 if (next(value)) then
758                     add(depth, key .. ' = ' .. start_bracket)
759                     add(0, stringify_inner(value, depth + 1))
760                     add(depth, end_bracket .. ',')
761                 else
762                     add(depth,
763                         key .. ' = ' .. start_bracket .. end_bracket .. ',')
764                 end
765             else
766                 if (type(value) == 'string') then
767                     value = string.format('[\'%s\'', value)
768                 else
769                     value = tostring(value)
770                 end
771
772                 add(depth, key .. ' = ' .. value .. ',')
773             end
774         end
775     end
776
777     return table.concat(output, line_break)
778 end
779
780 return start_bracket .. line_break .. stringify_inner(result, 1) ..
781     line_break .. end_bracket
782 end
783
784 ---
785 ---The function `debug(result)` pretty prints a Lua table to standard
786 ---output (stdout). It is a utility function that can be used to
787 ---debug and inspect the resulting Lua table of the function

```

```

788 ---`parse`. You have to compile your TeX document in a console to
789 ---see the terminal output.
790 ---
791 ---@param result table # A table to be printed to standard output for debugging
    ↪ purposes.
792 local function debug(result)
793     print('\n' .. stringify(result, false))
794 end
795
796 return { render = render, stringify = stringify, debug = debug }
797 end()
798
799 ---@class OptionCollection
800 ---@field accumulated_result? table
801 ---@field assignment_operator? string # default `=`
802 ---@field convert_dimensions? boolean # default `false`
803 ---@field debug? boolean # default `false`
804 ---@field default? boolean # default `true`
805 ---@field defaults? table
806 ---@field defs? DefinitionCollection
807 ---@field false_aliases? table default `{ 'false', 'FALSE', 'False' }`,
808 ---@field format_keys? boolean # default `false`,
809 ---@field group_begin? string default `{`,
810 ---@field group_end? string default `}`,
811 ---@field hooks? HookCollection
812 ---@field invert_flag? string default `!`
813 ---@field list_separator? string default `,`
814 ---@field naked_as_value? boolean # default `false`
815 ---@field no_error? boolean # default `false`
816 ---@field quotation_begin? string `"`
817 ---@field quotation_end? string `"`
818 ---@field true_aliases? table `{ 'true', 'TRUE', 'True' }`
819 ---@field unpack? boolean # default `true`
820
821 ---@alias KeysHook fun(key: string, value: any, depth: integer, current: table,
    ↪ result: table): string, any
822 ---@alias ResultHook fun(result: table): nil
823
824 ---@class HookCollection
825 ---@field kv_string? fun(kv_string: string): string
826 ---@field keys_before_opts? KeysHook
827 ---@field result_before_opts? ResultHook
828 ---@field keys_before_def? KeysHook
829 ---@field result_before_def? ResultHook
830 ---@field keys? KeysHook
831 ---@field result? ResultHook
832
833 ---@alias ProcessFunction fun(value: any, input: table, result: table, unknown:
    ↪ table): any
834
835 ---@alias PickDataType 'string'|'number'|'dimension'|'integer'|'boolean'|'any'
836
837 ---@class Definition
838 ---@field alias? string|table
839 ---@field always_present? boolean
840 ---@field choices? table
841 ---@field data_type? 'boolean'|'dimension'|'integer'|'number'|'string'|'list'
842 ---@field default? any
843 ---@field description? string
844 ---@field exclusive_group? string
845 ---@field l3_tl_set? string
846 ---@field macro? string

```

```

847 ---@field match? string
848 ---@field name? string
849 ---@field opposite_keys? table
850 ---@field pick? PickDataType/PickDataType[]|false
851 ---@field process? ProcessFunction
852 ---@field required? boolean
853 ---@field sub_keys? table<string, Definition>
854
855 ---@alias DefinitionCollection table<string|number, Definition>
856
857 local namespace = {
858     opts = {
859         accumulated_result = false,
860         assignment_operator = '=',
861         convert_dimensions = false,
862         debug = false,
863         default = true,
864         defaults = false,
865         defs = false,
866         false_aliases = { 'false', 'FALSE', 'False' },
867         format_keys = false,
868         group_begin = '{',
869         group_end = '}',
870         hooks = {},
871         invert_flag = '!',
872         list_separator = ',',
873         naked_as_value = false,
874         no_error = false,
875         quotation_begin = '"',
876         quotation_end = '"',
877         true_aliases = { 'true', 'TRUE', 'True' },
878         unpack = true,
879     },
880
881     hooks = {
882         kv_string = true,
883         keys_before_opts = true,
884         result_before_opts = true,
885         keys_before_def = true,
886         result_before_def = true,
887         keys = true,
888         result = true,
889     },
890
891     attrs = {
892         alias = true,
893         always_present = true,
894         choices = true,
895         data_type = true,
896         default = true,
897         description = true,
898         exclusive_group = true,
899         l3_tl_set = true,
900         macro = true,
901         match = true,
902         name = true,
903         opposite_keys = true,
904         pick = true,
905         process = true,
906         required = true,
907         sub_keys = true,
908     },

```

