

# expkv

## an expandable $\langle key \rangle = \langle value \rangle$ implementation

Jonathan P. Spratte\*

2020-12-28 v1.6

### Abstract

expkv provides a small interface for  $\langle key \rangle = \langle value \rangle$  parsing. The parsing macro is *fully expandable*, the  $\langle code \rangle$  of your keys might be not. expkv is *swift*, close to the fastest  $\langle key \rangle = \langle value \rangle$  implementation. However it is the fastest which copes with active commas and equal signs and doesn't strip braces accidentally.

## Contents

<b>1</b>	<b>Documentation</b>	<b>2</b>
1.1	Setting up Keys . . . . .	2
1.2	Parsing Keys . . . . .	3
1.3	Miscellaneous . . . . .	5
1.3.1	Other Macros . . . . .	5
1.3.2	Bugs . . . . .	6
1.3.3	Comparisons . . . . .	6
1.4	Examples . . . . .	8
1.4.1	Standard Use-Case . . . . .	8
1.4.2	An Expandable $\langle key \rangle = \langle value \rangle$ Macro Using <code>\ekvsneak</code> . . . . .	10
1.5	Error Messages . . . . .	12
1.5.1	Load Time . . . . .	12
1.5.2	Defining Keys . . . . .	12
1.5.3	Using Keys . . . . .	12
1.6	License . . . . .	13
<b>2</b>	<b>Implementation</b>	<b>14</b>
2.1	The L <sup>A</sup> T <sub>E</sub> X Package . . . . .	14
2.2	The Generic Code . . . . .	14
	<b>Index</b>	<b>31</b>

---

\*jspratte@yahoo.de

## 1 Documentation

`expkv` provides an expandable  $\langle key \rangle = \langle value \rangle$  parser. The  $\langle key \rangle = \langle value \rangle$  pairs should be given as a comma separated list and the separator between a  $\langle key \rangle$  and the associated  $\langle value \rangle$  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:

`expkvDEF` a key-defining frontend for `expkv` using a  $\langle key \rangle = \langle value \rangle$  syntax

`expkvICS` define expandable  $\langle key \rangle = \langle value \rangle$  macros using `expkv`

`expkvOPT` parse package and class options with `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 `expkvDEF` which provides a more sophisticated interface, similar to well established packages like `pgfkeys` or `l3keys`.

Keys in `expkv` (as in almost all other  $\langle key \rangle = \langle value \rangle$  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` (but *don't* use `\outer`, keys defined with it won't ever be usable), prefixes allowed for `\let` can prefix those with "let" in their name, accordingly. Neither  $\langle set \rangle$  nor  $\langle key \rangle$  are allowed to be empty for new keys.  $\langle set \rangle$  will be used as is inside of `\csname ... \endcsname` and  $\langle key \rangle$  will get `\detokenized`.

---

```
\ekvdef \ekvdef{\set}{\key}{\code}
```

Defines a  $\langle key \rangle$  taking a value in a  $\langle set \rangle$  to expand to  $\langle code \rangle$ . In  $\langle code \rangle$  you can use `#1` to refer to the given value.

---

```
\ekvdefNoVal \ekvdefNoVal{\set}{\key}{\code}
```

Defines a no value taking  $\langle key \rangle$  in a  $\langle set \rangle$  to expand to  $\langle code \rangle$ .

---

`\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 (but take a look at `\ekvifdefined`).

---

`\ekvletkvNoVal` `\ekvletkvNoVal{<set>}{<key>}{<set2>}{<key2>}`  
Let the `<key>` in `<set>` to `<key2>` in `<set2>`, it is not checked whether that second key exists (but take a look at `\ekvifdefinedNoVal`).

---

`\ekvdefunknown` `\ekvdefunknown{<set>}{<code>}`  
By default an error will be thrown if an unknown `<key>` is encountered. With this macro you can define `<code>` that will be executed for a given `<set>` when an unknown `<key>` with a `<value>` was encountered instead of throwing an error. You can refer to the given `<value>` with #1 and to the unknown `<key>`'s name with #2 in `<code>`.<sup>1</sup>

---

`\ekvdefunknownNoVal` `\ekvdefunknownNoVal{<set>}{<code>}`  
As already explained for `\ekvdefunknown`, `expkv` would throw an error when encountering an unknown `<key>`. With this you can instead let it execute `<code>` if an unknown `NoVal <key>` was encountered. You can refer to the given `<key>` with #1 in `<code>`.

## 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 `\{foobrcode\}{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.

---

`\ekvsetSneaked` `\ekvsetSneaked{<set>}{<sneak>}{<key>=<value>, ...}`  
Just like `\ekvset`, this macro parses the `<key>=<value>` pairs within the given `<set>`. But `\ekvsetSneaked` will behave as if `\ekvsneak` has been called with `<sneak>` as its argument as the first action.

---

<sup>1</sup>That order is correct, this way the code is faster.

---

`\ekvsetdef` `\ekvsetdef<cs>{<set>}`

With this function you can define a shorthand macro `<cs>` to parse keys of a specified `<set>`. It is always defined `\long`, but if you need to you can also prefix it with `\global`. The resulting macro is faster than but else equivalent to the idiomatic definition:

```
\long\def<cs>#1{\ekvset{<set>}{#1}}
```

---

`\ekvsetSneakeddef` `\ekvsetSneakeddef<cs>{<set>}`

Just like `\ekvsetdef` this defines a shorthand macro `<cs>`, but this macro will make it a shorthand for `\ekvsetSneaked`, meaning that `<cs>` will take two arguments, the first being stuff that should be given to `\ekvsneak` and the second the `<key>=<value>` list. The resulting macro is faster than but else equivalent to the idiomatic definition:

```
\long\def<cs>#1#2{\ekvsetSneaked{<set>}{#1}{#2}}
```

---

`\ekvsetdefSneaked` `\ekvsetSneakeddef<cs>{<set>}{<sneaked>}`

And this one behaves like `\ekvsetSneakeddef` but with a fixed `<sneaked>` argument. So the resulting macro is faster than but else equivalent to the idiomatic definition:

```
\long\def<cs>#1{\ekvsetSneaked{<set>}{<sneaked>}{#1}}
```

---

`\ekvparse` `\ekvparse{<code1>}{<code2>}{<key>=<value>,...}`

This macro parses the `<key>=<value>` pairs and provides those list elements which are only keys as an argument to `<code1>`, and those which are a `<key>=<value>` pair to `<code2>` as two arguments. It is fully expandable as well and returns each element of the parsed list in `\unexpanded`, which has no effect outside of an `\expanded` or `\edef` context<sup>2</sup>. If you need control over the necessary steps of expansion you can use `\expanded` around it. You can use multiple tokens in `<code1>` and `<code2>` or just a single control sequence name. In both cases the found `<key>` and `<value>` are provided as a brace group following them.

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

would be equivalent to

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

and afterwards `\handlekey` and `\handlekeyval` would have to further handle the `<key>`. There are no macros like these two contained in `explkv`, 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}}}
```

---

<sup>2</sup>This is a change in behaviour, previously (v0.3 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.

would expand to

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

## 1.3 Miscellaneous

### 1.3.1 Other Macros

`explkv` provides some other macros which might be of interest.

---

`\ekvVersion`  
`\ekvDate`

---

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

---

`\ekvifdefined`  
`\ekvifdefinedNoVal`

---

```
\ekvifdefined{<set>}{<key>}{<>true>}{<>false>}  
\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`.

---

`\ekvifdefinedset`

---

```
\ekvifdefinedset{<set>}{<>true>}{<>false>}
```

This macro tests whether `<set>` is defined (which it is if at least one key was defined for it). If it is `<>true>` will be run, else `<>false>`.

---

`\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. After `\ekvset` has parsed the entire `<key>=<value>` list everything that has been `\ekvsneaked` will be left in the input stream. 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`.

---

<code>\ekv@name</code>	<code>\ekv@name{&lt;set&gt;}{&lt;key&gt;}</code>
<code>\ekv@name@set</code>	<code>\ekv@name@set{&lt;set&gt;}</code>
<code>\ekv@name@key</code>	<code>\ekv@name@key{&lt;key&gt;}</code>

---

The names of the macros that correspond to a key in a set are built with these macros. The name is built from two blocks, one that is formatting the `<set>` name (`\ekv@name@set`) and one for formatting the `<key>` name (`\ekv@name@key`). To get the actual name the argument to `\ekv@name@key` must be `\detokenized`. Both blocks are put together (with the necessary `\detokenize`) by `\ekv@name`. For `NoVal` keys an additional `N` gets appended irrespective of these macros' definition, so their name is `\ekv@name{<set>}{<key>N}`.

