# expkv

## an expandable ⟨*key*⟩=⟨*value*⟩ implementation

Jonathan P. Spratte*

2020-04-04 v0.5b

**Abstract**

expkv provides a small interface for ⟨*key*⟩=⟨*value*⟩ parsing. The parsing macro is fully expandable, the ⟨*code*⟩ of your keys might be not. expkv is pretty fast, but not the fastest available ⟨*key*⟩=⟨*value*⟩ solution (keyval for instance is thrice as fast, but not expandable and it might strip braces it shouldn't have stripped).

## Contents

1 **Documentation** 2
- 1.1 Setting up Keys . . . 2
- 1.2 Parsing Keys . . . 3
- 1.3 Miscellaneous . . . 4
  - 1.3.1 Other Macros . . . 4
  - 1.3.2 Bugs . . . 5
  - 1.3.3 Comparisons . . . 5
- 1.4 Examples . . . 7
  - 1.4.1 Standard Use-Case . . . 7
  - 1.4.2 An Expandable ⟨*key*⟩=⟨*value*⟩ Macro Using `\ekvsneak` . . . 9
- 1.5 Error Messages . . . 11
  - 1.5.1 Load Time . . . 11
  - 1.5.2 Defining Keys . . . 11
  - 1.5.3 Using Keys . . . 11
- 1.6 License . . . 12

2 **Implementation** 13
- 2.1 The LaTeX Package . . . 13
- 2.2 The Generic Code . . . 13

**Index** 25

*jspratte@yahoo.de

# 1 Documentation

**expkv** provides an expandable ⟨*key*⟩=⟨*value*⟩ parser. The ⟨*key*⟩=⟨*value*⟩ pairs should be given as a comma separated list and the separator between a ⟨*key*⟩ and the associated ⟨*value*⟩ should be an equal sign. Both, the commas and the equal signs, might be of category 12 (other) or 13 (active). To support this is necessary as for example babel turns characters active for some languages, for instance the equal sign is turned active for Turkish.

**expkv** is usable as generic code or as a LaTeX package. To use it, just use one of:

```
\usepackage{expkv} % LaTeX
\input expkv       % plainTeX
```

The LaTeX package doesn't do more than `expkv.tex`, except calling `\ProvidesPackage` and setting things up such that `expkv.tex` will use `\ProvidesFile`.

In the **expkv** family are other packages contained which provide additional functionality. Those packages currently are:

**expkv|def** a key-defining frontend for **expkv** using a ⟨*key*⟩=⟨*value*⟩ syntax

**expkv|cs** define expandable ⟨*key*⟩=⟨*value*⟩ macros utilizing **expkv**

Note that while the package names are stylised with a vertical rule, their names are all lower case with a hyphen (*e.g.*, `expkv-def`).

## 1.1 Setting up Keys

**expkv** provides a rather simple approach to setting up keys, similar to keyval. However there is an auxiliary package named **expkv|def** which provides a more sophisticated interface, similar to well established packages like pgfkeys or l3keys.

Keys in **expkv** (as in almost all other ⟨*key*⟩=⟨*value*⟩ implementations) belong to a *set* such that different sets can contain keys of the same name. Unlike many other implementations **expkv** doesn't provide means to set a default value, instead we have keys that take values and keys that don't (the latter are called `NoVal` keys by **expkv**), but both can have the same name (on the user level).

The following macros are available to define new keys. Those macros containing "`def`" in their name can be prefixed by anything allowed to prefix `\def`, prefixes allowed for `\let` can prefix those with "`let`" in their name, accordingly. Neither ⟨*set*⟩ nor ⟨*key*⟩ are allowed to be empty for new keys and must not contain a `\par` or tokens that expand to it – they must be legal inside of `\csname ...\endcsname`.

`\ekvdef` `\ekvdef{`⟨*set*⟩`}{`⟨*key*⟩`}{`⟨*code*⟩`}`

Defines a ⟨*key*⟩ taking a value in a ⟨*set*⟩ to expand to ⟨*code*⟩. In ⟨*code*⟩ you can use `#1` to refer to the given value.

`\ekvdefNoVal` `\ekvdefNoVal{`⟨*set*⟩`}{`⟨*key*⟩`}{`⟨*code*⟩`}`

Defines a no value taking ⟨*key*⟩ in a ⟨*set*⟩ to expand to ⟨*code*⟩.

`\ekvlet` `\ekvlet{`⟨*set*⟩`}{`⟨*key*⟩`}`⟨*cs*⟩

Let the value taking ⟨*key*⟩ in ⟨*set*⟩ to ⟨*cs*⟩, there are no checks on ⟨*cs*⟩ enforced.

\ekvletNoVal `\ekvletNoVal{⟨set⟩}{⟨key⟩}⟨cs⟩`

Let the no value taking ⟨*key*⟩ in ⟨*set*⟩ to ⟨*cs*⟩, it is not checked whether ⟨*cs*⟩ exists or that it takes no parameter.

\ekvletkv `\ekvletkv{⟨set⟩}{⟨key⟩}{⟨set2⟩}{⟨key2⟩}`

Let the ⟨*key*⟩ in ⟨*set*⟩ to ⟨*key2*⟩ in ⟨*set2*⟩, it is not checked whether that second key exists.

\ekvletkvNoVal `\ekvletkvNoVal{⟨set⟩}{⟨key⟩}{⟨set2⟩}{⟨key2⟩}`

Let the ⟨*key*⟩ in ⟨*set*⟩ to ⟨*key2*⟩ in ⟨*set2*⟩, it is not checked whether that second key exists.

## 1.2 Parsing Keys

\ekvset `\ekvset{⟨set⟩}{⟨key⟩=⟨value⟩,...}`

Splits ⟨*key*⟩=⟨*value*⟩ pairs on commas. From both ⟨*key*⟩ and ⟨*value*⟩ up to one space is stripped from both ends, if then only a braced group remains the braces are stripped as well. So `\ekvset{foo}{bar=baz}` and `\ekvset{foo}{ {bar}= {baz} }` will both do `\`⟨*foobarcode*⟩`{baz}`, so you can hide commas, equal signs and spaces at the ends of either ⟨*key*⟩ or ⟨*value*⟩ by putting braces around them. If you omit the equal sign the code of the key created with the `NoVal` variants described in subsection 1.1 will be executed. If ⟨*key*⟩=⟨*value*⟩ contains more than a single unhidden equal sign, it will be split at the first one and the others are considered part of the value. `\ekvset` should be nestable.

\ekvparse `\ekvparse⟨cs1⟩⟨cs2⟩{⟨key⟩=⟨value⟩,...}`

This macro parses the ⟨*key*⟩=⟨*value*⟩ pairs and provides those list elements which are only keys as the argument to ⟨*cs1*⟩, and those which are a ⟨*key*⟩=⟨*value*⟩ pair to ⟨*cs2*⟩ as two arguments. It is fully expandable as well and returns the parsed list in `\unexpanded`, which has no effect outside of an `\expanded` or `\edef` context[1]. If you need control over the necessary steps of expansion you can use `\expanded` around it.

`\ekvbreak`, `\ekvsneak`, and `\ekvchangeset` and their relatives don't work in `\ekvparse`. It is analogue to expl3's `\keyval_parse:NNn`, but not with the same parsing rules – `\keyval_parse:NNn` throws an error on multiple equal signs per ⟨*key*⟩=⟨*value*⟩ pair and on empty ⟨*key*⟩ names in a ⟨*key*⟩=⟨*value*⟩ pair, both of which `\ekvparse` doesn't deal with.