```

909
910 error_messages = {
911     E001 = {
912         'Unknown parse option: @unknown!',
913         { 'The available options are:', '@opt_names' },
914     },
915     E002 = {
916         'Unknown hook: @unknown!',
917         { 'The available hooks are:', '@hook_names' },
918     },
919     E003 = 'Duplicate aliases @alias1 and @alias2 for key @key!',
920     E004 = 'The value @value does not exist in the choices: @choices',
921     E005 = 'Unknown data type: @unknown',
922     E006 = 'The value @value of the key @key could not be converted into the data
↪ type @data_type!',
923     E007 = 'The key @key belongs to the mutually exclusive group @exclusive_group
↪ and another key of the group named @another_key is already present!',
924     E008 = 'def.match has to be a string',
925     E009 = 'The value @value of the key @key does not match @match!',
926
927     E011 = 'Wrong data type in the "pick" attribute: @unknown. Allowed are:
↪ @data_types.',
928     E012 = 'Missing required key @key!',
929     E013 = 'The key definition must be a table! Got @data_type for key @key.',
930     E014 = {
931         'Unknown definition attribute: @unknown',
932         { 'The available attributes are:', '@attr_names' },
933     },
934     E015 = 'Key name couldn't be detected!',
935     E017 = 'Unknown style to format keys: @unknown! Allowed styles are: @styles',
936     E018 = 'The option "format_keys" has to be a table not @data_type',
937     E019 = 'Unknown keys: @unknown',
938
939     ---Input / parsing error
940     E021 = 'Opposite key was specified more than once: @key!',
941     E020 = 'Both opposite keys were given: @true and @false!',
942     ---Config error (wrong configuration of luakeys)
943     E010 = 'Usage: opposite_keys = { "true_key", "false_key" } or { [true] =
↪ "true_key", [false] = "false_key" } ',
944     E023 = {
945         'Don't use this function from the global luakeys table. Create a new instance
↪ using e. g.: local lk = luakeys.new()',
946         {
947             'This functions should not be used from the global luakeys table:',
948             'parse()',
949             'save()',
950             'get()',
951         },
952     },
953 },
954 }
955
956 ---
957 ---Main entry point of the module.
958 ---
959 ---The return value is intentional not documented so the Lua language server can
↪ figure out the types.
960 local function main()
961
962     ---The default options.
963     ---@type OptionCollection
964     local default_opts = utils.clone_table(namespace.opts)

```



```

965     local error_messages = utils.clone_table(namespace.error_messages)
966
967
968     ---
969     ---@param error_code string
970     ---@param args? table
971     local function throw_error(error_code, args)
972         utils.throw_error_code(error_messages, error_code, args)
973     end
974
975     ---
976     ---Normalize the parse options.
977     ---
978     ---@param opts? OptionCollection/unknown # Options in a raw format. The table may
979     ↪ be empty or some keys are not set.
980     ---
981     ---@return OptionCollection
982     local function normalize_opts(opts)
983         if type(opts) ~= 'table' then
984             opts = {}
985         end
986         for key, _ in pairs(opts) do
987             if namespace.opts[key] == nil then
988                 throw_error('E001', {
989                     unknown = key,
990                     opt_names = utils.get_table_keys(namespace.opts),
991                 })
992             end
993         end
994         local old_opts = opts
995         opts = {}
996         for name, _ in pairs(namespace.opts) do
997             if old_opts[name] ~= nil then
998                 opts[name] = old_opts[name]
999             else
1000                 opts[name] = default_opts[name]
1001             end
1002         end
1003         for hook in pairs(opts.hooks) do
1004             if namespace.hooks[hook] == nil then
1005                 throw_error('E002', {
1006                     unknown = hook,
1007                     hook_names = utils.get_table_keys(namespace.hooks),
1008                 })
1009             end
1010         end
1011         return opts
1012     end
1013
1014     local l3_code_cctab = 10
1015
1016     ---
1017     ---Parser / Lpeg related
1018     ---@section
1019
1020     ---Generate the PEG parser using Lpeg.
1021     ---
1022     ---Explanations of some LPeg notation forms:
1023     ---
1024     ---* `patt ^ 0` = `expression *`
1025     ---* `patt ^ 1` = `expression +`

```

```

1026 ----* `patt ^-1` = `expression ?`
1027 ----* `patt1 * patt2` = `expression1 expression2`: Sequence
1028 ----* `patt1 + patt2` = `expression1 / expression2`: Ordered choice
1029 ----
1030 ----* [TUGboat article: Parsing complex data formats in LuaTeX with
1031 ↪ LPEG](https://tug.or-g/TUGboat/tb40-2/tb125menke-Patterndf)
1032 ----
1033 ----@param initial_rule string # The name of the first rule of the grammar table
1034 ↪ passed to the `lpeg.P(attern)` function (e. g. `list`, `number`).
1035 ----@param opts? table # Whether the dimensions should be converted to scaled
1036 ↪ points (by default `false`).
1037 ----
1038 ----@return userdata # The parser.
1039 local function generate_parser(initial_rule, opts)
1040   if type(opts) ~= 'table' then
1041     opts = normalize_opts(opts)
1042   end
1043
1044   local Variable = lpeg.V
1045   local Pattern = lpeg.P
1046   local Set = lpeg.S
1047   local Range = lpeg.R
1048   local CaptureGroup = lpeg.Cg
1049   local CaptureFolding = lpeg.Cf
1050   local CaptureTable = lpeg.Ct
1051   local CaptureConstant = lpeg.Cc
1052   local CaptureSimple = lpeg.C
1053
1054   ----Optional whitespace
1055   local white_space = Set('\t\n\r')
1056
1057   ----Match literal string surrounded by whitespace
1058   local ws = function(match)
1059     return white_space ^ 0 * Pattern(match) * white_space ^ 0
1060   end
1061
1062   local line_up_pattern = function(patterns)
1063     local result
1064     for _, pattern in ipairs(patterns) do
1065       if result == nil then
1066         result = Pattern(pattern)
1067       else
1068         result = result + Pattern(pattern)
1069       end
1070     end
1071     return result
1072   end
1073
1074   ----
1075   ----Convert a dimension to an normalized dimension string or an
1076   ----integer in the scaled points format.
1077   ----
1078   ----@param input string
1079   ----
1080   ----@return integer/string # A dimension as an integer or a dimension string.
1081   local capture_dimension = function(input)
1082     ----Remove all whitespaces
1083     input = input:gsub('%s+', '')
1084     ----Convert the unit string into lowercase.
1085     input = input:lower()
1086     if opts.convert_dimensions then
1087       return tex.sp(input)

```