You can use these macros to implement additional functionality or access key macros outside of `expkv`, but *don't* change them! `expkv` relies on their exact definitions internally.

### 1.3.2 Bugs

Just like `keyval`, `expkv` is bug free. But if you find `bugshidden` features<sup>3</sup> 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](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}}
\ekvsetdef\expkvtest{test}
\expkvtest{ height = 6 }
```

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 subsection 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). Comparing just the two, `\ekvparse` is a tad faster than `\keyval_parse:NNn` because of the two tests (for empty key names and only a single equal sign) which are omitted.

`keyval` is about 30% to 40% 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.

---

<sup>3</sup>Thanks, David!

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=bar}
\setkeys{foo}{bar= {bar}}
\setkeys{foo}{bar={ bar}} % should be ' bar'
\setkeys{foo}{bar={{bar}}} % should be '{bar}'
```

`xkeyval` is roughly twenty 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 no longer compatible with the L<sup>A</sup>T<sub>E</sub>X kernel starting with the release 2020-10-01. It is over 380 times slower – which is funny, because it aims to be “[...] faster [...] than these earlier packages [referring to `keyval` and `xkeyval`].” It needs more time to parse zero keys than five of the packages in this comparison need to parse 100 keys. Since it aims to have a bigger feature set than `xkeyval`, it most definitely also has a bigger feature set than `explv`. 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>). Because it is no longer compatible with the kernel, I stop benchmarking it (so the numbers listed here and in [Table 1](#) regarding `ltxkeys` were last updated on 2020-10-05).

`l3keys` is around four and a half 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 around 2.7 times slower for one key if one uses the `//<path>/ .cd` syntax and almost 20% slower if one uses `\pgfqkeys`, but has an *enormous* feature set. To get the best performance `\pgfqkeys` was used in the benchmark. This reduces the overhead for setting the base directory of the benchmark keys by about 43 ops (so both  $p_0$  and  $T_0$  would be about 43 ops bigger if `\pgfkeys{<path>/ .cd, <keys>}` was used instead). 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 `pgfkeyx` is no longer compatible with versions of `pgfkeys` newer than 2020-05-25.

`kvsetkeys` with `kvdefinekeys` is about 4.4 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 1.7 times slower for only a single value. 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). There was an update released on 2020-04-27 which greatly improved the package's performance and adds functionality so that it can be used more like most of the other  $\langle key \rangle = \langle value \rangle$  packages. It has problems with stripping braces and spaces in a hard to predict manner just like `keyval`. Also, while it tries to be robust against category code changes of commas and equal signs, the used mechanism fails if the  $\langle key \rangle = \langle value \rangle$  list already got tokenised. Regarding unknown keys it got a very interesting behaviour. It doesn't throw an error, but stores the  $\langle value \rangle$  in a new entry accessible with `\useKV`. Also if you omit  $\langle value \rangle$  it stores `true` for that  $\langle key \rangle$ . For up to three keys, `explkv` is a bit faster, for more keys `simplekv` takes the lead.

`yax` is over twenty 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  $\langle key \rangle = \langle value \rangle$  interface. The macro has a parameter, which is stored in the dimension `\ourdim` having a default value from its initialisation. 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

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

Now we set up our macro to use this  $\langle key \rangle = \langle value \rangle$  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
```

Table 1: Comparison of  $\langle key \rangle = \langle value \rangle$  packages. The packages are ordered from fastest to slowest for one  $\langle key \rangle = \langle value \rangle$  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	13.7	1.5	7.3	yes	yes	2014-10-28
expkv	19.7	2.2	6.6	no	no	2020-10-10
simplekv	18.3	7.0	17.7	yes	yes	2020-04-27
pgfkeys	24.3	1.7	10.7	yes	yes	2020-09-05
options	23.6	15.6	20.8	yes	yes	2015-03-01
kvsetkeys	*	*	40.3	no	no	2019-12-15
l3keys	71.3	33.1	31.6	no	no	2020-09-24
xkeyval	253.6	202.2	168.3	yes	yes	2014-12-03
yax	421.9	157.0	114.7	yes	yes	2010-01-22
ltxkeys	3400.1	4738.0	5368.0	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  $p_2 = 8.2$ ,  $p_1 = 44.9$ , and  $p_0 = 60.8$ . 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. If one extrapolates the fits for 100  $\langle key \rangle = \langle value \rangle$  pairs one finds that most of them match pretty well, the exception being `ltxkeys`, which behaves quadratic as well with  $p_2 = 23.5$ ,  $p_1 = 2906.6$ , and  $p_0 = 6547.5$ .

**The same key using `expkvDEF`** Using `expkvDEF` 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. `expkvDEF` 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 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 for the user.

```

\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`s 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  $\epsilon$ -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 subsection 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 NoVa1 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 T<sub>E</sub>X 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

Copyright © 2020 Jonathan P. Spratte

This work may be distributed and/or modified under the conditions of the L<sup>A</sup>T<sub>E</sub>X Project Public License (LPPL), either version 1.3c of this license or (at your option) any later version. The latest version of this license is in the file:

<http://www.latex-project.org/lppl.txt>

This work is “maintained” (as per LPPL maintenance status) by  
Jonathan P. Spratte.

## 2 Implementation

### 2.1 The L<sup>A</sup>T<sub>E</sub>X Package

First we set up the L<sup>A</sup>T<sub>E</sub>X package. That one doesn't really do much except `\inputting` the generic code and identifying itself as a package.

```
1 \def\ekv@tmp
2   {%
3     \ProvidesFile{expkv.tex}%
4     [\ekvDate\space v\ekvVersion\space an expandable key=val implementation]%
5   }
6 \input{expkv.tex}
7 \ProvidesPackage{expkv}%
8   [\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.

We make sure that it's only input once:

```
9 \expandafter\ifx\csname ekvVersion\endcsname\relax
10 \else
11   \expandafter\endinput
12 \fi
    Check whether  $\varepsilon$ -TEX is available – expkv requires  $\varepsilon$ -TEX.
13 \begingroup\expandafter\expandafter\expandafter\endgroup
14 \expandafter\ifx\csname numexpr\endcsname\relax
15   \errmessage{expkv requires e-TeX}
16 \expandafter\endinput
17 \fi
```

`\ekvVersion` We're on our first input, so let's store the version and date in a macro.

```
\ekvDate
18 \def\ekvVersion{1.6}
19 \def\ekvDate{2020-12-28}
```

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

If the L<sup>A</sup>T<sub>E</sub>X 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.

```
20 \csname ekv@tmp\endcsname
    Store the category code of @ to later be able to reset it and change it to 11 for now.
21 \expandafter\chardef\csname ekv@tmp\endcsname=\catcode'\@
22 \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.

`\ekv@if@lastnamedcs` If the primitive `\lastnamedcs` is available, we can be a bit faster than without it. So we test for this and save the test's result in this macro.

```

23 \begingroup
24 \edef\ekv@tmpa{\string \lastnamedcs}
25 \edef\ekv@tmpb{\meaning\lastnamedcs}
26 \ifx\ekv@tmpa\ekv@tmpb
27 \def\ekv@if@lastnamedcs{\long\def\ekv@if@lastnamedcs##1##2{##1}}
28 \else
29 \def\ekv@if@lastnamedcs{\long\def\ekv@if@lastnamedcs##1##2{##2}}
30 \fi
31 \expandafter
32 \endgroup
33 \ekv@if@lastnamedcs

```

*(End definition for `\ekv@if@lastnamedcs`.)*

`\ekv@empty` Sometimes we have to introduce a token to prevent accidental brace stripping. This token would then need to be removed by `\@gobble` or similar. Instead we can use `\ekv@empty` which will just expand to nothing, that is faster than gobbling an argument.

```

34 \def\ekv@empty{}