As a small example:

`\ekvparse\handlekey\handlekeyval{foo = bar, key, baz={zzz}}`

would expand to

`\handlekeyval{foo}{bar}\handlekey{key}\handlekeyval{baz}{zzz}`

and afterwards `\handlekey` and `\handlekeyval` would have to further handle the ⟨*key*⟩. There are no macros like these two contained in expkv, you have to set them up yourself if you want to use `\ekvparse` (of course the names might differ). If you need the results of `\ekvparse` as the argument for another macro, you should use `\expanded` as only then the input stream will contain the output above:

`\expandafter\handle\expanded{\ekvparse\k\kv{foo = bar, key, baz={zzz}}}`

would expand to

`\handle\kv{foo}{bar}\k{key}\kv{baz}{zzz}`

.

## 1.3 Miscellaneous

### 1.3.1 Other Macros

expkv provides some other macros which might be of interest.

\ekvVersion
\ekvDate

These two macros store the version and date of the package.

\ekvifdefined `\ekvifdefined{⟨set⟩}{⟨key⟩}{⟨true⟩}{⟨false⟩}`
\ekvifdefinedNoVal `\ekvifdefinedNoVal{⟨set⟩}{⟨key⟩}{⟨true⟩}{⟨false⟩}`

These two macros test whether there is a ⟨*key*⟩ in ⟨*set*⟩. It is false if either a hash table entry doesn't exist for that key or its meaning is `\relax`.

[1] This is a change in behaviour, previously (vo.3 and before) `\ekvparse` would expand in exactly two steps. This isn't always necessary, but makes the parsing considerably slower. If this is necessary for your application you can put an `\expanded` around it and will still be faster since you need only a single `\expandafter` this way.

\ekvbreak
\ekvbreakPreSneak
\ekvbreakPostSneak

`\ekvbreak{`⟨*after*⟩`}`

Gobbles the remainder of the current `\ekvset` macro and its argument list and reinserts ⟨*after*⟩. So this can be used to break out of `\ekvset`. The first variant will also gobble anything that has been sneaked out using `\ekvsneak` or `\ekvsneakPre`, while `\ekvbreakPreSneak` will put ⟨*after*⟩ before anything that has been smuggled and `\ekvbreakPostSneak` will put ⟨*after*⟩ after the stuff that has been sneaked out.

\ekvsneak
\ekvsneakPre

`\ekvsneak{`⟨*after*⟩`}`

Puts ⟨*after*⟩ after the effects of `\ekvset`. The first variant will put ⟨*after*⟩ after any other tokens which might have been sneaked before, while `\ekvsneakPre` will put ⟨*after*⟩ before other smuggled stuff. This reads and reinserts the remainder of the current `\ekvset` macro and its argument list to do its job. A small usage example is shown in subsubsection 1.4.2.

\ekvchangeset

`\ekvchangeset{`⟨*new-set*⟩`}`

Replaces the current set with ⟨*new-set*⟩, so for the rest of the current `\ekvset` call, that call behaves as if it was called with `\ekvset{`⟨*new-set*⟩`}`. Just like `\ekvsneak` this reads and reinserts the remainder of the current `\ekvset` macro to do its job. It is comparable to using ⟨*key*⟩`/.cd` in pgfkeys.

\ekv@name
\ekv@name@set
\ekv@name@key

`\ekv@name{`⟨*set*⟩`}{`⟨*key*⟩`}`
`\ekv@name@set{`⟨*set*⟩`}`
`\ekv@name@key{`⟨*key*⟩`}`

The names of the macros that correspond to a key in a set are build with these macros. The default definition of `\ekv@name@set` is "`ekv`⟨*set*⟩`(`" and the default of `\ekv@name@key` is "⟨*key*⟩`)`". The complete name is build using `\ekv@name` which is equivalent to `\ekv@name@set{`⟨*set*⟩`}\ekv@name@key{`⟨*key*⟩`}`. For `NoVal` keys an additional `N` gets appended irrespective of these macros' definition, so their name is `\ekv`⟨*set*⟩`(`⟨*key*⟩`)N`. You might redefine `\ekv@name@set` and `\ekv@name@key` locally but *don't redefine* `\ekv@name`!

### 1.3.2 Bugs

Just like keyval, **expkv** is bug free. But if you find ~~bugs~~hidden features[2] you can tell me about them either via mail (see the first page) or directly on GitHub if you have an account there: https://github.com/Skillmon/tex_expkv

### 1.3.3 Comparisons

Comparisons of speed are done with a very simple test key and the help of the l3benchmark package. The key and its usage should be equivalent to

```
\protected\ekvdef{test}{height}{\def\myheight{#1}}
\ekvset{test}{ height = 6 }
```

[2] Thanks, David!

and only the usage of the key, not its definition, is benchmarked. For the impatient, the essence of these comparisons regarding speed and buggy behaviour is contained in Table 1.

As far as I know **expkv** is the only fully expandable ⟨*key*⟩=⟨*value*⟩ parser. I tried to compare **expkv** to every ⟨*key*⟩=⟨*value*⟩ package listed on CTAN, however, one might notice that some of those are missing from this list. That's because I didn't get the others to work due to bugs, or because they just provide wrappers around other packages in this list.

In this subsubsection is no benchmark of `\ekvparse` and `\keyval_parse:NNn` contained, as most other packages don't provide equivalent features to my knowledge. `\ekvparse` is slightly faster than `\ekvset`, but keep in mind that it does less. The same is true for `\keyval_parse:NNn` compared to `\keys_set:nn` of expl3 (where the difference is much bigger).

**keyval** is about two times faster and has a comparable feature set just a slightly different way how it handles keys without values. That might be considered a drawback, as it limits the versatility, but also as an advantage, as it might reduce doubled code. Keep in mind that as soon as someone loads xkeyval the performance of keyval gets replaced by xkeyval's.

Also keyval has a bug, which unfortunately can't really be resolved without breaking backwards compatibility for *many* documents, namely it strips braces from the argument before stripping spaces if the argument isn't surrounded by spaces, also it might strip more than one set of braces. Hence all of the following are equivalent in their outcome, though the last two lines should result in something different than the first two:

```
\setkeys{foo}{bar=baz}
\setkeys{foo}{bar= {baz}}
\setkeys{foo}{bar={ baz}}
\setkeys{foo}{bar={{baz}}}
```

**xkeyval** is roughly fourteen times slower, but it provides more functionality, e.g., it has choice keys, boolean keys, and so on. It contains the same bug as keyval as it has to be compatible with it by design (it replaces keyval's frontend), but also adds even more cases in which braces are stripped that shouldn't be stripped, worsening the situation.

**ltxkeys** is over 300 times slower – which is funny, because it aims to be "[...] faster [...] than these earlier packages [referring to keyval and xkeyval]." Since it aims to have a bigger feature set than xkeyval, it most definitely also has a bigger feature set than **expkv**. Also, it can't parse `\long` input, so as soon as your values contain a `\par`, it'll throw errors. Furthermore, ltxkeys doesn't strip outer braces at all by design, which, imho, is a weird design choice. In addition ltxkeys loads catoptions which is known to introduce bugs (e.g., see https://tex.stackexchange.com/questions/461783).

**l3keys** is almost five times slower, but has an, imho, great interface to define keys. It strips *all* outer spaces, even if somehow multiple spaces ended up on either end. It offers more features, but is pretty much bound to expl3 code. Whether that's a drawback is up to you.

**pgfkeys** is a bit more than two times slower for one key, but has an *enormous* feature set. However, since adding additional keys doesn't add as much needed time for pgfkeys compared to **expkv**, it gets faster than **expkv** at around eight ⟨*key*⟩=⟨*value*⟩ pairs. It has the same or a very similar bug keyval has. The brace bug (and also the category fragility) can be fixed by pgfkeyx, but this package was last updated in 2012 and it slows down `\pgfkeys` by factor 8. Also I don't know whether this might introduce new bugs.