```

1085     else
1086         return input
1087     end
1088 end
1089
1090 ---
1091 ---Add values to a table in two modes:
1092 ---
1093 ---Key-value pair:
1094 ---
1095 ---If `arg1` and `arg2` are not nil, then `arg1` is the key and `arg2` is the
1096 ---value of a new table entry.
1097 ---
1098 ---Indexed value:
1099 ---
1100 ---If `arg2` is nil, then `arg1` is the value and is added as an indexed
1101 --- (by an integer) value.
1102 ---
1103 ---@param result table # The result table to which an additional key-value pair
1104 ↪ or value should be added
1105 ---@param arg1 any # The key or the value.
1106 ---@param arg2? any # Always the value.
1107 ---
1108 ---@return table # The result table to which an additional key-value pair or
1109 ↪ value has been added.
1110 local add_to_table = function(result, arg1, arg2)
1111     if arg2 == nil then
1112         local index = #result + 1
1113         return rawset(result, index, arg1)
1114     else
1115         return rawset(result, arg1, arg2)
1116     end
1117 end
1118
1119 -- LuaFormatter off
1120 return Pattern({
1121     [1] = initial_rule,
1122
1123     ---list_item*
1124     list = CaptureFolding(
1125         CaptureTable('') * Variable('list_item')^0,
1126         add_to_table
1127     ),
1128
1129     ---{' list '}
1130     list_container =
1131         ws(opts.group_begin) * Variable('list') * ws(opts.group_end),
1132
1133     --- ( list_container / key_value_pair / value ) ', '?'
1134     list_item =
1135         CaptureGroup(
1136             Variable('list_container') +
1137             Variable('key_value_pair') +
1138             Variable('value')
1139         ) * ws(opts.list_separator)^-1,
1140
1141     ---key '=' (list_container / value)
1142     key_value_pair =
1143         (Variable('key') * ws(opts.assignment_operator)) *
1144         ↪ (Variable('list_container') + Variable('value')),
1145
1146     ---number / string_quoted / string_unquoted

```

```

1144     key =
1145         Variable('number') +
1146         Variable('string_quoted') +
1147         Variable('string_unquoted'),
1148
1149     ---boolean !value / dimension !value / number !value / string_quoted !value /
1150     ↪ string_unquoted
1151     ---!value -> Not-predicate -> * -Variable('value')
1152     value =
1153         Variable('boolean') * -Variable('value') +
1154         Variable('dimension') * -Variable('value') +
1155         Variable('number') * -Variable('value') +
1156         Variable('string_quoted') * -Variable('value') +
1157         Variable('string_unquoted'),
1158
1159     ---for is.boolean()
1160     boolean_only = Variable('boolean') * -1,
1161
1162     ---boolean_true / boolean_false
1163     boolean =
1164         (
1165             Variable('boolean_true') * CaptureConstant(true) +
1166             Variable('boolean_false') * CaptureConstant(false)
1167         ),
1168     boolean_true = line_up_pattern(opts.true_aliases),
1169     boolean_false = line_up_pattern(opts.false_aliases),
1170
1171     ---for is.dimension()
1172     dimension_only = Variable('dimension') * -1,
1173
1174     dimension = (
1175         Variable('tex_number') * white_space^0 *
1176         Variable('unit')
1177     ) / capture_dimension,
1178
1179     sign = Set('-+'),
1180
1181     digit = Range('09'),
1182
1183     integer = (Variable('sign')^-1) * white_space^0 * (Variable('digit')^1),
1184
1185     fractional = (Pattern('.') ) * (Variable('digit')^1),
1186
1187     ---(integer fractional?) / (sign? white_space? fractional)
1188     tex_number = (Variable('integer') * (Variable('fractional')^-1)) +
1189         ((Variable('sign')^-1) * white_space^0 *
1190         ↪ Variable('fractional')),
1191
1192     ---for is.number()
1193     number_only = Variable('number') * -1,
1194
1195     ---capture number
1196     number = Variable('tex_number') / tonumber,
1197
1198     ---'bp' / 'BP' / 'cc' / etc.
1199
1200     ↪ ---https://raw.githubusercontent.com/latex3/lualibs/master/lualibs-util-dim.lua
1201     ↪ ---https://github.com/TeX-Live/luatex/blob/51db1985f5500dafd2393aa2e403fefa57d3cb76/source/teck/w
1202     unit =

```

```

1202     Pattern('bp') + Pattern('BP') +
1203     Pattern('cc') + Pattern('CC') +
1204     Pattern('cm') + Pattern('CM') +
1205     Pattern('dd') + Pattern('DD') +
1206     Pattern('em') + Pattern('EM') +
1207     Pattern('ex') + Pattern('EX') +
1208     Pattern('in') + Pattern('IN') +
1209     Pattern('mm') + Pattern('MM') +
1210     Pattern('mu') + Pattern('MU') +
1211     Pattern('nc') + Pattern('NC') +
1212     Pattern('nd') + Pattern('ND') +
1213     Pattern('pc') + Pattern('PC') +
1214     Pattern('pt') + Pattern('PT') +
1215     Pattern('px') + Pattern('PX') +
1216     Pattern('sp') + Pattern('SP'),
1217
1218     ---''' ('\" / !'')* '''
1219     string_quoted =
1220         white_space^0 * Pattern(opts.quotation_begin) *
1221         CaptureSimple((Pattern('\\' .. opts.quotation_end) + 1 -
1222             ↪ Pattern(opts.quotation_end))^0) *
1223         Pattern(opts.quotation_end) * white_space^0,
1224
1225     string_unquoted =
1226         white_space^0 *
1227         CaptureSimple(
1228             Variable('word_unquoted')^1 *
1229             (Set(' \\t')^1 * Variable('word_unquoted')^1)^0) *
1230         white_space^0,
1231
1232     word_unquoted = (1 - white_space - Set(
1233         opts.group_begin ..
1234         opts.group_end ..
1235         opts.assignment_operator ..
1236         opts.list_separator))^1
1237 })
1238 -- LuaFormatter on
1239 end
1240
1241 local is = {
1242     boolean = function(value)
1243         if value == nil then
1244             return false
1245         end
1246         if type(value) == 'boolean' then
1247             return true
1248         end
1249         local parser = generate_parser('boolean_only')
1250         local result = parser:match(tostring(value))
1251         return result ~= nil
1252     end,
1253
1254     dimension = function(value)
1255         if value == nil then
1256             return false
1257         end
1258         local parser = generate_parser('dimension_only')
1259         local result = parser:match(tostring(value))
1260         return result ~= nil
1261     end,
1262
1263     integer = function(value)

```