```

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

`\@gobble` 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 L<sup>A</sup>T<sub>E</sub>X don't use the `ekv` prefix. Not all of the ones defined here are really needed by `expl3` but are provided because packages like `expl3DEF` or `expl3OPT` need them (and I don't want to define them in each package which might need them).

```

35 \long\def\@gobble#1{}
36 \long\def\@firstofone#1{#1}
37 \long\def\@firstoftwo#1#2{#1}
38 \long\def\@secondoftwo#1#2{#2}
39 \long\def\ekv@fi@gobble\fi\@firstofone#1{\fi}
40 \long\def\ekv@fi@firstoftwo\fi\@secondoftwo#1#2{\fi#1}
41 \long\def\ekv@fi@secondoftwo\fi\@firstoftwo#1#2{\fi#2}
42 \long\def\ekv@gobbleto@stop#1\ekv@stop{}
43 \def\ekv@gobble@mark\ekv@mark{}
44 \long\def\ekv@gobble@from@mark@to@stop\ekv@mark#1\ekv@stop{}

```

*(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` 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.

```

45 \long\def\ekv@ifempty#1%
46 {%
\ekv@ifempty@
\ekv@ifempty@true
\ekv@ifempty@false
\ekv@ifempty@true@F
\ekv@ifempty@true@F@gobble
\ekv@ifempty@true@F@gobbleto

```

```

47   \ekv@ifempty@\ekv@ifempty@A#1\ekv@ifempty@B\ekv@ifempty@true
48   \ekv@ifempty@A\ekv@ifempty@B\@secondoftwo
49   }
50 \long\def\ekv@ifempty@#1\ekv@ifempty@A\ekv@ifempty@B{}
51 \long\def\ekv@ifempty@true\ekv@ifempty@A\ekv@ifempty@B\@secondoftwo#1#2{#1}
52 \long\def\ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo#1#2{#2}
53 \long\def\ekv@ifempty@true@F\ekv@ifempty@A\ekv@ifempty@B\@firstofone#1{}
54 \long\def\ekv@ifempty@true@F@gobble\ekv@ifempty@A\ekv@ifempty@B\@firstofone#1#2%
55   {}
56 \long\def\ekv@ifempty@true@F@gobbletwo
57   \ekv@ifempty@A\ekv@ifempty@B\@firstofone#1#2#3%
58   {}

```

(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. The wrapper `\ekv@ifblank` will not be used by `explkv` for speed reasons but `explkv|OPT` uses it.

`\ekv@ifblank@`

```

59 \long\def\ekv@ifblank#1%
60   {%
61     \ekv@ifblank@#1\ekv@nil\ekv@ifempty@B\ekv@ifempty@true
62     \ekv@ifempty@A\ekv@ifempty@B\@secondoftwo
63   }
64 \long\def\ekv@ifblank@\ekv@mark#1{\ekv@ifempty@\ekv@ifempty@A}

```

(End definition for `\ekv@ifblank` and `\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 and pretty fast. The version with `\lastnamedcs` is the fastest version to test for an undefined macro I know of (that considers both undefined macros and those with the meaning `\relax`).

```

65 \ekv@if@lastnamedcs
66   {%
67     \def\ekv@ifdefined#1{\ifcsname#1\endcsname\ekv@ifdef@fi\@secondoftwo}
68     \def\ekv@ifdef@fi\@secondoftwo
69       {%
70         \fi
71         \expandafter\ifx\lastnamedcs\relax
72         \ekv@fi@secondoftwo
73         \fi
74         \@firstoftwo
75       }
76   }
77   {%
78     \def\ekv@ifdefined#1%
79     {%
80       \ifcsname#1\endcsname\ekv@ifdef@fi\ekv@ifdef@false#1\endcsname\relax
81       \ekv@fi@secondoftwo
82       \fi
83       \@firstoftwo
84     }
85     \def\ekv@ifdef@fi\ekv@ifdef@false{\fi\expandafter\ifx\cename}

```

```

86 \long\def\ekv@ifdef@false
87   #1\endcsname\relax\ekv@fi@secondoftwo\fi\@firstoftwo#2#3%
88   {#3}
89 }

```

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

`\ekv@strip` 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 (which should be fine most of the time, since `TEX` reads consecutive spaces as a single one during tokenisation).

```

90 \def\ekv@strip#1%
91   {%
92   \long\def\ekv@strip##1%
93     {%
94       \ekv@strip@a
95       ##1\ekv@nil
96       \ekv@mark#1%
97       #1\ekv@nil
98     }%
99   \long\def\ekv@strip@a##1\ekv@mark#1{\ekv@strip@b##1\ekv@mark}%
100  }
101 \ekv@strip{ }
102 \long\def\ekv@strip@b#1 \ekv@nil{\ekv@strip@c#1\ekv@nil}
103 \long\def\ekv@strip@c\ekv@mark#1\ekv@nil\ekv@mark#2\ekv@nil#3{#3{#1}}

```

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

`\ekv@exparg` To reduce some code doublets while gaining some speed (and also as convenience for other packages in the family), it is often useful to expand the first token in a definition once. Let's define a wrapper for this.

`\ekv@expargtwice` Also, to end a `\romannumeral` expansion, we want to use `\z@`, which is contained in both plain `TEX` and `LATEX`, but we use a private name for it to make it easier to spot and hence easier to manage.

```

104 \let\ekv@zero\z@
105 \long\def\ekv@exparg#1#2{\expandafter\ekv@exparg@\expandafter{#2}{#1}}
106 \long\def\ekv@exparg@#1#2{#2{#1}}%
107 \long\def\ekv@expargtwice#1#2{\expandafter\ekv@expargtwice@\expandafter{#2}{#1}}
108 \def\ekv@expargtwice@{\expandafter\ekv@exparg@\expandafter}

```

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

`\ekv@name` The keys will all follow the same naming scheme, so we define it here.

```

\ekv@name@set 109 \def\ekv@name@set#1{ekv#1{}}
\ekv@name@key 110 \def\ekv@name@key#1{#1}
111 \edef\ekv@name
112   {%
113   \unexpanded\expandafter{\ekv@name@set{#1}}%
114   \unexpanded\expandafter{\ekv@name@key{\detokenize{#2}}}%
115   }
116 \ekv@exparg{\def\ekv@name#1#2}{\ekv@name}

```

(End definition for `\ekv@name`, `\ekv@name@set`, and `\ekv@name@key`. These functions are documented on page 6.)

`\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.

```
117 \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). The `\def\ekv@tmp` gobbles any TeX prefixes which would otherwise throw errors. This will, however, break the package if an `\outer` has been gobbled this way. I consider that good, because keys shouldn’t be defined `\outer` anyways.

```
118 \edef\ekv@checkvalid
119   {%
120     \unexpanded\expandafter{\ekv@ifempty{#1}}%
121     \unexpanded
122     {%
123       \def\ekv@tmp{}%
124       \errmessage{expkv Error: empty set name not allowed}%
125     }}%
126   {%
127     \unexpanded\expandafter{\ekv@ifempty{#2}}%
128     \unexpanded
129     {%
130       {%
131         \def\ekv@tmp{}%
132         \errmessage{expkv Error: empty key name not allowed}%
133       }%
134       \@secondoftwo
135     }%
136   }%
137   \unexpanded{\@gobble}%
138 }
139 \ekv@exparg{\protected\def\ekv@checkvalid#1#2}{\ekv@checkvalid}%
```

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

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

```
\ekvifdefinedNoVal 140 \ekv@expargtwice{\def\ekvifdefined#1#2}%
141   {\expandafter\ekv@ifdefined\expandafter{\ekv@name{#1}{#2}}}
142 \ekv@expargtwice{\def\ekvifdefinedNoVal#1#2}%
143   {\expandafter\ekv@ifdefined\expandafter{\ekv@name{#1}{#2}N}}
```

(End definition for `\ekvifdefined` and `\ekvifdefinedNoVal`. These functions are documented on page 5.)

`\ekvdef` Set up the key defining macros `\ekvdef` etc. We use temporary macros to set these up with a few expansions already done.