**kvsetkeys with kvdefinekeys** is about three times slower, but it works even if commas and equals have category codes different from 12 (just as some other packages in this list). Else the features of the keys are equal to those of keyval, the parser has more features, though.

**options** is a bit slower for only a single value, but gets a tad faster than **expkv** at around 10 ⟨*key*⟩=⟨*value*⟩ pairs. It has a much bigger feature set. Unfortunately it also suffers from the premature unbracing bug keyval has.

**simplekv** is hard to compare because I don't speak French (so I don't understand the documentation) and from what I can see, there is no direct way to define the equivalent test key. Nevertheless, I tested the closest possible equivalent of my test key while siding for simplekv's design not forcing something into it it doesn't seem to be designed for. It is more than five times slower and has hard to predict behaviour regarding brace and space stripping, similar to keyval. The tested definition was:

```
\usepackage{simplekv}
\setKVdefault[simplekv]{height={ abc}}                % key setup
\setKV[simplekv]{ height = 6 }                        % benchmarked
```

**yax** is over eighteen times slower. It has a pretty strange syntax, imho, and again a direct equivalent is hard to define. It has the premature unbracing bug, too. Also somehow loading yax broke options for me. The tested definition was:

```
\usepackage{yax}
\defactiveparameter yax {\storevalue\myheight yax:height } % key setup
\setparameterlist{yax}{ height = 6 }                      % benchmarked
```

## 1.4 Examples

### 1.4.1 Standard Use-Case

Say we have a macro for which we want to create a ⟨*key*⟩=⟨*value*⟩ interface. The macro has a parameter, which is stored in the dimension `\ourdim` having a default value from its initialization. Now we want to be able to change that dimension with the `width` key to some specified value. For that we'd do

```
\newdimen\ourdim
\ourdim=150pt
\protected\ekvdef{our}{width}{\ourdim=#1\relax}
```

as you can see, we use the set `our` here. We want the key to behave different if no value is specified. In that case the key should not use its initial value, but be smart and determine the available space from `\hsize`, so we also define

Table 1: Comparison of ⟨*key*⟩=⟨*value*⟩ packages. The packages are ordered from fastest to slowest for one ⟨*key*⟩=⟨*value*⟩ pair. Benchmarking was done using l3benchmark and the scripts in the `Benchmarks` folder of the git repository. The columns $p_i$ are the polynomial coefficients of a linear fit to the run-time, $p_0$ can be interpreted as the overhead for initialisation and $p_1$ the cost per key. The $T_0$ column is the actual mean ops needed for an empty list argument, as the linear fit doesn't match that point well in general. The column "BB" lists whether the parsing is affected by some sort of brace bug, "CF" stands for category code fragile and lists whether the parsing breaks with active commas or equal signs.

| Package | $p_1$ | $p_0$ | $T_0$ | BB | CF | Date |
|---|---|---|---|---|---|---|
| keyval | 13.5 | 1.8 | 7.0 | yes | yes | 2014-10-28 |
| **expkv** | 26.4 | 2.4 | 12.6 | no | no | 2020-02-22 |
| options | 25.5 | 10.5 | 21.4 | yes | yes | 2015-03-01 |
| pgfkeys | 26.4 | 40.9 | 55.8 | yes | yes | 2020-01-08 |
| kvsetkeys | * | * | 41.6 | no | no | 2019-12-15 |
| l3keys | 114.5 | 35.4 | 52.7 | no | no | 2020-02-14 |
| simplekv | 160.5 | 10.8 | 8.7 | yes | yes | 2017-08-08 |
| xkeyval | 260.9 | 180.4 | 164.2 | yes | yes | 2014-12-03 |
| yax | 507.6 | 62.9 | 123.5 | yes | yes | 2010-01-22 |
| ltxkeys | 3932.8 | 4737.8 | 5883.0 | no | no | 2012-11-17 |

*For kvsetkeys the linear model used for the other packages is a poor fit, kvsetkeys seems to have approximately quadratic run-time, the coefficients of the second degree polynomial fit are $p_2 = 9.9$, $p_1 = 40.5$, and $p_0 = 61.5$. Of course the other packages might not really have linear run-time, but at least from 1 to 20 keys the fits don't seem too bad (the maximum ratio $p_2/p_1$ for the other packages is $9.7 \times 10^{-3}$). If one extrapolates the fits for 100 ⟨*key*⟩=⟨*value*⟩ pairs one finds that most of them match pretty well, the exception being ltxkeys, which behaves quadratic as well with $p_2 = 31.7$, $p_1 = 3267.4$, and $p_0 = 7177.6$.

```
\protected\ekvdefNoVal{our}{width}{\ourdim=.9\hsize}
```

Now we set up our macro to use this ⟨key⟩=⟨value⟩ interface

```
\protected\def\ourmacro#1{\begingroup\ekvset{our}{#1}\the\ourdim\endgroup}
```

Finally we can use our macro like in the following

```
\ourmacro{}\par                 150.0pt
\ourmacro{width}\par            192.85382pt
\ourmacro{width=5pt}\par        5.0pt
```

**The same key using expkv|def** Using expkv|def we can set up the equivalent key using a ⟨key⟩=⟨value⟩ interface, after the following we could use `\ourmacro` in the same way as above. expkv|def will allocate and initialise `\ourdim` and define the `width` key `\protected` for us, so the result will be exactly the same – with the exception that the default will use `\ourdim=.9\hsize\relax` instead.

```
\input expkv-def                % or \usepackage{expkv-def}
\ekvdefinekeys{our}
  {
    dimen     width = \ourdim,
    qdefault  width = .9\hsize,
    initial   width = 150pt
  }
```

### 1.4.2 An Expandable ⟨key⟩=⟨value⟩ Macro Using `\ekvsneak`

Let's set up an expandable macro, that uses a ⟨key⟩=⟨value⟩ interface. The problems we'll face for this are:

1. ignoring duplicate keys
2. default values for keys which weren't used
3. providing the values as the correct argument to a macro (ordered)

First we need to decide which ⟨key⟩=⟨value⟩ parsing macro we want to do this with, `\ekvset` or `\ekvparse`. For this example we also want to show the usage of `\ekvsneak`, hence we'll choose `\ekvset`. And we'll have to use `\ekvset` such that it builds a parsable list for our macro internals. To gain back control after `\ekvset` is done we have to put an internal of our macro at the start of that list, so we use an internal key that uses `\ekvsneakPre` after any user input.

To ignore duplicates will be easy if the value of the key used last will be put first in the list, so the following will use `\ekvsneakPre` for the user-level keys. If we wanted some key for which the first usage should be the the binding one we would use `\ekvsneak` instead for that key.

Providing default values can be done in different ways, we'll use a simple approach in which we'll just put the outcome of our keys if they were used with default values before the parsing list terminator.

Ordering the keys can be done simply by searching for a specific token for each argument which acts like a flag, so our sneaked out values will include specific tokens acting as markers.

Now that we have answers for our technical problems, we have to decide what our example macro should do. How about we define a macro that calculates the sine of a number and rounds that to a specified precision? As a small extra this macro should understand input in radian and degree and the used trigonometric function should be selectable as well. For the hard part of this task (expandably evaluating trigonometric functions) we'll use the xfp package.