```

1263     local n = tonumber(value)
1264     if n == nil then
1265         return false
1266     end
1267     return n == math.floor(n)
1268 end,
1269
1270 number = function(value)
1271     if value == nil then
1272         return false
1273     end
1274     if type(value) == 'number' then
1275         return true
1276     end
1277     local parser = generate_parser('number_only')
1278     local result = parser:match(tostring(value))
1279     return result ~= nil
1280 end,
1281
1282 string = function(value)
1283     return type(value) == 'string'
1284 end,
1285
1286 list = function(value)
1287     if type(value) ~= 'table' then
1288         return false
1289     end
1290
1291     for k, _ in pairs(value) do
1292         if type(k) ~= 'number' then
1293             return false
1294         end
1295     end
1296     return true
1297 end,
1298
1299 any = function(value)
1300     return true
1301 end,
1302 }
1303
1304 ---
1305 ---Apply the key-value-pair definitions (defs) on an input table in a
1306 ---recursive fashion.
1307 ---
1308 ---@param defs table # A table containing all definitions.
1309 ---@param opts table # The parse options table.
1310 ---@param input table # The current input table.
1311 ---@param output table # The current output table.
1312 ---@param unknown table # Always the root unknown table.
1313 ---@param key_path table # An array of key names leading to the current
1314 ---@param input_root table # The root input table input and output table.
1315 local function apply_definitions(defs,
1316     opts,
1317     input,
1318     output,
1319     unknown,
1320     key_path,
1321     input_root)
1322     local exclusive_groups = {}
1323
1324     local function add_to_key_path(key_path, key)

```

```

1325     local new_key_path = {}
1326
1327     for index, value in ipairs(key_path) do
1328         new_key_path[index] = value
1329     end
1330
1331     table.insert(new_key_path, key)
1332     return new_key_path
1333 end
1334
1335 local function get_default_value(def)
1336     if def.default ~= nil then
1337         return def.default
1338     elseif opts ~= nil and opts.default ~= nil then
1339         return opts.default
1340     end
1341     return true
1342 end
1343
1344 local function find_value(search_key, def)
1345     if input[search_key] ~= nil then
1346         local value = input[search_key]
1347         input[search_key] = nil
1348         return value
1349         ---naked keys: values with integer keys
1350     elseif utils.remove_from_table(input, search_key) ~= nil then
1351         return get_default_value(def)
1352     end
1353 end
1354
1355 local apply = {
1356     alias = function(value, key, def)
1357         if type(def.alias) == 'string' then
1358             def.alias = { def.alias }
1359         end
1360         local alias_value
1361         local used_alias_key
1362         ---To get an error if the key and an alias is present
1363         if value ~= nil then
1364             alias_value = value
1365             used_alias_key = key
1366         end
1367         for _, alias in ipairs(def.alias) do
1368             local v = find_value(alias, def)
1369             if v ~= nil then
1370                 if alias_value ~= nil then
1371                     throw_error('E003', {
1372                         alias1 = used_alias_key,
1373                         alias2 = alias,
1374                         key = key,
1375                     })
1376                 end
1377                 used_alias_key = alias
1378                 alias_value = v
1379             end
1380         end
1381         if alias_value ~= nil then
1382             return alias_value
1383         end
1384     end,
1385
1386     always_present = function(value, key, def)

```

```

1387     if value == nil and def.always_present then
1388         return get_default_value(def)
1389     end
1390 end,
1391
1392 choices = function(value, key, def)
1393     if value == nil then
1394         return
1395     end
1396     if def.choices ~= nil and type(def.choices) == 'table' then
1397         local is_in_choices = false
1398         for _, choice in ipairs(def.choices) do
1399             if value == choice then
1400                 is_in_choices = true
1401             end
1402         end
1403         if not is_in_choices then
1404             throw_error('E004', { value = value, choices = def.choices })
1405         end
1406     end
1407 end,
1408
1409 data_type = function(value, key, def)
1410     if value == nil then
1411         return
1412     end
1413     if def.data_type ~= nil then
1414         local converted
1415         ---boolean
1416         if def.data_type == 'boolean' then
1417             if value == 0 or value == '' or not value then
1418                 converted = false
1419             else
1420                 converted = true
1421             end
1422             ---dimension
1423             elseif def.data_type == 'dimension' then
1424                 if is.dimension(value) then
1425                     converted = value
1426                 end
1427             ---integer
1428             elseif def.data_type == 'integer' then
1429                 if is.number(value) then
1430                     local n = tonumber(value)
1431                     if type(n) == 'number' and n ~= nil then
1432                         converted = math.floor(n)
1433                     end
1434                 end
1435             ---number
1436             elseif def.data_type == 'number' then
1437                 if is.number(value) then
1438                     converted = tonumber(value)
1439                 end
1440             ---string
1441             elseif def.data_type == 'string' then
1442                 converted = tostring(value)
1443             ---list
1444             elseif def.data_type == 'list' then
1445                 if is.list(value) then
1446                     converted = value
1447                 end
1448             else

```



```

1449         throw_error('E005', { data_type = def.data_type })
1450     end
1451     if converted == nil then
1452         throw_error('E006', {
1453             value = value,
1454             key = key,
1455             data_type = def.data_type,
1456         })
1457     else
1458         return converted
1459     end
1460 end
1461 end,
1462
1463 exclusive_group = function(value, key, def)
1464     if value == nil then
1465         return
1466     end
1467     if def.exclusive_group ~= nil then
1468         if exclusive_groups[def.exclusive_group] ~= nil then
1469             throw_error('E007', {
1470                 key = key,
1471                 exclusive_group = def.exclusive_group,
1472                 another_key = exclusive_groups[def.exclusive_group],
1473             })
1474         else
1475             exclusive_groups[def.exclusive_group] = key
1476         end
1477     end
1478 end,
1479
1480 l3_tl_set = function(value, key, def)
1481     if value == nil then
1482         return
1483     end
1484     if def.l3_tl_set ~= nil then
1485         tex.print(l3_code_cctab,
1486             '\\tl_set:Nn \\g_' .. def.l3_tl_set .. '_tl')
1487         tex.print('{ ' .. value .. '}')
1488     end
1489 end,
1490
1491 macro = function(value, key, def)
1492     if value == nil then
1493         return
1494     end
1495     if def.macro ~= nil then
1496         token.set_macro(def.macro, value, 'global')
1497     end
1498 end,
1499
1500 match = function(value, key, def)
1501     if value == nil then
1502         return
1503     end
1504     if def.match ~= nil then
1505         if type(def.match) ~= 'string' then
1506             throw_error('E008')
1507         end
1508         local match = string.match(value, def.match)
1509         if match == nil then
1510             throw_error('E009', {

```