```
\ekvlet 144 \def\ekvdef#1#2#3#4%
145   {%
```

`\ekvletkv`

`\ekvletkvNoVal`

`\ekvdefunknown`

`\ekvdefunknownNoVal`

```

146 \protected\long\def\ekvdef##1##2##3%
147   {#1{\expandafter\def\csname#2\endcsname####1{##3}##3}}%
148 \protected\long\def\ekvdefNoVal##1##2##3%
149   {#1{\expandafter\def\csname#2N\endcsname{##3}##3}}%
150 \protected\def\ekvlet##1##2##3%
151   {#1{\expandafter\let\csname#2\endcsname##3##3}}%
152 \protected\def\ekvletNoVal##1##2##3%
153   {#1{\expandafter\let\csname#2N\endcsname##3##3}}%
154 \ekv@expargtwice{\protected\long\def\ekvdefunknown##1##2}%
155   {%
156     \romannumeral
157     \expandafter\ekv@exparg@\expandafter
158     {%
159       \expandafter\expandafter\expandafter
160       \def\expandafter\csname\ekv@name{##1}{u}\endcsname####1####2{##2}%
161       #3%
162     }%
163     {\ekv@zero\ekv@checkvalid{##1}.}%
164   }%
165 \ekv@expargtwice{\protected\long\def\ekvdefunknownNoVal##1##2}%
166   {%
167     \romannumeral
168     \expandafter\ekv@exparg@\expandafter
169     {%
170       \expandafter\expandafter\expandafter
171       \def\expandafter\csname\ekv@name{##1}{uN}\endcsname####1{##2}%
172       #3%
173     }%
174     {\ekv@zero\ekv@checkvalid{##1}.}%
175   }%
176 \protected\def\ekvletkv##1##2##3##4%
177   {%
178     #1%
179     {%
180       \expandafter\let\csname#2\expandafter\endcsname
181       \csname#4\endcsname
182     }%
183     #3%
184   }%
185 \protected\def\ekvletkvNoVal##1##2##3##4%
186   {%
187     #1%
188     {%
189       \expandafter\let\csname#2N\expandafter\endcsname
190       \csname#4N\endcsname
191     }%
192     #3%
193   }%
194 }
195 \edef\ekvdefNoVal
196   {%
197     {\unexpanded\expandafter{\ekv@checkvalid{##1}{##2}}}%
198     {\unexpanded\expandafter{\ekv@name{##1}{##2}}}%
199     {%

```

```

200     \unexpanded{\expandafter\ekv@defsetmacro\csname}%
201     \unexpanded\expandafter{\ekv@undefined@set{#1}\endcsname{#1}}%
202   }%
203   {\unexpanded\expandafter{\ekv@name{#3}{#4}}}%
204 }%
205 \expandafter\ekvdef\ekvdefNoVal

```

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

`\ekv@defsetmacro` 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.

```

206 \edef\ekv@defsetmacro
207   {%
208     \unexpanded{\ifx#1\relax\edef#1##1}%
209     {%
210       \unexpanded\expandafter{\ekv@name@set{#2}}%
211       \unexpanded\expandafter{\ekv@name@key{##1}}%
212     }%
213     \unexpanded{\fi}%
214   }
215 \ekv@exparg{\protected\def\ekv@defsetmacro#1#2}{\ekv@defsetmacro}

```

(End definition for `\ekv@defsetmacro`.)

`\ekvifdefinedset`

```

216 \ekv@expargtwice{\def\ekvifdefinedset#1}%
217   {\expandafter\ekv@ifdefined\expandafter{\ekv@undefined@set{#1}}}

```

(End definition for `\ekvifdefinedset`. This function is documented on page 5.)

`\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 the active commas can be handled by just doing two comma-splitting loops one at a time. 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  $=_{13}$ .

```

218 \begingroup
219 \def\ekvset#1#2{%
220 \endgroup
221 \ekv@exparg{\long\def\ekvset##1##2}%
222   {%
223     \expandafter\expandafter\expandafter
224     \ekv@set\expandafter\csname\ekv@undefined@set{##1}\endcsname
225     \ekv@mark##2#1\ekv@stop#1}%
226   }

```

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

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

```

227 \long\def\ekv@set##1##2#1%
228   {%

```

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

```

229   \ekv@gobble@from@mark@to@stop##2\ekv@endset\ekv@stop

```

else go on with other commas.

```
230     \ekv@set@other##1##2,\ekv@stop,%  
231   }
```

(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.

```
232 \long\def\ekv@endset  
233     \ekv@stop\ekv@set@other##1\ekv@mark\ekv@stop,\ekv@stop,##2%  
234   {##2}
```

(End definition for \ekv@endset.)

`\ekv@eq@other` Splitting at equal signs will be done in a way that checks whether there is an equal sign and splits at the same time. This gets quite messy and the code might look complicated, but this is pretty fast (faster than first checking for an equal sign and splitting if one is found). The splitting code will be adapted for `\ekvset` and `\ekvparse` to get the most speed, but some of these macros don't require such adaptations. `\ekv@eq@other` and `\ekv@eq@active` will split the argument at the first equal sign and insert the macro which comes after the first following `\ekv@mark`. This allows for fast branching based on T<sub>E</sub>X's argument grabbing rules and we don't have to split after the branching if the equal sign was there.

```
235 \long\def\ekv@eq@other##1=##2\ekv@mark##3{##3##1\ekv@stop\ekv@mark##2}  
236 \long\def\ekv@eq@active##1#2##2\ekv@mark##3{##3##1\ekv@stop\ekv@mark##2}
```

(End definition for \ekv@eq@other and \ekv@eq@active.)

`\ekv@set@other` The macro `\ekv@set@other` is guaranteed to get only single `(key)=(value)` pairs.

```
237 \long\def\ekv@set@other##1##2,%  
238   {%
```

First we test whether we're done.

```
239     \ekv@gobble@from@mark@to@stop##2\ekv@endset@other\ekv@stop
```

If not we split at the equal sign of category other.

```
240     \ekv@eq@other##2\ekv@nil\ekv@mark\ekv@set@eq@other@a  
241     =\ekv@mark\ekv@set@eq@active
```

And insert the set name for the next recursion step of `\ekv@set@other`.

```
242     ##1%  
243     \ekv@mark  
244   }
```

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

`\ekv@set@eq@other@a` The first of these two macros runs the split-test for equal signs of category active. It will only be inserted if the `(key)=(value)` pair contained at least one equal sign of category other and `##1` will contain everything up to that equal sign.

```
245 \long\def\ekv@set@eq@other@a##1\ekv@stop  
246   {%  
247     \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@set@eq@other@active  
248     #2\ekv@mark\ekv@set@eq@other@b  
249   }
```

The second macro will have been called by `\ekv@eq@active` if no active equal sign was found. All it does is remove the excess tokens of that test and forward the  $\langle key \rangle = \langle value \rangle$  pair to `\ekv@set@pair`. Normally we would have to also gobble an additional `\ekv@mark` after `\ekv@stop`, but this mark is needed to delimit `\ekv@set@pair`'s argument anyway, so we just leave it there.

```

250 \ekv@exparg
251   {%
252     \long\def\ekv@set@eq@other@b
253       ##1\ekv@nil\ekv@mark\ekv@set@eq@other@active\ekv@stop\ekv@mark
254       ##2\ekv@nil=\ekv@mark\ekv@set@eq@active
255   }%
256   {\ekv@strip{##1}{\expandafter\ekv@set@pair\detokenize}\ekv@mark##2\ekv@nil}

```

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

`\ekv@set@eq@other@active` `\ekv@set@eq@other@active` will be called if the  $\langle key \rangle = \langle value \rangle$  pair was wrongly split on an equal sign of category other but has an earlier equal sign of category active. `##1` will be the contents up to the active equal sign and `##2` everything that remains until the first found other equal sign. It has to reinsert the equal sign and forward things to `\ekv@set@pair`.

```

257 \ekv@exparg
258   {%
259     \long\def\ekv@set@eq@other@active
260       ##1\ekv@stop##2\ekv@nil#2\ekv@mark
261       \ekv@set@eq@other@b\ekv@mark##3=\ekv@mark\ekv@set@eq@active
262   }%
263   {\ekv@strip{##1}{\expandafter\ekv@set@pair\detokenize}\ekv@mark##2=##3}

```

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

`\ekv@set@eq@active` `\ekv@set@eq@active@` `\ekv@set@eq@active@` `\ekv@set@eq@active@` will be called when there was no equal sign of category other in the  $\langle key \rangle = \langle value \rangle$  pair. It removes the excess tokens of the prior test and split-checks for an active equal sign.