First we set up our keys according to our earlier considerations and set up the user facing macro `\sine`. The end marker of the parsing list will be a `\sine@stop` token, which we don't need to define and we put our defaults right before it.

```
\RequirePackage{xfp}
\makeatletter
\ekvdef{expex}{f}{\ekvsneakPre{\f{#1}}}
\ekvdef{expex}{round}{\ekvsneakPre{\rnd{#1}}}
\ekvdefNoVal{expex}{degree}{\ekvsneakPre{\deg{d}}}
\ekvdefNoVal{expex}{radian}{\ekvsneakPre{\deg{}}}
\ekvdefNoVal{expex}{internal}{\ekvsneakPre{\sine@rnd}}
\newcommand*\sine[2]
  {\ekvset{expex}{#1,internal}\rnd{3}\deg{d}\f{sin}\sine@stop{#2}}
```

For the sake of simplicity we defined the macro `\sine` with two mandatory arguments, the first being the ⟨*key*⟩=⟨`value`⟩ list, the second the argument to the trigonometric function. We could've used xparse's facilities here to define an expandable macro which takes an optional argument instead.

Now we need to define some internal macros to extract the value of each key's last usage (remember that this will be the group after the first special flag-token). For that we use one delimited macro per key.

```
\def\sine@rnd#1\rnd#2#3\sine@stop{\sine@deg#1#3\sine@stop{#2}}
\def\sine@deg#1\deg#2#3\sine@stop{\sine@f#1#3\sine@stop{#2}}
\def\sine@f#1\f#2#3\sine@stop{\sine@final{#2}}
```

After the macros `\sine@rnd`, `\sine@deg`, and `\sine@f` the macro `\sine@final` will see `\sine@final{`⟨*f*⟩`}{`⟨*degree/radian*⟩`}{`⟨*round*⟩`}{`⟨*num*⟩`}`. Now `\sine@final` has to expandably deal with those arguments such that the `\fpeval` macro of xfp gets the correct input. Luckily this is pretty straight forward in this example. In `\fpeval` the trigonometric functions have names such as `sin` or `cos` and the degree taking variants `sind` or `cosd`. And since the `degree` key puts a `d` in `#2` and the `radian` key leaves `#2` empty all we have to do to get the correct function name is stick the two together.

```
\newcommand*\sine@final[4]{\fpeval{round(#1#2(#4),#3)}}
\makeatother
```

Let's test our macro:

```
\sine{}{60}\par                                          0.866
\sine{round=10}{60}\par                                  0.8660254038
\sine{f=cos,radian}{pi}\par                              -1
\edef\myval{\sine{f=tan}{1}}\texttt{\meaning\myval}      macro:->0.017
```

**The same macro using expkvics** Using expkvics we can set up something equivalent with a bit less code. The implementation chosen in expkvics is more efficient than the example above and way easier to code.

```
\makeatletter
\ekvcSplitAndForward\sine\sine@
  {
    f=sin ,
    unit=d,
    round=3,
  }
\ekvcSecondaryKeys\sine
  {
    nmeta degree={unit=d},
    nmeta radian={unit={}},
  }
\newcommand*\sine@[4]{\fpeval{round(#1#2(#4),#3)}}
\makeatother
```

The resulting macro will behave just like the one previously defined, but will have an additional `unit` key, since in **expkv**ics every argument must have a value taking key which defines it.

## 1.5 Error Messages

**expkv** should only send messages in case of errors, there are no warnings and no info messages. In this subsection those errors are listed.

### 1.5.1 Load Time

`expkv.tex` checks whether $\varepsilon$-TEX is available. If it isn't, an error will be thrown using `\errmessage`:

```
! expkv Error: e–TeX required.
```

### 1.5.2 Defining Keys

If you get any error from **expkv** while you're trying to define a key, the definition will be aborted and gobbled.

If you try to define a key with an empty set name you'll get:

```
! expkv Error: empty set name not allowed.
```

Similarly, if you try to define a key with an empty key name:

```
! expkv Error: empty key name not allowed.
```

Both of these messages are done in a way that doesn't throw additional errors due to `\global`, `\long`, etc., not being used correctly if you prefixed one of the defining macros.

### 1.5.3 Using Keys

This subsubsection contains the errors thrown during `\ekvset`. The errors are thrown in an expandable manner by providing an undefined macro. In the following messages <*key*> gets replaced with the problematic key's name, and <*set*> with the corresponding set. If any errors during ⟨*key*⟩=⟨`value`⟩ handling are encountered, the entry

in the comma separated list will be omitted after the error is thrown and the next ⟨*key*⟩=⟨`value`⟩ pair will be parsed.

If you're using an undefined key you'll get:

```
! Undefined control sequence.
<argument> \! expkv Error:
                              unknown key ('<key>', set '<set>').
```

If you're using a key for which only a normal version and no `NoVal` version is defined, but don't provide a value, you'll get:

```
! Undefined control sequence.
<argument> \! expkv Error:
                              value required ('<key>', set '<set>').
```

If you're using a key for which only a `NoVal` version and no normal version is defined, but provide a value, you'll get:

```
! Undefined control sequence.
<argument> \! expkv Error:
                              value forbidden ('<key>', set '<set>').
```

If you're using a set for which you never executed one of the defining macros from subsection 1.1 you'll get a low level TEX error, as that isn't actively tested by the parser (and hence will lead to undefined behaviour and not be gracefully ignored). The error will look like

```
! Missing \endcsname inserted.
<to be read again>
                   \! expkv Error: Set '<set>' undefined.
```

## 1.6 License

# 2 Implementation

## 2.1 The LaTeX Package

First we set up the LaTeX package. That one doesn't really do much except \inputting the generic code and identifying itself as a package.

```
\def\ekv@tmp
  {%
    \ProvidesFile{expkv.tex}%
      [\ekvDate\space v\ekvVersion\space an expandable key=val implementation]%
  }
\input{expkv.tex}
\ProvidesPackage{expkv}%
  [\ekvDate\space v\ekvVersion\space an expandable key=val implementation]
```

## 2.2 The Generic Code

The rest of this implementation will be the generic code.

Check whether ε-TeX is available – **expkv** requires ε-TeX.

```
\begingroup\expandafter\expandafter\expandafter\endgroup
\expandafter\ifx\csname numexpr\endcsname\relax
  \errmessage{expkv requires e-TeX}
  \expandafter\endinput
\fi
```

We make sure that it's only input once:

```
\expandafter\ifx\csname ekvVersion\endcsname\relax
\else
  \expandafter\endinput
\fi
```

\ekvVersion
\ekvDate

We're on our first input, so lets store the version and date in a macro.

```
\def\ekvVersion{0.5b}
\def\ekvDate{2020-04-04}
```

(*End definition for* \ekvVersion *and* \ekvDate*. These functions are documented on page 4.*)

If the LaTeX format is loaded we want to be a good file and report back who we are, for this the package will have defined \ekv@tmp to use \ProvidesFile, else this will expand to a \relax and do no harm.

```
\csname ekv@tmp\endcsname
```

Store the category code of @ to later be able to reset it and change it to 11 for now.

```
\expandafter\chardef\csname ekv@tmp\endcsname=\catcode`\@
\catcode`\@=11
```

\ekv@tmp might later be reused to gobble any prefixes which might be provided to \ekvdef and similar in case the names are invalid, we just temporarily use it here as means to store the current category code of @ to restore it at the end of the file, we never care for the actual definition of it.