```

1511         value = value,
1512         key = key,
1513         match = def.match:gsub('%', '%%'),
1514     })
1515     else
1516         return match
1517     end
1518 end
1519 end,
1520
1521 opposite_keys = function(value, key, def)
1522     if def.opposite_keys ~= nil then
1523         local function get_value(key1, key2)
1524             local opposite_name
1525             if def.opposite_keys[key1] ~= nil then
1526                 opposite_name = def.opposite_keys[key1]
1527             elseif def.opposite_keys[key2] ~= nil then
1528                 opposite_name = def.opposite_keys[key2]
1529             end
1530             return opposite_name
1531         end
1532         local true_key = get_value(true, 1)
1533         local false_key = get_value(false, 2)
1534         if true_key == nil or false_key == nil then
1535             throw_error('E010')
1536         end
1537
1538         --@param v string
1539         local function remove_values(v)
1540             local count = 0
1541             while utils.remove_from_table(input, v) do
1542                 count = count + 1
1543             end
1544             return count
1545         end
1546
1547         local true_count = remove_values(true_key)
1548         local false_count = remove_values(false_key)
1549
1550         if true_count > 1 then
1551             throw_error('E021', { key = true_key })
1552         end
1553
1554         if false_count > 1 then
1555             throw_error('E021', { key = false_key })
1556         end
1557
1558         if true_count > 0 and false_count > 0 then
1559             throw_error('E020',
1560                 { ['true'] = true_key, ['false'] = false_key })
1561         end
1562         if true_count == 0 and false_count == 0 then
1563             return
1564         end
1565         return true_count == 1 or false_count == 0
1566     end
1567 end,
1568
1569 process = function(value, key, def)
1570     if value == nil then
1571         return
1572     end

```

```

1573         if def.process ~= nil and type(def.process) == 'function' then
1574             return def.process(value, input_root, output, unknown)
1575         end
1576     end,
1577
1578     pick = function(value, key, def)
1579         if def.pick then
1580             local pick_types
1581
1582             ---Allow old deprecated attribut pick = true
1583             if def.pick == true then
1584                 pick_types = { 'any' }
1585             elseif type(def.pick) == 'table' then
1586                 pick_types = def.pick
1587             else
1588                 pick_types = { def.pick }
1589             end
1590
1591             ---Check if the pick attribute is valid
1592             for _, pick_type in ipairs(pick_types) do
1593                 if type(pick_type) == 'string' and is[pick_type] == nil then
1594                     throw_error('E011', {
1595                         unknown = tostring(pick_type),
1596                         data_types = {
1597                             'any',
1598                             'boolean',
1599                             'dimension',
1600                             'integer',
1601                             'number',
1602                             'string',
1603                         },
1604                     })
1605                 end
1606             end
1607
1608             ---The key has already a value. We leave the function at this
1609             ---point to be able to check the pick attribute for errors
1610             ---beforehand.
1611             if value ~= nil then
1612                 return value
1613             end
1614
1615             for _, pick_type in ipairs(pick_types) do
1616                 for i, v in pairs(input) do
1617                     ---We can not use ipairs here. `ipairs(t)` iterates up to the
1618                     ---first absent index. Values are deleted from the `input`
1619                     ---table.
1620                     if type(i) == 'number' then
1621                         local picked_value = nil
1622                         if is[pick_type](v) then
1623                             picked_value = v
1624                         end
1625
1626                         if picked_value ~= nil then
1627                             input[i] = nil
1628                             return picked_value
1629                         end
1630                     end
1631                 end
1632             end
1633         end
1634     end,

```

```

1635
1636     required = function(value, key, def)
1637         if def.required ~= nil and def.required and value == nil then
1638             throw_error('E012', { key = key })
1639         end
1640     end,
1641
1642     sub_keys = function(value, key, def)
1643         if def.sub_keys ~= nil then
1644             local v
1645             ---To get keys defined with always_present
1646             if value == nil then
1647                 v = {}
1648             elseif type(value) == 'string' then
1649                 v = { value }
1650             elseif type(value) == 'table' then
1651                 v = value
1652             end
1653             v = apply_definitions(def.sub_keys, opts, v, output[key],
1654                 unknown, add_to_key_path(key_path, key), input_root)
1655             if utils.get_table_size(v) > 0 then
1656                 return v
1657             end
1658         end
1659     end,
1660 }
1661
1662 ---standalone values are removed.
1663 ---For some callbacks and the third return value of parse, we
1664 ---need an unchanged raw result from the parse function.
1665 input = utils.clone_table(input)
1666 if output == nil then
1667     output = {}
1668 end
1669 if unknown == nil then
1670     unknown = {}
1671 end
1672 if key_path == nil then
1673     key_path = {}
1674 end
1675
1676 for index, def in pairs(defs) do
1677     ---Find key and def
1678     local key
1679     ---`{ key1 = { }, key2 = { } }`
1680     if type(def) == 'table' and def.name == nil and type(index) ==
1681         'string' then
1682         key = index
1683         ---`{ { name = 'key1' }, { name = 'key2' } }`
1684     elseif type(def) == 'table' and def.name ~= nil then
1685         key = def.name
1686         ---Definitions as strings in an array: `{ 'key1', 'key2' }`
1687     elseif type(index) == 'number' and type(def) == 'string' then
1688         key = def
1689         def = { default = get_default_value({}) }
1690     end
1691
1692     if type(def) ~= 'table' then
1693         throw_error('E013', { data_type = tostring(def), key = index }) ---key is
1694         ↪ nil
1695     end

```