```

264 \long\def\ekv@set@eq@active
265   ##1\ekv@nil\ekv@mark\ekv@set@eq@other@a\ekv@stop\ekv@mark
266   {%
267     \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@set@eq@active@
268     #2\ekv@mark\ekv@set@noeq
269   }

```

If an active equal sign was found in `\ekv@set@eq@active` we'll have to pass the now split  $\langle key \rangle = \langle value \rangle$  pair on to `\ekv@set@pair`.

```

270 \ekv@exparg
271   {\long\def\ekv@set@eq@active@##1\ekv@stop##2\ekv@nil#2\ekv@mark\ekv@set@noeq}%
272   {\ekv@strip{##1}{\expandafter\ekv@set@pair\detokenize}\ekv@mark##2\ekv@nil}

```

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

`\ekv@set@noeq` If no active equal sign was found by `\ekv@set@eq@active` there is no equal sign contained in the parsed list entry. In that case we have to check whether the entry is blank in order to ignore it (in which case we'll have to gobble the set-name which was put after these tests by `\ekv@set@other`). Else this is a NoVal key and the entry is passed on to `\ekv@set@key`.

```

273 \edef\ekv@set@noeq
274   {%
275     \unexpanded
276     {%
277       \ekv@ifblank@##1\ekv@nil\ekv@ifempty@B\ekv@set@was@blank
278       \ekv@ifempty@A\ekv@ifempty@B
279     }%
280     \unexpanded\expandafter
281     {\ekv@strip{##1}{\expandafter\ekv@set@key\detokenize}\ekv@mark}%
282   }
283 \ekv@exparg
284   {%
285     \long\def\ekv@set@noeq
286       ##1\ekv@nil\ekv@mark\ekv@set@eq@active@\ekv@stop\ekv@mark
287     }%
288     {\ekv@set@noeq}
289 \expandafter\def\expandafter\ekv@set@was@blank
290   \expandafter\ekv@ifempty@A\expandafter\ekv@ifempty@B
291   \ekv@strip{\ekv@mark##1}##2\ekv@mark
292   {\ekv@set@other}

```

(End definition for \ekv@set@noeq.)

`\ekv@endset@other` All that's left for \ekv@set@other is the macro which breaks the recursion loop at the end. This is done by gobbling all the remaining tokens.

```

293 \long\def\ekv@endset@other
294   \ekv@stop
295   \ekv@eq@other\ekv@mark\ekv@stop\ekv@nil\ekv@mark\ekv@set@eq@other@a
296   =\ekv@mark\ekv@set@eq@active
297   {\ekv@set}

```

(End definition for \ekv@endset@other.)

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

```

\ekvbreakPreSneak
\ekvbreakPostSneak
298 \long\def\ekvbreak##1##2\ekv@stop#1##3{##1}
299 \long\def\ekvbreakPreSneak ##1##2\ekv@stop#1##3{##1##3}
300 \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` One last thing we want to do for \ekvset is to provide macros that just smuggle stuff after \ekvset's effects.

```

\ekvsneakPre
301 \long\def\ekvsneak##1##2\ekv@stop#1##3{##2\ekv@stop#1{##3##1}}
302 \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.

```

303 \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

```
304 \long\def\ekv@parse##1##2##3#1%
305   {%
306     \ekv@gobble@from@mark@to@stop##3\ekv@endparse\ekv@stop
307     \ekv@parse@other{##1}{##2}##3,\ekv@stop,%
308   }
```

(End definition for \ekv@parse.)

\ekv@endparse

```
309 \long\def\ekv@endparse
310   \ekv@stop\ekv@parse@other##1\ekv@mark\ekv@stop,\ekv@stop,%
311   {}
```

(End definition for \ekv@endparse.)

\ekv@parse@other

```
312 \long\def\ekv@parse@other##1##2##3,%
313   {%
314     \ekv@gobble@from@mark@to@stop##3\ekv@endparse@other\ekv@stop
315     \ekv@eq@other##3\ekv@nil\ekv@mark\ekv@parse@eq@other@a
316     =\ekv@mark\ekv@parse@eq@active
317     {##1}{##2}%
318     \ekv@mark
319   }
```

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

\ekv@parse@eq@other@a

\ekv@parse@eq@other@b

```
320 \long\def\ekv@parse@eq@other@a##1\ekv@stop
321   {%
322     \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@parse@eq@other@active
323     #2\ekv@mark\ekv@parse@eq@other@b
324   }
325 \ekv@exparg
326   {%
327     \long\def\ekv@parse@eq@other@b
328       ##1\ekv@nil\ekv@mark\ekv@parse@eq@other@active\ekv@stop\ekv@mark
329       ##2\ekv@nil=\ekv@mark\ekv@parse@eq@active
330     }%
331     {\ekv@strip{##1}\ekv@parse@pair##2\ekv@nil}
```

(End definition for \ekv@parse@eq@other@a and \ekv@parse@eq@other@b.)

\ekv@parse@eq@other@active

```
332 \ekv@exparg
333   {%
334     \long\def\ekv@parse@eq@other@active
335       ##1\ekv@stop##2\ekv@nil#2\ekv@mark
336       \ekv@parse@eq@other@b\ekv@mark##3=\ekv@mark\ekv@parse@eq@active
337     }%
338     {\ekv@strip{##1}\ekv@parse@pair##2=##3}
```

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

\ekv@parse@eq@active  
\ekv@parse@eq@active@

```
339 \long\def\ekv@parse@eq@active
340   ##1\ekv@nil\ekv@mark\ekv@parse@eq@other@a\ekv@stop\ekv@mark
341   {%
342     \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@parse@eq@active@
343     #2\ekv@mark\ekv@parse@noeq
344   }
345 \ekv@exparg
346 {\long\def\ekv@parse@eq@active@##1\ekv@stop##2#2\ekv@mark\ekv@parse@noeq}%
347 {\ekv@strip{##1}\ekv@parse@pair##2}
```

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

\ekv@parse@noeq

```
348 \edef\ekv@parse@noeq
349   {%
350     \unexpanded
351     {%
352       \ekv@ifblank@##1\ekv@nil\ekv@ifempty@B\ekv@parse@was@blank
353       \ekv@ifempty@A\ekv@ifempty@B
354     }%
355     \unexpanded\expandafter{\ekv@strip{##1}\ekv@parse@key}%
356   }
357 \ekv@exparg
358 {%
359   \long\def\ekv@parse@noeq
360     ##1\ekv@nil\ekv@mark\ekv@parse@eq@active@\ekv@stop\ekv@mark
361   }%
362   {\ekv@parse@noeq}
363 \expandafter\def\expandafter\ekv@parse@was@blank
364   \expandafter\ekv@ifempty@A\expandafter\ekv@ifempty@B
365   \ekv@strip{\ekv@mark##1}\ekv@parse@key
366   {\ekv@parse@other}
```

(End definition for \ekv@parse@noeq.)

\ekv@endparse@other

```
367 \long\def\ekv@endparse@other
368   \ekv@stop
369   \ekv@eq@other\ekv@mark\ekv@stop\ekv@nil\ekv@mark\ekv@parse@eq@other@a
370   =\ekv@mark\ekv@parse@eq@active
371   {\ekv@parse}
```

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

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

```
372 \ekv@exparg{\long\def\ekv@parse@pair##1##2\ekv@nil}%
373   {\ekv@strip{##2}\ekv@parse@pair@{##1}}
374 \long\def\ekv@parse@pair@##1##2##3##4%
375   {%
376     \unexpanded{##4{##2}{##1}}%
377     \ekv@parse@other{##3}{##4}%
378   }
```

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

`\ekv@parse@key`

```
379 \long\def\ekv@parse@key##1##2%
380   {%
381     \unexpanded{##2{##1}}%
382     \ekv@parse@other{##2}%
383   }
```

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

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

```
384 }
385 \catcode'\,=13
386 \catcode'\==13
387 \ekvset,=
```

**`\ekvsetSneaked`** This macro can be defined just by expanding `\ekvsneak` once after expanding `\ekvset`. To expand everything as much as possible early on we use a temporary definition.

```
388 \edef\ekvsetSneaked
389   {%
390     \unexpanded{\ekvsneak{#2}}%
391     \unexpanded\expandafter{\ekvset{#1}{#3}}%
392   }
393 \ekv@expargtwice{\long\def\ekvsetSneaked#1#2#3}{\ekvsetSneaked}
```