\@gobble
\@firstofone
\@firstoftwo
\@secondoftwo
\ekv@gobbleto@stop
\ekv@fi@secondoftwo
\ekv@gobble@mark

Since branching tests are often more versatile than `\if...\else...\fi` constructs, we define helpers that are branching pretty fast. Also here are some other utility functions that just grab some tokens. The ones that are also contained in LaTeX don't use the `ekv` prefix.

```
\long\def\@gobble#1{}
\long\def\@firstofone#1{#1}
\long\def\@firstoftwo#1#2{#1}
\long\def\@secondoftwo#1#2{#2}
\long\def\ekv@fi@secondoftwo\fi\@firstoftwo#1#2{\fi#2}
\long\def\ekv@gobbleto@stop#1\ekv@stop{}
\def\ekv@gobble@mark\ekv@mark{}
```

(*End definition for* `\@gobble` *and others.*)

As you can see `\ekv@gobbleto@stop` uses a special marker `\ekv@stop`. The package will use three such markers, the one you've seen already, `\ekv@mark` and `\ekv@nil`. Contrarily to how for instance expl3 does things, we don't define them, as we don't need them to have an actual meaning. This has the advantage that if they somehow get expanded – which should never happen if things work out – they'll throw an error directly.

\ekv@ifempty
\ekv@ifempty@
\ekv@ifempty@true
\ekv@ifempty@false
\ekv@ifempty@true@F

We can test for a lot of things building on an if-empty test, so lets define a really fast one. Since some tests might have reversed logic (true if something is not empty) we also set up macros for the reversed branches.

```
\long\def\ekv@ifempty#1%
  {%
    \ekv@ifempty@\ekv@ifempty@A#1\ekv@ifempty@B\ekv@ifempty@true
      \ekv@ifempty@A\ekv@ifempty@B\@secondoftwo
  }
\long\def\ekv@ifempty@#1\ekv@ifempty@A\ekv@ifempty@B{}
\long\def\ekv@ifempty@true\ekv@ifempty@A\ekv@ifempty@B\@secondoftwo#1#2{#1}
\long\def\ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo#1#2{#2}
\long\def\ekv@ifempty@true@F\ekv@ifempty@A\ekv@ifempty@B\@firstofone#1{}
```

(*End definition for* `\ekv@ifempty` *and others.*)

\ekv@ifblank@

The obvious test that can be based on an if-empty is if-blank, meaning a test checking whether the argument is empty or consists only of spaces. Our version here will be tweaked a bit, as we want to check this, but with one leading `\ekv@mark` token that is to be ignored.

```
\long\def\ekv@ifblank@\ekv@mark#1{\ekv@ifempty@\ekv@ifempty@A}
```

(*End definition for* `\ekv@ifblank@`.)

\ekv@ifdefined

We'll need to check whether something is defined quite frequently, so why not define a macro that does this. The following test is expandable, slower than the typical expandable test for undefined control sequences, but faster for defined ones. Since we want to be as fast as possible for correct input, this is to be preferred.

```
\def\ekv@ifdefined#1%
  {%
    \expandafter
    \ifx\csname\ifcsname #1\endcsname #1\else relax\fi\endcsname\relax
      \ekv@fi@secondoftwo
    \fi
```

```
    \@firstoftwo
  }
```

(*End definition for* `\ekv@ifdefined`.)

`\ekv@ifdefined@pair` `\ekv@ifdefined@pair@` `\ekv@ifdefined@key` `\ekv@ifdefined@key@`

Since we can save some time if we only have to create the control sequence once when we know beforehand how we want to use it, we build some other macros for those cases (which we'll have quite often, once per key usage).

```
\def\ekv@ifdefined@pair#1#2%
  {%
    \expandafter\ekv@ifdefined@pair@
      \csname
        \ifcsname #1{#2}\endcsname
          #1{#2}%
        \else
          relax%
        \fi
      \endcsname
  }
\def\ekv@ifdefined@pair@#1%
  {%
    \ifx#1\relax
      \ekv@fi@secondoftwo
    \fi
    \@firstoftwo
    {\ekv@set@pair@#1\ekv@mark}%
  }
\def\ekv@ifdefined@key#1#2%
  {%
    \expandafter\ekv@ifdefined@key@
      \csname
        \ifcsname #1{#2}N\endcsname
          #1{#2}N%
        \else
          relax%
        \fi
      \endcsname
  }
\def\ekv@ifdefined@key@#1%
  {%
    \ifx#1\relax
      \ekv@fi@secondoftwo
    \fi
    \@firstoftwo#1%
  }
```

(*End definition for* `\ekv@ifdefined@pair` *and others.*)

`\ekv@name` `\ekv@name@set` `\ekv@name@key`

The keys will all follow the same naming scheme, so we define it here.

```
\def\ekv@name#1#2{\ekv@name@set{#1}\ekv@name@key{#2}}
\def\ekv@name@set#1{ekv#1(}
\def\ekv@name@key#1{#1)}
```

(*End definition for* `\ekv@name`, `\ekv@name@set`, *and* `\ekv@name@key`. *These functions are documented on page* 5.)

\ekv@undefined@set

We can misuse the macro name we use to expandably store the set-name in a single token – since this increases performance drastically, especially for long set-names – to throw a more meaningful error message in case a set isn't defined. The name of \ekv@undefined@set is a little bit misleading, as it is called in either case inside of \csname, but the result will be a control sequence with meaning \relax if the set is undefined, hence will break the \csname building the key-macro which will throw the error message.

```
\def\ekv@undefined@set#1{! expkv Error: Set `#1' undefined.}
```

(*End definition for* \ekv@undefined@set.)

\ekv@checkvalid

We place some restrictions on the allowed names, though, namely sets and keys are not allowed to be empty – blanks are fine (meaning set- or key-names consisting of spaces).

```
\protected\def\ekv@checkvalid#1#2%
  {%
    \ekv@ifempty{#1}%
      {%
        \def\ekv@tmp{}%
        \errmessage{expkv Error: empty set name not allowed}%
      }%
      {%
        \ekv@ifempty{#2}%
          {%
            \def\ekv@tmp{}%
            \errmessage{expkv Error: empty key name not allowed}%
          }%
          \@secondoftwo
      }%
    \@gobble
  }
```

(*End definition for* \ekv@checkvalid.)

\ekvifdefined
\ekvifdefinedNoVal

And provide user-level macros to test whether a key is defined.

```
\def\ekvifdefined#1#2{\ekv@ifdefined{\ekv@name{#1}{#2}}}
\def\ekvifdefinedNoVal#1#2{\ekv@ifdefined{\ekv@name{#1}{#2}N}}
```

(*End definition for* \ekvifdefined *and* \ekvifdefinedNoVal. *These functions are documented on page* 4.)

\ekvdef
\ekvdefNoVal
\ekvlet
\ekvletNoVal
\ekvletkv
\ekvletkvNoVal

Set up the key defining macros \ekvdef etc.

```
\protected\long\def\ekvdef#1#2#3%
  {%
    \ekv@checkvalid{#1}{#2}%
      {%
        \expandafter\def\csname\ekv@name{#1}{#2}\endcsname##1{#3}%
        \ekv@defset{#1}%
      }%
  }
\protected\long\def\ekvdefNoVal#1#2#3%
  {%
    \ekv@checkvalid{#1}{#2}%
      {%
        \expandafter\def\csname\ekv@name{#1}{#2}N\endcsname{#3}%
        \ekv@defset{#1}%
```

```
      }%
  }
\protected\def\ekvlet#1#2#3%
  {%
    \ekv@checkvalid{#1}{#2}%
      {%
        \expandafter\let\csname\ekv@name{#1}{#2}\endcsname#3%
        \ekv@defset{#1}%
      }%
  }
\protected\def\ekvletNoVal#1#2#3%
  {%
    \ekv@checkvalid{#1}{#2}%
      {%
        \expandafter\let\csname\ekv@name{#1}{#2}N\endcsname#3%
        \ekv@defset{#1}%
      }%
  }
\protected\def\ekvletkv#1#2#3#4%
  {%
    \ekv@checkvalid{#1}{#2}%
      {%
        \expandafter\let\csname\ekv@name{#1}{#2}\expandafter\endcsname
        \csname\ekv@name{#3}{#4}\endcsname
        \ekv@defset{#1}%
      }%
  }
\protected\def\ekvletkvNoVal#1#2#3#4%
  {%
    \ekv@checkvalid{#1}{#2}%
      {%
        \expandafter\let\csname\ekv@name{#1}{#2}N\expandafter\endcsname
        \csname\ekv@name{#3}{#4}N\endcsname
        \ekv@defset{#1}%
      }%
  }
```

(*End definition for* `\ekvdef` *and others. These functions are documented on page* 2.)