```

1696     for attr, _ in pairs(def) do
1697         if namespace.attrs[attr] == nil then
1698             throw_error('E014', {
1699                 unknown = attr,
1700                 attr_names = utils.get_table_keys(namespace.attrs),
1701             })
1702         end
1703     end
1704
1705     if key == nil then
1706         throw_error('E015')
1707     end
1708
1709     local value = find_value(key, def)
1710
1711     for _, def_opt in ipairs({
1712         'alias',
1713         'opposite_keys',
1714         'pick',
1715         'always_present',
1716         'required',
1717         'data_type',
1718         'choices',
1719         'match',
1720         'exclusive_group',
1721         'macro',
1722         'l3_tl_set',
1723         'process',
1724         'sub_keys',
1725     }) do
1726         if def[def_opt] ~= nil then
1727             local tmp_value = apply[def_opt](value, key, def)
1728             if tmp_value ~= nil then
1729                 value = tmp_value
1730             end
1731         end
1732     end
1733
1734     output[key] = value
1735 end
1736
1737 if utils.get_table_size(input) > 0 then
1738     --Move to the current unknown table.
1739     local current_unknown = unknown
1740     for _, key in ipairs(key_path) do
1741         if current_unknown[key] == nil then
1742             current_unknown[key] = {}
1743         end
1744         current_unknown = current_unknown[key]
1745     end
1746
1747     --Copy all unknown key-value-pairs to the current unknown table.
1748     for key, value in pairs(input) do
1749         current_unknown[key] = value
1750     end
1751 end
1752
1753 return output, unknown
1754 end
1755
1756 ---
1757 ---Parse a LaTeX/TeX style key-value string into a Lua table.

```

```

1758 ---
1759 ---@param kv_string string # A string in the TeX/LaTeX style key-value format as
      ⇨ described above.
1760 ---@param opts? OptionCollection # A table containing options.
1761 ---
1762 ---@return table result # The final result of all individual parsing and
      ⇨ normalization steps.
1763 ---@return table unknown # A table with unknown, undefined key-value pairs.
1764 ---@return table raw # The unprocessed, raw result of the LPeg parser.
1765 local function parse(kv_string, opts)
1766     opts = normalize_opts(opts)
1767
1768     local function log_result(caption, result)
1769         utils.log
1770             .debug('%s: \n%s', caption, visualizers.stringify(result))
1771     end
1772
1773     if kv_string == nil then
1774         return {}, {}, {}
1775     end
1776
1777     if opts.debug then
1778         utils.log.set('debug')
1779     end
1780
1781     utils.log.debug('kv_string: "%s"', kv_string)
1782
1783     if type(opts.hooks.kv_string) == 'function' then
1784         kv_string = opts.hooks.kv_string(kv_string)
1785     end
1786
1787     local result = generate_parser('list', opts):match(kv_string)
1788     local raw = utils.clone_table(result)
1789
1790     log_result('result after Lpeg Parsing', result)
1791
1792     local function apply_hook(name)
1793         if type(opts.hooks[name]) == 'function' then
1794             if name:match('^keys') then
1795                 result = utils.visit_tree(result, opts.hooks[name])
1796             else
1797                 opts.hooks[name](result)
1798             end
1799
1800             if opts.debug then
1801                 print('After the execution of the hook: ' .. name)
1802                 visualizers.debug(result)
1803             end
1804         end
1805     end
1806
1807     local function apply_hooks(at)
1808         if at ~= nil then
1809             at = '_' .. at
1810         else
1811             at = ''
1812         end
1813         apply_hook('keys' .. at)
1814         apply_hook('result' .. at)
1815     end
1816
1817     apply_hooks('before_opts')

```

```

1818
1819 log_result('after hooks before_opts', result)
1820
1821 ---
1822 ---Normalize the result table of the LPeg parser. This normalization
1823 ---tasks are performed on the raw input table coming directly from
1824 ---the PEG parser:
1825 --
1826 ---@param result table # The raw input table coming directly from the PEG parser
1827 ---@param opts table # Some options.
1828 local function apply_opts(result, opts)
1829     local callbacks = {
1830         unpack = function(key, value)
1831             if type(value) == 'table' and utils.get_array_size(value) == 1 and
1832                 utils.get_table_size(value) == 1 and type(value[1]) ~=
1833                 'table' then
1834                 return key, value[1]
1835             end
1836             return key, value
1837         end,
1838
1839         process_naked = function(key, value)
1840             if type(key) == 'number' and type(value) == 'string' then
1841                 return value, opts.default
1842             end
1843             return key, value
1844         end,
1845
1846         format_key = function(key, value)
1847             if type(key) == 'string' then
1848                 for _, style in ipairs(opts.format_keys) do
1849                     if style == 'lower' then
1850                         key = key:lower()
1851                     elseif style == 'snake' then
1852                         key = key:gsub('[~%w]+', '_')
1853                     elseif style == 'upper' then
1854                         key = key:upper()
1855                     else
1856                         throw_error('E017', {
1857                             unknown = style,
1858                             styles = { 'lower', 'snake', 'upper' },
1859                         })
1860                     end
1861                 end
1862             end
1863             return key, value
1864         end,
1865
1866         apply_invert_flag = function(key, value)
1867             if type(key) == 'string' and key:find(opts.invert_flag) then
1868                 return key:gsub(opts.invert_flag, ''), not value
1869             end
1870             return key, value
1871         end,
1872     }
1873
1874     if opts.unpack then
1875         result = utils.visit_tree(result, callbacks.unpack)
1876     end
1877
1878     if not opts.naked_as_value and opts.defs == false then
1879         result = utils.visit_tree(result, callbacks.process_naked)

```