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

**`\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.

```
394 \ekv@exparg{\def\ekvchangeset#1}%
395   {%
396     \expandafter\expandafter\expandafter
397     \ekv@changeset\expandafter\csname\ekv@undefined@set{#1}\endcsname\ekv@empty
398   }
```

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

`\ekv@changeset` This macro does the real change-out of `\ekvchangeset`. #2 will have a leading `\ekv@empty` so that braces aren't stripped accidentally, but that will not hurt and just expand to nothing in one step.

```
399 \long\def\ekv@changeset#1#2\ekv@set@other#3{#2\ekv@set@other#1}
```

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

`\ekv@set@pair` `\ekv@set@pair` gets invoked with the space and brace stripped and `\detokenized` key-name as its first, the value as the second, and the set name as the third argument. It provides tests for the key-macros and everything to be able to throw meaningful error messages if it isn't defined. We have two routes here, one if `\lastnamedcs` is defined and one if it isn't. The big difference is that if it is we can omit a `\csname` and instead just expand `\lastnamedcs` once to get the control sequence. If the macro is defined the value

will be space and brace stripped and the key-macro called. Else branch into the error handling provided by `\ekv@set@pair`.

```

400 \ekv@if@lastnamedcs
401   {%
402     \long\def\ekv@set@pair#1\ekv@mark#2\ekv@nil#3%
403       {%
404         \ifcsname #3{#1}\endcsname\ekv@set@pair@a\fi\@secondoftwo
405         {#2}%
406         {%
407           \ifcsname #3{u}\endcsname\ekv@set@pair@a\fi\@secondoftwo
408           {#2}%
409           {%
410             \ekv@ifdefined{#3{#1}N}%
411             \ekv@err@noarg
412             \ekv@err@unknown
413             #3%
414           }%
415         {#1}%
416       }%
417     \ekv@set@other#3%
418   }
419 \def\ekv@set@pair@a\fi\@secondoftwo
420   {\fi\expandafter\ekv@set@pair@b\lastnamedcs}
421 }
422 {%
423 \long\def\ekv@set@pair#1\ekv@mark#2\ekv@nil#3%
424   {%
425     \ifcsname #3{#1}\endcsname
426     \ekv@set@pair@a\fi\ekv@set@pair@c#3{#1}\endcsname
427     {#2}%
428     {%
429       \ifcsname #3{u}\endcsname
430       \ekv@set@pair@a\fi\ekv@set@pair@c#3{u}\endcsname
431       {#2}%
432       {%
433         \ekv@ifdefined{#3{#1}N}%
434         \ekv@err@noarg
435         \ekv@err@unknown
436         #3%
437       }%
438     {#1}%
439   }%
440 \ekv@set@other#3%
441 }
442 \def\ekv@set@pair@a\fi\ekv@set@pair@c{\fi\expandafter\ekv@set@pair@b\csname}
443 \long\def\ekv@set@pair@c#1\endcsname#2#3{#3}
444 }
445 \long\def\ekv@set@pair@b#1%
446   {%
447     \ifx#1\relax
448     \ekv@set@pair@e
449     \fi
450     \ekv@set@pair@d#1%
451   }

```

```

452 \ekv@exparg{\long\def\ekv@set@pair@d#1#2#3}{\ekv@strip{#2}#1}
453 \long\def\ekv@set@pair@e\fi\ekv@set@pair@d#1#2#3{\fi#3}

```

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

\ekv@set@key Analogous to \ekv@set@pair, \ekv@set@key builds the NoVal key-macro and provides an error-branch. \ekv@set@key@a will test whether the key-macro is defined and if so call it, else the errors are thrown.

```

\ekv@set@key@a
\ekv@set@key@b
\ekv@set@key@c
454 \ekv@if@lastnamedcs
455   {%
456   \long\def\ekv@set@key#1\ekv@mark#2%
457   {%
458     \ifcsname #2{#1}N\endcsname\ekv@set@key@a\fi\@firstofone
459     {%
460       \ifcsname #2{u}N\endcsname\ekv@set@key@a\fi\@firstofone
461       {%
462         \ekv@ifdefined{#2{#1}}%
463         \ekv@err@reqval
464         \ekv@err@unknown
465         #2%
466       }%
467     }%
468   }%
469   \ekv@set@other#2%
470 }
471 \def\ekv@set@key@a\fi\@firstofone{\fi\expandafter\ekv@set@key@b\lastnamedcs}
472 }
473 {%
474 \long\def\ekv@set@key#1\ekv@mark#2%
475 {%
476   \ifcsname #2{#1}N\endcsname
477   \ekv@set@key@a\fi\ekv@set@key@c#2{#1}N\endcsname
478   {%
479     \ifcsname #2{u}N\endcsname
480     \ekv@set@key@a\fi\ekv@set@key@c#2{u}N\endcsname
481     {%
482       \ekv@ifdefined{#2{#1}}%
483       \ekv@err@reqval
484       \ekv@err@unknown
485       #2%
486     }%
487   }%
488 }%
489 \ekv@set@other#2%
490 }
491 \def\ekv@set@key@a\fi\ekv@set@key@c{\fi\expandafter\ekv@set@key@b\csname}
492 \long\def\ekv@set@key@c#1N\endcsname#2{#2}
493 }
494 \long\def\ekv@set@key@b#1%
495 {%
496   \ifx#1\relax
497   \ekv@fi@secondoftwo
498   \fi
499   \@firstoftwo#1%

```

```
500 }
```

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

**\ekvsetdef** Provide a macro to define a shorthand to use \ekvset on a specified *<set>*. To gain the maximum speed \ekvset is expanded twice by \ekv@exparg so that during runtime the macro storing the set name is already built and one \expandafter doesn't have to be used.

```
501 \ekv@expargtwice{\protected\def\ekvsetdef#1#2}%
502   {%
503     \romannumeral
504     \ekv@exparg{\ekv@zero\ekv@exparg{\long\def#1##1}}%
505     {\ekvset{#2}{##1}}%
506   }
```

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

**\ekvsetSneakeddef** And do the same for \ekvsetSneaked in the two possible ways, with a fixed sneaked argument and with a flexible one.

**\ekvsetdefSneaked**

```
507 \ekv@expargtwice{\protected\def\ekvsetSneakeddef#1#2}%
508   {%
509     \romannumeral
510     \ekv@exparg{\ekv@zero\ekv@exparg{\long\def#1##1##2}}%
511     {\ekvsetSneaked{#2}{##1}{##2}}%
512   }
513 \ekv@expargtwice{\protected\def\ekvsetdefSneaked#1#2#3}%
514   {%
515     \romannumeral
516     \ekv@exparg{\ekv@zero\ekv@exparg{\long\def#1##1}}%
517     {\ekvsetSneaked{#2}{#3}{##1}}%
518   }
```

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

**\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.

**\ekv@err@**

```
519 \begingroup
520 \edef\ekv@err
521   {%
522     \endgroup
523     \unexpanded{\long\def\ekv@err}##1%
524     {%
525       \unexpanded{\expandafter\ekv@err@\@firstofone}%
526       {\unexpanded{\expandafter{\csname ! expkv Error:\endcsname}##1.}}%
527       \unexpanded{\ekv@stop}%
528     }%
529   }
530 \ekv@err
531 \def\ekv@err@{\expandafter\ekv@gobbleto@stop}
```

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

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

```
\ekv@err@unknown 532 \long\def\ekv@err@common #1#2{\expandafter\ekv@err@common@\string#2{#1}}  
  \ekv@err@noarg 533 \long\def\ekv@err@common@#1'#2' #3.#4#5{\ekv@err{#4 ('#5', set '#2')}}  
  \ekv@err@reqval 534 \ekv@exparg{\long\def\ekv@err@unknown#1}{\ekv@err@common{unknown key}{#1}}  
 535 \ekv@exparg{\long\def\ekv@err@noarg #1}{\ekv@err@common{value forbidden}{#1}}  
 536 \ekv@exparg{\long\def\ekv@err@reqval #1}{\ekv@err@common{value required}{#1}}
```