`\ekv@defset` In order to enhance the speed the set name given to `\ekvset` will be turned into a control sequence pretty early, so we have to define that control sequence.

```
\protected\def\ekv@defset#1%
  {%
    \expandafter\edef\csname\ekv@undefined@set{#1}\endcsname##1%
      {\ekv@name@set{#1}\ekv@name@key{##1}}%
  }
```

(*End definition for* `\ekv@defset`.)

`\ekvset` Set up `\ekvset`, which should not be affected by active commas and equal signs. The equal signs are a bit harder to cope with and we'll do that later, but replacing the active commas with commas of category other can be done beforehand. That's why we define `\ekvset` here with a temporary meaning just to set up the things with two different category codes. `#1` will be a `,`$_{13}$ and `#2` will be a `=`$_{13}$.

```
\def\ekvset#1#2{%
\endgroup
\long\def\ekvset##1##2%
  {%
    \expandafter\ekv@set\csname\ekv@undefined@set{##1}\endcsname
      \ekv@mark##2#1\ekv@stop#1{}%
  }
```

(*End definition for* `\ekvset`. *This function is documented on page* 3.)

`\ekv@set` `\ekv@set` will split the ⟨*key*⟩=⟨*value*⟩ list at active commas. Then it has to check whether there were unprotected other commas and resplit there.

```
\long\def\ekv@set##1##2#1%
  {%
```

Test whether we're at the end, if so invoke `\ekv@endset`,

```
    \ekv@gobbleto@markstop##2\ekv@endset\ekv@mark\ekv@stop
```

else go on with other commas,

```
    \ekv@set@other##1##2,\ekv@stop,%
```

and get the next active comma delimited ⟨*key*⟩=⟨*value*⟩ pair.

```
    \ekv@set##1\ekv@mark
  }
```

(*End definition for* `\ekv@set`.)

`\ekv@endset` `\ekv@endset` is a hungry little macro. It will eat everything that remains of `\ekv@set` and unbrace the sneaked stuff.

```
\long\def\ekv@endset
    \ekv@mark\ekv@stop\ekv@set@other##1,\ekv@stop,\ekv@set##2\ekv@mark
    ##3%
  {##3}
```

(*End definition for* `\ekv@endset`.)

`\ekv@set@other` The macro `\ekv@set@other` is guaranteed to get only single ⟨*key*⟩=⟨*value*⟩ pairs. First we test whether we're done, if not split at equal signs. It is faster to first split at category 12 equal signs and only after that on actives. If there is no equal sign, we need to test whether we got a blank argument and if not this is a `NoVal` key.

```
\long\def\ekv@set@other##1##2,%
  {%
    \ekv@gobbleto@markstop##2\ekv@endset@other\ekv@mark\ekv@stop
    \ekv@ifhas@eq@other##2=\ekv@ifempty@B\ekv@ifempty@false
      \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
      {\ekv@set@eq@other##1##2\ekv@stop}%
      {%
        \ekv@ifhas@eq@active##2#2\ekv@ifempty@B\ekv@ifempty@false
          \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
          {\ekv@set@eq@active##1##2\ekv@stop}%
          {%
            \ekv@ifblank@##2\ekv@nil\ekv@ifempty@B\ekv@ifempty@true@F
              \ekv@ifempty@A\ekv@ifempty@B\@firstofone
              {\ekv@strip{##2}\ekv@set@key##1}%
          }%
```

```
      }%
    \ekv@set@other##1\ekv@mark%
  }
```

(*End definition for* `\ekv@set@other`.)

`\ekv@set@eq@other` `\ekv@set@eq@other` might not be the correct break point, there might be an active equal sign in the currently parsed key-name. If so, we have to resplit. If the split is correct strip the key-name of outer spaces and braces and feed it to `\ekv@set@pair`.

```
\long\def\ekv@set@eq@other##1##2=%
  {%
    \ekv@ifhas@eq@active##2#2\ekv@ifempty@B\ekv@ifempty@false
      \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
      {\ekv@set@eq@active##1##2=}%
      {\ekv@strip{##2}\ekv@set@pair##1}%
  }
```

(*End definition for* `\ekv@set@eq@other`.)

`\ekv@set@eq@active` We need to handle the active equal signs.

```
\long\def\ekv@set@eq@active##1##2#2%
  {%
    \ekv@strip{##2}\ekv@set@pair##1%
  }
```

(*End definition for* `\ekv@set@eq@active`.)

`\ekv@ifhas@eq@other` `\ekv@ifhas@eq@active` `\ekv@endset@other` And we have to set up the testing macros for our equal signs and `\ekv@endset@other`.

```
\long\def\ekv@ifhas@eq@other\ekv@mark##1={\ekv@ifempty@\ekv@ifempty@A}
\long\def\ekv@ifhas@eq@active\ekv@mark##1#2{\ekv@ifempty@\ekv@ifempty@A}
\long\def\ekv@endset@other
    \ekv@mark\ekv@stop
    \ekv@ifhas@eq@other##1=\ekv@ifempty@B\ekv@ifempty@false
    \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
    ##2%
    \ekv@set@other##3\ekv@mark
  {}
```

(*End definition for* `\ekv@ifhas@eq@other`, `\ekv@ifhas@eq@active`, *and* `\ekv@endset@other`.)

`\ekvbreak` `\ekvbreakPreSneak` `\ekvbreakPostSneak` Provide macros that can completely stop the parsing of `\ekvset`, who knows what it'll be useful for.

```
\long\def\ekvbreak##1##2\ekv@stop#1##3{##1}
\long\def\ekvbreakPreSneak ##1##2\ekv@stop#1##3{##1##3}
\long\def\ekvbreakPostSneak##1##2\ekv@stop#1##3{##3##1}
```

(*End definition for* `\ekvbreak`, `\ekvbreakPreSneak`, *and* `\ekvbreakPostSneak`. *These functions are documented on page* 5.)

`\ekvsneak` `\ekvsneakPre` One last thing we want to do for `\ekvset` is to provide macros that just smuggle stuff after `\ekvset`'s effects.

```
\long\def\ekvsneak##1##2\ekv@stop#1##3%
  {%
    ##2\ekv@stop#1{##3##1}%
  }
```

```
\long\def\ekvsneakPre##1##2\ekv@stop#1##3%
  {%
    ##2\ekv@stop#1{##1##3}%
  }
```

(*End definition for* `\ekvsneak` *and* `\ekvsneakPre`. *These functions are documented on page* 5.)

`\ekvparse` Additionally to the `\ekvset` macro we also want to provide an `\ekvparse` macro, that has the same scope as `\keyval_parse:NNn` from expl3. This is pretty analogue to the `\ekvset` implementation, we just put an `\unexpanded` here and there instead of other macros to stop the `\expanded` on our output.

```
\long\def\ekvparse##1##2##3%
  {%
    \ekv@parse##1##2\ekv@mark##3#1\ekv@stop#1%
  }
```

(*End definition for* `\ekvparse`. *This function is documented on page* 4.)