```

1880     end
1881
1882     if opts.format_keys then
1883         if type(opts.format_keys) ~= 'table' then
1884             throw_error('E018', { data_type = type(opts.format_keys) })
1885         end
1886         result = utils.visit_tree(result, callbacks.format_key)
1887     end
1888
1889     if opts.invert_flag then
1890         result = utils.visit_tree(result, callbacks.apply_invert_flag)
1891     end
1892
1893     return result
1894 end
1895 result = apply_opts(result, opts)
1896
1897 log_result('after apply opts', result)
1898
1899 ---All unknown keys are stored in this table
1900 local unknown = nil
1901 if type(opts.defs) == 'table' then
1902     apply_hooks('before_defs')
1903     result, unknown = apply_definitions(opts.defs, opts, result, {},
1904         {}, {}, utils.clone_table(result))
1905 end
1906
1907 log_result('after apply_definitions', result)
1908
1909 apply_hooks()
1910
1911 if opts.defaults ~= nil and type(opts.defaults) == 'table' then
1912     utils.merge_tables(result, opts.defaults, false)
1913 end
1914
1915 log_result('End result', result)
1916
1917 if opts.accumulated_result ~= nil and type(opts.accumulated_result) ==
1918     'table' then
1919     utils.merge_tables(opts.accumulated_result, result, true)
1920 end
1921
1922 ---no_error
1923 if not opts.no_error and type(unknown) == 'table' and
1924     utils.get_table_size(unknown) > 0 then
1925     throw_error('E019', { unknown = visualizers.render(unknown) })
1926 end
1927 return result, unknown, raw
1928 end
1929
1930 ---
1931 ---@param defs DefinitionCollection
1932 ---@param opts? OptionCollection
1933 local function define(defs, opts)
1934     return function(kv_string, inner_opts)
1935         local options
1936
1937         if inner_opts ~= nil and opts ~= nil then
1938             options = utils.merge_tables(opts, inner_opts)
1939         elseif inner_opts ~= nil then
1940             options = inner_opts
1941         elseif opts ~= nil then

```



```

1942     options = opts
1943 end
1944
1945 if options == nil then
1946     options = {}
1947 end
1948
1949 options.defs = defs
1950
1951 return parse(kv_string, options)
1952 end
1953 end
1954
1955 ---@alias KeySpec table<integer|string, string>
1956
1957 local DefinitionManager = (function()
1958     ---@class DefinitionManager
1959     DefinitionManager = {}
1960
1961     ---@private
1962     DefinitionManager.__index = DefinitionManager
1963
1964     ---
1965     ---@param key string
1966     ---
1967     ---@return Definition
1968     function DefinitionManager:get(key)
1969         return self.defs[key]
1970     end
1971
1972     ---
1973     ---@param key_spec KeySpec
1974     ---@param clone? boolean
1975     ---
1976     ---@return DefinitionCollection
1977     function DefinitionManager:include(key_spec, clone)
1978         local selection = {}
1979         for key, value in pairs(key_spec) do
1980             local src
1981             local dest
1982             if type(key) == 'number' then
1983                 src = value
1984                 dest = value
1985             else
1986                 src = key
1987                 dest = value
1988             end
1989             if clone then
1990                 selection[dest] = utils.clone_table(self.defs[src])
1991             else
1992                 selection[dest] = self.defs[src]
1993             end
1994         end
1995         return selection
1996     end
1997
1998     ---
1999     ---@param key_spec KeySpec
2000     ---@param clone? boolean
2001     ---
2002     ---@return DefinitionCollection
2003     function DefinitionManager:exclude(key_spec, clone)

```

```

2004     local spec = {}
2005     for key, value in pairs(key_spec) do
2006         if type(key) == 'number' then
2007             spec[value] = value
2008         else
2009             spec[key] = value
2010         end
2011     end
2012
2013     local selection = {}
2014     for key, def in pairs(self.defs) do
2015         if spec[key] == nil then
2016             if clone then
2017                 selection[key] = utils.clone_table(def)
2018             else
2019                 selection[key] = def
2020             end
2021         end
2022     end
2023     return selection
2024 end
2025
2026 ---
2027 ---@param key_selection KeySpec
2028 function DefinitionManager:parse(kv_string, key_selection)
2029     return parse(kv_string, { defs = self:include(key_selection) })
2030 end
2031
2032 ---
2033 ---@param key_selection KeySpec
2034 function DefinitionManager:define(key_selection)
2035     return define(self:include(key_selection))
2036 end
2037
2038 ---@param defs DefinitionCollection
2039 ---
2040 ---@return DefinitionManager
2041 return function(defs)
2042     local manager = {}
2043
2044     for key, def in pairs(defs) do
2045         if def.name ~= nil and type(key) == 'number' then
2046             defs[def.name] = def
2047             defs[key] = nil
2048         end
2049     end
2050
2051     setmetatable(manager, DefinitionManager)
2052     manager.defs = defs
2053     return manager
2054 end
2055 end()
2056
2057 ---
2058 ---A table to store parsed key-value results.
2059 local result_store = {}
2060
2061 return {
2062     new = main,
2063
2064     version = { 0, 14, 0 },
2065

```

```

2066     parse = parse,
2067
2068     define = define,
2069
2070     DefinitionManager = DefinitionManager,
2071
2072     ---@see default_opts
2073     opts = default_opts,
2074
2075     error_messages = error_messages,
2076
2077     ---@see visualizers.render
2078     render = visualizers.render,
2079
2080     ---@see visualizers.stringify
2081     stringify = visualizers.stringify,
2082
2083     ---@see visualizers.debug
2084     debug = visualizers.debug,
2085
2086     ---
2087     ---Save a result (a
2088     ---table from a previous run of `parse`) under an identifier.
2089     ---Therefore, it is not necessary to pollute the global namespace to
2090     ---store results for the later usage.
2091     ---
2092     ---@param identifier string # The identifier under which the result is saved.
2093     ---
2094     ---@param result table/any # A result to be stored and that was created by the
2095     ↪ key-value parser.
2096     save = function(identifier, result)
2097         result_store[identifier] = result
2098     end,
2099
2100     ---
2101     ---The function `get(identifier): table` retrieves a saved result
2102     ---from the result store.
2103     ---
2104     ---@param identifier string # The identifier under which the result was saved.
2105     ---
2106     ---@return table/any
2107     get = function(identifier)
2108         ---if result_store[identifier] == nil then
2109         --- throw_error('No stored result was found for the identifier \'' ..
2110         ↪ identifier .. '\')
2111         ---end
2112         return result_store[identifier]
2113     end,
2114
2115     is = is,
2116
2117     utils = utils,
2118
2119     ---
2120     ---Exported but intentionally undocumented functions
2121     ---
2122
2123     namespace = utils.clone_table(namespace),
2124
2125     ---
2126     ---This function is used in the documentation.
2127     ---

```