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

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

```
537 \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.

<b>E</b>	
<code>\ekvbreak</code> .....	<i>5</i> , <u>298</u>
<code>\ekvbreakPostSneak</code> .....	<i>5</i> , <u>298</u>
<code>\ekvbreakPreSneak</code> .....	<i>5</i> , <u>298</u>
<code>\ekvchangeset</code> .....	<i>5</i> , <u>394</u>
<code>\ekvDate</code> .....	<i>4</i> , <i>5</i> , <i>8</i> , <u>18</u>
<code>\ekvdef</code> .....	<i>2</i> , <u>144</u>
<code>\ekvdefNoVal</code> .....	<i>2</i> , <u>144</u>
<code>\ekvdefunknown</code> .....	<i>3</i> , <u>144</u>
<code>\ekvdefunknownNoVal</code> .....	<i>3</i> , <u>144</u>
<code>\ekvifdefined</code> .....	<i>5</i> , <u>140</u>
<code>\ekvifdefinedNoVal</code> .....	<i>5</i> , <u>140</u>
<code>\ekvifdefinedset</code> .....	<i>5</i> , <u>216</u>
<code>\ekvlet</code> .....	<i>3</i> , <u>144</u>
<code>\ekvletkv</code> .....	<i>3</i> , <u>144</u>
<code>\ekvletkvNoVal</code> .....	<i>3</i> , <u>144</u>
<code>\ekvletNoVal</code> .....	<i>3</i> , <u>144</u>
<code>\ekvparse</code> .....	<i>4</i> , <u>303</u>
<code>\ekvset</code> .....	<i>3</i> , <u>218</u> , <i>387</i> , <i>391</i> , <u>505</u>
<code>\ekvsetdef</code> .....	<i>4</i> , <u>501</u>
<code>\ekvsetdefSneaked</code> .....	<i>4</i> , <u>507</u>
<code>\ekvsetSneaked</code> .....	<i>3</i> , <u>388</u> , <i>511</i> , <u>517</u>
<code>\ekvsetSneakeddef</code> .....	<i>4</i> , <i>4</i> , <u>507</u>
<code>\ekvsneak</code> .....	<i>5</i> , <u>301</u> , <u>390</u>
<code>\ekvsneakPre</code> .....	<i>5</i> , <u>301</u>
<code>\ekvVersion</code> .....	<i>4</i> , <i>5</i> , <i>8</i> , <u>18</u>
<b>R</b>	
<code>\romannumeral</code> .....	<i>156</i> , <i>167</i> , <i>503</i> , <i>509</i> , <u>515</u>
<b>T</b>	
T <sub>E</sub> X and L <sup>A</sup> T <sub>E</sub> X <sub>2<math>\epsilon</math></sub> commands:	
<code>\@firstofone</code> .....	..... ..... <i>35</i> , <i>53</i> , <i>54</i> , <i>57</i> , <i>458</i> , <i>460</i> , <i>471</i> , <u>525</u>
<code>\@firstoftwo</code> .....	<i>35</i> , <i>52</i> , <i>74</i> , <i>83</i> , <i>87</i> , <u>499</u>
<code>\@gobble</code> .....	<i>35</i> , <u>137</u>
<code>\@secondoftwo</code> .....	<i>35</i> , <i>48</i> , <i>51</i> , <i>62</i> , <i>67</i> , <i>68</i> , <i>134</i> , <i>404</i> , <i>407</i> , <i>419</i>
<code>\ekv@changeset</code> .....	<i>397</i> , <u>399</u>
<code>\ekv@checkvalid</code> .....	<i>118</i> , <i>163</i> , <i>174</i> , <u>197</u>
<code>\ekv@defsetmacro</code> .....	<i>200</i> , <u>206</u>
<code>\ekv@empty</code> .....	<i>34</i> , <u>397</u>
<code>\ekv@endparse</code> .....	<i>306</i> , <u>309</u>
<code>\ekv@endparse@other</code> .....	<i>314</i> , <u>367</u>
<code>\ekv@endset</code> .....	<i>229</i> , <u>232</u>
<code>\ekv@endset@other</code> .....	<i>239</i> , <u>293</u>
<code>\ekv@eq@active</code> .....	<i>235</i> , <i>247</i> , <i>267</i> , <i>322</i> , <u>342</u>
<code>\ekv@eq@other</code> .....	<i>235</i> , <i>240</i> , <i>295</i> , <i>315</i> , <u>369</u>
<code>\ekv@err</code> .....	<i>519</i> , <u>533</u>
<code>\ekv@err@</code> .....	<u>519</u>
<code>\ekv@err@common</code> .....	<u>532</u>
<code>\ekv@err@common@</code> .....	<u>532</u>
<code>\ekv@err@noarg</code> .....	<i>411</i> , <i>434</i> , <u>532</u>
<code>\ekv@err@reqval</code> .....	<i>463</i> , <i>483</i> , <u>532</u>
<code>\ekv@err@unknown</code> .....	<i>412</i> , <i>435</i> , <i>464</i> , <i>484</i> , <u>532</u>
<code>\ekv@exparg</code> .....	..... ..... <i>104</i> , <i>116</i> , <i>139</i> , <i>215</i> , <i>221</i> , <i>250</i> , <i>257</i> , <i>270</i> , <i>283</i> , <i>325</i> , <i>332</i> , <i>345</i> , <i>357</i> , <i>372</i> , <i>394</i> , <i>452</i> , <i>504</i> , <i>510</i> , <i>516</i> , <i>534</i> , <i>535</i> , <u>536</u>
<code>\ekv@exparg@</code> .....	<i>104</i> , <i>157</i> , <u>168</u>
<code>\ekv@expargtwice</code> .....	<i>104</i> , <i>140</i> , <i>142</i> , <i>154</i> , <i>165</i> , <i>216</i> , <i>393</i> , <i>501</i> , <i>507</i> , <u>513</u>
<code>\ekv@expargtwice@</code> .....	<u>104</u>
<code>\ekv@fi@firstoftwo</code> .....	<u>35</u>
<code>\ekv@fi@gobble</code> .....	<u>35</u>
<code>\ekv@fi@secondoftwo</code> .....	<i>35</i> , <i>72</i> , <i>81</i> , <i>87</i> , <u>497</u>
<code>\ekv@gobble@from@mark@to@stop</code> ..	..... ..... <i>35</i> , <i>229</i> , <i>239</i> , <i>306</i> , <i>314</i>
<code>\ekv@gobble@mark</code> .....	<u>35</u>
<code>\ekv@gobbleto@stop</code> .....	<i>35</i> , <u>531</u>
<code>\ekv@if@lastnamedcs</code> .....	<i>23</i> , <i>65</i> , <i>400</i> , <u>454</u>
<code>\ekv@ifblank</code> .....	<u>59</u>
<code>\ekv@ifblank@</code> .....	<i>59</i> , <i>277</i> , <u>352</u>
<code>\ekv@ifdef@</code> .....	<i>67</i> , <i>68</i> , <i>80</i> , <u>85</u>
<code>\ekv@ifdef@false</code> .....	<i>80</i> , <i>85</i> , <u>86</u>
<code>\ekv@ifdefined</code> .....	..... ..... <i>65</i> , <i>141</i> , <i>143</i> , <i>217</i> , <i>410</i> , <i>433</i> , <i>462</i> , <u>482</u>
<code>\ekv@ifempty</code> .....	<u>45</u> , <i>120</i> , <i>127</i>
<code>\ekv@ifempty@</code> .....	<u>45</u> , <u>64</u>
<code>\ekv@ifempty@A</code> .....	<i>47</i> , <i>48</i> , <i>50</i> , <i>51</i> , <i>52</i> , <i>53</i> , <i>54</i> , <i>57</i> , <i>62</i> , <i>64</i> , <i>278</i> , <i>290</i> , <i>353</i> , <u>364</u>
<code>\ekv@ifempty@B</code> .....	..... ..... <i>47</i> , <i>48</i> , <i>50</i> , <i>51</i> , <i>52</i> , <i>53</i> , <i>54</i> , <i>57</i> , <i>61</i> , <i>62</i> , <i>277</i> , <i>278</i> , <i>290</i> , <i>352</i> , <i>353</i> , <u>364</u>
<code>\ekv@ifempty@false</code> .....	<u>45</u>
<code>\ekv@ifempty@true</code> .....	<u>45</u> , <u>61</u>

<code>\ekv@ifempty@true@F</code> .....	<a href="#">45</a>	<code>\ekv@set@eq@active@</code> .....	<a href="#">264</a> , <a href="#">286</a>
<code>\ekv@ifempty@true@F@gobble</code> .....	<a href="#">45</a>	<code>\ekv@set@eq@other@a</code> .....	<a href="#">240</a> , <a href="#">245</a> , <a href="#">265</a> , <a href="#">295</a>
<code>\ekv@ifempty@true@F@gobbletwo</code> ...	<a href="#">45</a>	<code>\ekv@set@eq@other@active</code> .....	<a href="#">247</a> , <a href="#">253</a> , <a href="#">257</a>