`\ekv@parse`

```
\long\def\ekv@parse##1##2##3#1%
  {%
    \ekv@gobbleto@markstop##3\ekv@endparse\ekv@mark\ekv@stop
    \ekv@parse@other##1##2##3,\ekv@stop,%
    \ekv@parse##1##2\ekv@mark
  }
```

(*End definition for* `\ekv@parse`.)

`\ekv@endparse`

```
\long\def\ekv@endparse
    \ekv@mark\ekv@stop\ekv@parse@other##1,\ekv@stop,\ekv@parse##2\ekv@mark
  {}
```

(*End definition for* `\ekv@endparse`.)

`\ekv@parse@other`

```
\long\def\ekv@parse@other##1##2##3,%
  {%
    \ekv@gobbleto@markstop##3\ekv@endparse@other\ekv@mark\ekv@stop
    \ekv@ifhas@eq@other##3=\ekv@ifempty@B\ekv@ifempty@false
      \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
      {\ekv@parse@eq@other##3\ekv@stop##2}%
      {%
        \ekv@ifhas@eq@active##3#2\ekv@ifempty@B\ekv@ifempty@false
          \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
          {\ekv@parse@eq@active##3\ekv@stop##2}%
          {%
            \ekv@ifblank@##3\ekv@nil\ekv@ifempty@B\ekv@ifempty@true@F
              \ekv@ifempty@A\ekv@ifempty@B\@firstofone
              {\ekv@strip{##3}\ekv@parse@key##1}%
          }%
      }%
    \ekv@parse@other##1##2\ekv@mark
  }
```

(*End definition for* `\ekv@parse@other`.)

`\ekv@parse@eq@other`

```
\long\def\ekv@parse@eq@other##1=%
  {%
    \ekv@ifhas@eq@active##1#2\ekv@ifempty@B\ekv@ifempty@false
      \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
      {\ekv@parse@eq@active##1=}%
      {\ekv@strip{##1}\ekv@parse@pair\ekv@mark}%
  }
```

(*End definition for* `\ekv@parse@eq@other`.)

`\ekv@parse@eq@active`

```
\long\def\ekv@parse@eq@active##1#2%
  {%
    \ekv@strip{##1}\ekv@parse@pair\ekv@mark
  }
```

(*End definition for* `\ekv@parse@eq@active`.)

`\ekv@endparse@other`

```
\long\def\ekv@endparse@other
    \ekv@mark\ekv@stop
    \ekv@ifhas@eq@other##1=\ekv@ifempty@B\ekv@ifempty@false
    \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
    ##2%
    \ekv@parse@other##3\ekv@mark
  {}
```

(*End definition for* `\ekv@endparse@other`.)

`\ekv@parse@pair`
`\ekv@parse@pair@`

```
\long\def\ekv@parse@pair##1##2\ekv@stop
  {%
    \ekv@strip{##2}\ekv@parse@pair@{##1}%
  }
\long\def\ekv@parse@pair@##1##2##3%
  {%
    \unexpanded{##3{##2}{##1}}%
  }
```

(*End definition for* `\ekv@parse@pair` *and* `\ekv@parse@pair@`.)

`\ekv@parse@key`

```
\long\def\ekv@parse@key##1##2%
  {%
    \unexpanded{##2{##1}}%
  }
```

(*End definition for* `\ekv@parse@key`.)

Finally really setting things up with `\ekvset`'s temporary meaning:

```
}
\begingroup
\catcode`\,=13
\catcode`\==13
\ekvset,=
```

\ekvchangeset Provide a macro that is able to switch out the current ⟨*set*⟩ in \ekvset. This operation is slow (by comparison, it should be slightly faster than \ekvsneak), but allows for something similar to pgfkeys's ⟨*key*⟩/.cd mechanism. However this operation is more expensive than /.cd as we can't just redefine some token to reflect this, but have to switch out the set expandably, so this works similar to the \ekvsneak macros reading and reinserting the remainder of the ⟨*key*⟩=⟨*value*⟩ list.

```
\def\ekvchangeset#1%
  {%
    \expandafter\ekv@changeset\csname\ekv@undefined@set{#1}\endcsname\ekv@mark
  }
```

(*End definition for* \ekvchangeset. *This function is documented on page* 5.)

\ekv@changeset This macro does the real change-out of \ekvchangeset. We introduced an \ekv@mark to not accidentally remove some braces which we have to remove again.

```
\long\def\ekv@changeset#1#2\ekv@set@other#3#4\ekv@set#5%
  {%
    \ekv@gobble@mark#2\ekv@set@other#1#4\ekv@set#1%
  }
```

(*End definition for* \ekv@changeset.)

\ekv@gobbleto@markstop The \ekv@gobbleto@markstop can be used to test for \ekv@stop similar to our if-empty test, but instead of using tokens which are used nowhere else (\ekv@ifempty@A and \ekv@ifempty@B) it uses \ekv@mark and \ekv@stop.

```
\long\def\ekv@gobbleto@markstop#1\ekv@mark\ekv@stop{}
```

(*End definition for* \ekv@gobbleto@markstop.)

\ekv@set@pair \ekv@set@pair@ \ekv@set@pair gets invoked with the space and brace stripped key-name as its first argument, the set-macro as the second argument, and following that is the key-value right delimited by an \ekv@stop.

```
\long\def\ekv@set@pair#1#2%
  {%
    \ekv@ifdefined@pair#2{#1}%
      {%
```

This branch will be executed if the key is not defined as an argument grabbing one. If so test whether there is a NoVal key of the same name or whether the key is unknown. Throw a meaningful error message and gobble the value.

```
        \ekv@ifdefined{#2{#1}N}%
          \ekv@err@noarg
          \ekv@err@unknown
          #2{#1}%
        \ekv@gobbleto@stop
      }%
  }
```

\ekv@ifdefined@pair will call \ekv@set@pair@ if the key is correctly defined. This will then grab the value, strip outer spaces and braces from it and feed it to the key-macro. Afterwards \ekv@set@other will take control again.

```
\long\def\ekv@set@pair@#1#2\ekv@stop
  {%
    \ekv@strip{#2}#1%
  }
```

(*End definition for* `\ekv@set@pair` *and* `\ekv@set@pair@`.)

`\ekv@set@key` Analogous to `\ekv@set@pair`, `\ekv@set@key` lets `\ekv@ifdefined@key` test whether a `NoVal` key is defined, else it'll throw a meaningful error message. Since we don't have to grab any value `\ekv@ifdefined@key` will invoke the key-macro and we're done here, `\ekv@set@other` will take over again.

```
\long\def\ekv@set@key#1#2%
  {%
    \ekv@ifdefined@key#2{#1}%
      {%
        \ekv@ifdefined{#2{#1}}%
          \ekv@err@reqval
          \ekv@err@unknown
          #2{#1}%
      }%
  }
```

(*End definition for* `\ekv@set@key`.)

`\ekv@err` `\ekv@err@` Since `\ekvset` is fully expandable as long as the code of the keys is (which is unlikely) we want to somehow throw expandable errors, in our case via undefined control sequences.

```
\begingroup
\edef\ekv@err
  {%
    \endgroup
    \unexpanded{\long\def\ekv@err}##1%
      {%
        \unexpanded{\expandafter\ekv@err@\@firstofone}%
        {\unexpanded\expandafter{\csname ! expkv Error:\endcsname}##1.}%
        \unexpanded{\ekv@stop}%
      }%
  }
\ekv@err
\def\ekv@err@{\expandafter\ekv@gobbleto@stop}
```

(*End definition for* `\ekv@err` *and* `\ekv@err@`.)