```

2126     ---@param from string # A key in the namespace table, either `opts`, `hook` or
2127     ↪  `attrs`.
2127     print_names = function(from)
2128         local names = utils.get_table_keys(namespace[from])
2129         tex.print(table.concat(names, ', '))
2130     end,
2131
2132     print_default = function(from, name)
2133         tex.print(tostring(namespace[from][name]))
2134     end,
2135
2136     print_error_messages = function()
2137         local msgs = namespace.error_messages
2138         local keys = utils.get_table_keys(namespace.error_messages)
2139         for _, key in ipairs(keys) do
2140             local msg = msgs[key]
2141             ---@type string
2142             local msg_text
2143             if type(msg) == 'table' then
2144                 msg_text = msg[1]
2145             else
2146                 msg_text = msg
2147             end
2148             utils.tex_printf('\item[\\texttt{%s}]: \\texttt{%s}', key,
2149                 msg_text)
2150         end
2151     end,
2152
2153     ---
2154     ---@param exported_table table
2155     depublish_functions = function(exported_table)
2156         local function warn_global_import()
2157             throw_error('E023')
2158         end
2159
2160         exported_table.parse = warn_global_import
2161         exported_table.define = warn_global_import
2162         exported_table.save = warn_global_import
2163         exported_table.get = warn_global_import
2164     end,
2165 }
2166
2167 end
2168
2169 return main

```

8.2 luakeys.tex

```
1 %% luakeys.tex
2 %% Copyright 2021-2024 Josef Friedrich
3 %
4 % This work may be distributed and/or modified under the
5 % conditions of the LaTeX Project Public License, either version 1.3c
6 % of this license or (at your option) any later version.
7 % The latest version of this license is in
8 % http://www.latex-project.org/lppl.txt
9 % and version 1.3c or later is part of all distributions of LaTeX
10 % version 2008/05/04 or later.
11 %
12 % This work has the LPL maintenance status `maintained'.
13 %
14 % The Current Maintainer of this work is Josef Friedrich.
15 %
16 % This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
17 % luakeys-debug.sty and luakeys-debug.tex.
18
19 \directlua{
20   if luakeys == nil then
21     luakeys = require('luakeys')()
22     luakeys.depublish_functions(luakeys)
23   end
24 }
```

8.3 luakeys.sty

```
1 %% luakeys.sty
2 %% Copyright 2021-2024 Josef Friedrich
3 %
4 % This work may be distributed and/or modified under the
5 % conditions of the LaTeX Project Public License, either version 1.3c
6 % of this license or (at your option) any later version.
7 % The latest version of this license is in
8 % http://www.latex-project.org/lppl.txt
9 % and version 1.3c or later is part of all distributions of LaTeX
10 % version 2008/05/04 or later.
11 %
12 % This work has the LPL maintenance status `maintained'.
13 %
14 % The Current Maintainer of this work is Josef Friedrich.
15 %
16 % This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
17 % luakeys-debug.sty and luakeys-debug.tex.
18
19 \NeedsTeXFormat{LaTeX2e}
20 \ProvidesPackage{luakeys}[2024/04/09 v0.14.0 Parsing key-value options using Lua.]
21 \directlua{
22   if luakeys == nil then
23     luakeys = require('luakeys')()
24     luakeys.depublish_functions(luakeys)
25   end
26 }
27
28 \def\LuakeysGetPackageOptions{\luaescapestring{\@optionlist{\@currname.\@currext}}}
29
30 \def\LuakeysGetClassOptions{\luaescapestring{\@raw@classoptionslist}}
```

8.4 luakeys-debug.tex

```
1  %% luakeys-debug.tex
2  %% Copyright 2021-2024 Josef Friedrich
3  %
4  % This work may be distributed and/or modified under the
5  % conditions of the LaTeX Project Public License, either version 1.3c
6  % of this license or (at your option) any later version.
7  % The latest version of this license is in
8  % http://www.latex-project.org/lppl.txt
9  % and version 1.3c or later is part of all distributions of LaTeX
10 % version 2008/05/04 or later.
11 %
12 % This work has the LPL maintenance status `maintained'.
13 %
14 % The Current Maintainer of this work is Josef Friedrich.
15 %
16 % This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
17 % luakeys-debug.sty and luakeys-debug.tex.
18
19 \directlua
20 {
21   luakeys = require('luakeys')()
22   if lparse == nil then
23     lparse = require('lparse')
24   end
25 }
26
27 \def\luakeysdebug%
28 {%
29   \directlua%
30   {
31     local oarg, marg = lparse.scan('o v')
32     local opts
33     if oarg then
34       opts = luakeys.parse(oarg, { format_keys = { 'snake', 'lower' } })
35     end
36     local result = luakeys.parse(marg, opts)
37     luakeys.debug(result)
38     tex.print(
39       '{' ..
40         '\string\\tt' ..
41         '\string\\parindent=0pt' ..
42         luakeys.stringify(result, true) ..
43       '}'
44     )
45   }%
46 }
```

8.5 luakeys-debug.sty

```
1 %% luakeys-debug.sty
2 %% Copyright 2021-2024 Josef Friedrich
3 %
4 % This work may be distributed and/or modified under the
5 % conditions of the LaTeX Project Public License, either version 1.3c
6 % of this license or (at your option) any later version.
7 % The latest version of this license is in
8 % http://www.latex-project.org/lppl.txt
9 % and version 1.3c or later is part of all distributions of LaTeX
10 % version 2008/05/04 or later.
11 %
12 % This work has the LPL maintenance status `maintained'.
13 %
14 % The Current Maintainer of this work is Josef Friedrich.
15 %
16 % This work consists of the files luakeys.lua, luakeys.sty, luakeys.tex
17 % luakeys-debug.sty and luakeys-debug.tex.
18
19 \NeedsTeXFormat{LaTeX2e}
20 \ProvidesPackage{luakeys-debug}[2024/04/09 v0.14.0 Debug package for luakeys.]
21
22 \input luakeys-debug.tex
```