`\ekv@err@common` `\ekv@err@common@` `\ekv@err@unknown` `\ekv@err@noarg` `\ekv@err@reqval` Now we can use `\ekv@err` to set up some error messages so that we can later use those instead of the full strings.

```
\long\def\ekv@err@common #1#2{\expandafter\ekv@err@common@\string#2{#1}}
\long\def\ekv@err@common@#1‘#2’ #3.#4#5{\ekv@err{#4 (‘#5’, set ‘#2’)}}
\long\def\ekv@err@unknown#1#2{\ekv@err@common{unknown key}#1{#2}}
\long\def\ekv@err@noarg  #1#2{\ekv@err@common{value forbidden}#1{#2}}
\long\def\ekv@err@reqval #1#2{\ekv@err@common{value required}#1{#2}}
```

(*End definition for* `\ekv@err@common` *and others.*)

`\ekv@strip` `\ekv@strip@a` `\ekv@strip@b` `\ekv@strip@c` Finally we borrow some ideas of expl3's l3tl to strip spaces from keys and values. This `\ekv@strip` also strips one level of outer braces *after* stripping spaces, so an input of `  {abc}  ` becomes `abc` after stripping. It should be used with `#1` prefixed by `\ekv@mark`. Also this implementation at most strips *one* space from both sides.

```
\def\ekv@strip#1%
  {%
    \long\def\ekv@strip##1%
```

```
      {%
        \ekv@strip@a
          ##1%
          \ekv@nil
          \ekv@mark#1%
          #1\ekv@nil{}%
        \ekv@stop
      }%
    \long\def\ekv@strip@a##1\ekv@mark#1##2\ekv@nil##3%
      {%
        \ekv@strip@b##3##1##2\ekv@nil
      }%
    \long\def\ekv@strip@b##1#1\ekv@nil
      {%
        \ekv@strip@c##1\ekv@nil
      }%
    \long\def\ekv@strip@c\ekv@mark##1\ekv@nil##2\ekv@stop##3%
      {%
        ##3{##1}%
      }%
  }
\ekv@strip{ }
```

(*End definition for* `\ekv@strip` *and others.*)

Now everything that's left is to reset the category code of `@`.

```
\catcode`\@=\ekv@tmp
```

# Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.


**E**

\ekvbreak . . . . . 5, 218
\ekvbreakPostSneak . . . . . 5, 218
\ekvbreakPreSneak . . . . . 5, 218
\ekvchangeset . . . . . 5, 295
\ekvDate . . . . . 4, 4, 8, 18
\ekvdef . . . . . 2, 108
\ekvdefNoVal . . . . . 2, 108
\ekvifdefined . . . . . 4, 106
\ekvifdefinedNoVal . . . . . 4, 106
\ekvlet . . . . . 2, 108
\ekvletkv . . . . . 3, 108
\ekvletkvNoVal . . . . . 3, 108
\ekvletNoVal . . . . . 3, 108
\ekvparse . . . . . 4, 229
\ekvset . . . . . 3, 163, 294
\ekvsneak . . . . . 5, 221
\ekvsneakPre . . . . . 5, 221
\ekvVersion . . . . . 4, 4, 8, 18

**T**

TEX and LATEX 2ε commands:

\@firstofone . . . . . 23, 38, 192, 254, 335
\@firstoftwo . . . . 23, 37, 46, 64, 83, 184, 188, 201, 214, 246, 250, 263, 274
\@gobble . . . . . 23, 104
\@secondoftwo . . . . . 23, 33, 36, 102
\ekv@changeset . . . . . 297, 299
\ekv@checkvalid . . . . . 89, 110, 118, 126, 134, 142, 151
\ekv@defset . . . . . 113, 121, 129, 137, 146, 155, 158
\ekv@endparse . . . . . 235, 239
\ekv@endparse@other . . . . . 244, 271
\ekv@endset . . . . . 172, 176
\ekv@endset@other . . . . . 182, 209
\ekv@err . . . . . 329, 343
\ekv@err@ . . . . . 329
\ekv@err@common . . . . . 342
\ekv@err@common@ . . . . . 342
\ekv@err@noarg . . . . . 309, 342
\ekv@err@reqval . . . . . 324, 342
\ekv@err@unknown . . . . . 310, 325, 342
\ekv@fi@secondoftwo . . . . 23, 44, 62, 81
\ekv@gobble@mark . . . . . 23, 301
\ekv@gobbleto@markstop . . . . . 172, 182, 235, 244, 303
\ekv@gobbleto@stop . . . . . 23, 312, 341
\ekv@ifblank@ . . . . . 39, 191, 253
\ekv@ifdefined . . 40, 106, 107, 308, 323
\ekv@ifdefined@key . . . . . 48, 321
\ekv@ifdefined@key@ . . . . . 48
\ekv@ifdefined@pair . . . . . 48, 306
\ekv@ifdefined@pair@ . . . . . 48
\ekv@ifempty . . . . . 30, 91, 97
\ekv@ifempty@ . . . . . 30, 39, 209, 210
\ekv@ifempty@A . . . . . 32, 33, 35, 36, 37, 38, 39, 184, 188, 192, 201, 209, 210, 214, 246, 250, 254, 263, 274
\ekv@ifempty@B . . . . . 32, 33, 35, 36, 37, 38, 183, 184, 187, 188, 191, 192, 200, 201, 213, 214, 245, 246, 249, 250, 253, 254, 262, 263, 273, 274
\ekv@ifempty@false . . . . . 30, 183, 187, 200, 213, 245, 249, 262, 273
\ekv@ifempty@true . . . . . 30
\ekv@ifempty@true@F . . . . . 30, 191, 253
\ekv@ifhas@eq@active . . . . . 187, 200, 209, 249, 262
\ekv@ifhas@eq@other 183, 209, 245, 273
\ekv@mark . 29, 39, 65, 168, 172, 174, 177, 182, 196, 209, 210, 212, 216, 231, 235, 237, 240, 244, 258, 265, 269, 272, 276, 297, 303, 354, 358, 366
\ekv@name . . . . . 5, 85, 106, 107, 112, 120, 128, 136, 144, 145, 153, 154
\ekv@name@key . . . . . 5, 85, 161
\ekv@name@set . . . . . 5, 85, 161
\ekv@nil . . . . . 191, 253, 353, 355, 358, 360, 362, 364, 366
\ekv@parse . . . . . 231, 233, 240
\ekv@parse@eq@active . . . 251, 264, 267
\ekv@parse@eq@other . . . . . 247, 260
\ekv@parse@key . . . . . 255, 286
\ekv@parse@other . . 236, 240, 242, 276
\ekv@parse@pair . . . . . 265, 269, 278
\ekv@parse@pair@ . . . . . 278
\ekv@set . . . . . 167, 170, 177, 299, 301

\ekv@set@eq@active . . . . . 189, 202, 205
\ekv@set@eq@other . . . . . 185, 198
\ekv@set@key . . . . . 193, 319
\ekv@set@other . . . . . . . . . . . . . . . . . 173, 177, 180, 216, 299, 301
\ekv@set@pair . . . . . 203, 207, 304
\ekv@set@pair@ . . . . . 65, 304
\ekv@stop . . . . . 28, 168, 172, 173, 177, 182, 185, 189, 212, 218, 219, 220, 221, 223, 225, 227, 231, 235, 236, 240, 244, 247, 251, 272, 278, 303, 315, 337, 356, 366
\ekv@strip . . . . . 193, 203, 207, 255, 265, 269, 280, 317, 347
\ekv@strip@a . . . . . 347
\ekv@strip@b . . . . . 347
\ekv@strip@c . . . . . 347
\ekv@tmp . . . . . 1, 93, 99, 372
\ekv@undefined@set . . 88, 160, 167, 297