<code>\ekv@mark</code> .....	<a href="#">43</a> , <a href="#">44</a> , <a href="#">64</a> , <a href="#">96</a> , <a href="#">99</a> , <a href="#">103</a> , <a href="#">225</a> , <a href="#">233</a> , <a href="#">235</a> , <a href="#">236</a> , <a href="#">240</a> , <a href="#">241</a> , <a href="#">243</a> , <a href="#">247</a> , <a href="#">248</a> , <a href="#">253</a> , <a href="#">254</a> , <a href="#">256</a> , <a href="#">260</a> , <a href="#">261</a> , <a href="#">263</a> , <a href="#">265</a> , <a href="#">267</a> , <a href="#">268</a> , <a href="#">271</a> , <a href="#">272</a> , <a href="#">281</a> , <a href="#">286</a> , <a href="#">291</a> , <a href="#">295</a> , <a href="#">296</a> , <a href="#">303</a> , <a href="#">310</a> , <a href="#">315</a> , <a href="#">316</a> , <a href="#">318</a> , <a href="#">322</a> , <a href="#">323</a> , <a href="#">328</a> , <a href="#">329</a> , <a href="#">335</a> , <a href="#">336</a> , <a href="#">340</a> , <a href="#">342</a> , <a href="#">343</a> , <a href="#">346</a> , <a href="#">360</a> , <a href="#">365</a> , <a href="#">369</a> , <a href="#">370</a> , <a href="#">402</a> , <a href="#">423</a> , <a href="#">456</a> , <a href="#">474</a>	<code>\ekv@set@eq@other@b</code> .....	<a href="#">245</a> , <a href="#">261</a>
<code>\ekv@name</code> .....	.. <a href="#">6</a> , <a href="#">109</a> , <a href="#">141</a> , <a href="#">143</a> , <a href="#">160</a> , <a href="#">171</a> , <a href="#">198</a> , <a href="#">203</a>	<code>\ekv@set@key</code> .....	<a href="#">281</a> , <a href="#">454</a>
<code>\ekv@name@key</code> .....	<a href="#">6</a> , <a href="#">109</a> , <a href="#">211</a>	<code>\ekv@set@key@a</code> .....	<a href="#">454</a>
<code>\ekv@name@set</code> .....	<a href="#">6</a> , <a href="#">109</a> , <a href="#">210</a>	<code>\ekv@set@key@b</code> .....	<a href="#">454</a>
<code>\ekv@nil</code> .....	<a href="#">61</a> , <a href="#">95</a> , <a href="#">97</a> , <a href="#">102</a> , <a href="#">103</a> , <a href="#">240</a> , <a href="#">247</a> , <a href="#">253</a> , <a href="#">254</a> , <a href="#">256</a> , <a href="#">260</a> , <a href="#">265</a> , <a href="#">267</a> , <a href="#">271</a> , <a href="#">272</a> , <a href="#">277</a> , <a href="#">286</a> , <a href="#">295</a> , <a href="#">315</a> , <a href="#">322</a> , <a href="#">328</a> , <a href="#">329</a> , <a href="#">331</a> , <a href="#">335</a> , <a href="#">340</a> , <a href="#">342</a> , <a href="#">352</a> , <a href="#">360</a> , <a href="#">369</a> , <a href="#">372</a> , <a href="#">402</a> , <a href="#">423</a>	<code>\ekv@set@key@c</code> .....	<a href="#">454</a>
<code>\ekv@parse</code> .....	<a href="#">303</a> , <a href="#">304</a> , <a href="#">371</a>	<code>\ekv@set@noeq</code> .....	<a href="#">268</a> , <a href="#">271</a> , <a href="#">273</a>
<code>\ekv@parse@eq@active</code> .....	..... <a href="#">316</a> , <a href="#">329</a> , <a href="#">336</a> , <a href="#">339</a> , <a href="#">370</a>	<code>\ekv@set@other</code> .....	<a href="#">230</a> , <a href="#">233</a> , <a href="#">237</a> , <a href="#">292</a> , <a href="#">399</a> , <a href="#">417</a> , <a href="#">440</a> , <a href="#">469</a> , <a href="#">489</a>
<code>\ekv@parse@eq@active@</code> .....	<a href="#">339</a> , <a href="#">360</a>	<code>\ekv@set@pair</code> .....	<a href="#">256</a> , <a href="#">263</a> , <a href="#">272</a> , <a href="#">400</a>
<code>\ekv@parse@eq@other@a</code> .....	..... <a href="#">315</a> , <a href="#">320</a> , <a href="#">340</a> , <a href="#">369</a>	<code>\ekv@set@pair@a</code> .....	<a href="#">400</a>
<code>\ekv@parse@eq@other@active</code> .....	..... <a href="#">322</a> , <a href="#">328</a> , <a href="#">332</a>	<code>\ekv@set@pair@b</code> .....	<a href="#">400</a>
<code>\ekv@parse@eq@other@b</code> .....	<a href="#">320</a> , <a href="#">336</a>	<code>\ekv@set@pair@c</code> .....	<a href="#">400</a>
<code>\ekv@parse@key</code> .....	<a href="#">355</a> , <a href="#">365</a> , <a href="#">379</a>	<code>\ekv@set@pair@d</code> .....	<a href="#">400</a>
<code>\ekv@parse@noeq</code> .....	<a href="#">343</a> , <a href="#">346</a> , <a href="#">348</a>	<code>\ekv@set@pair@e</code> .....	<a href="#">400</a>
<code>\ekv@parse@other</code> .....	..... <a href="#">307</a> , <a href="#">310</a> , <a href="#">312</a> , <a href="#">366</a> , <a href="#">377</a> , <a href="#">382</a>	<code>\ekv@set@was@blank</code> .....	<a href="#">277</a> , <a href="#">289</a>
<code>\ekv@parse@pair</code> ...	<a href="#">331</a> , <a href="#">338</a> , <a href="#">347</a> , <a href="#">372</a>	<code>\ekv@stop</code> .....	..... <a href="#">42</a> , <a href="#">44</a> , <a href="#">225</a> , <a href="#">229</a> , <a href="#">230</a> , <a href="#">233</a> , <a href="#">235</a> , <a href="#">236</a> , <a href="#">239</a> , <a href="#">245</a> , <a href="#">253</a> , <a href="#">260</a> , <a href="#">265</a> , <a href="#">271</a> , <a href="#">286</a> , <a href="#">294</a> , <a href="#">295</a> , <a href="#">298</a> , <a href="#">299</a> , <a href="#">300</a> , <a href="#">301</a> , <a href="#">302</a> , <a href="#">303</a> , <a href="#">306</a> , <a href="#">307</a> , <a href="#">310</a> , <a href="#">314</a> , <a href="#">320</a> , <a href="#">328</a> , <a href="#">335</a> , <a href="#">340</a> , <a href="#">346</a> , <a href="#">360</a> , <a href="#">368</a> , <a href="#">369</a> , <a href="#">527</a>
<code>\ekv@parse@pair@</code> .....	<a href="#">372</a>	<code>\ekv@strip</code> ...	<a href="#">90</a> , <a href="#">256</a> , <a href="#">263</a> , <a href="#">272</a> , <a href="#">281</a> , <a href="#">291</a> , <a href="#">331</a> , <a href="#">338</a> , <a href="#">347</a> , <a href="#">355</a> , <a href="#">365</a> , <a href="#">373</a> , <a href="#">452</a>
<code>\ekv@parse@was@blank</code> .....	<a href="#">352</a> , <a href="#">363</a>	<code>\ekv@strip@a</code> .....	<a href="#">90</a>
<code>\ekv@set</code> .....	<a href="#">224</a> , <a href="#">227</a> , <a href="#">297</a>	<code>\ekv@strip@b</code> .....	<a href="#">90</a>
<code>\ekv@set@eq@active</code> .....	..... <a href="#">241</a> , <a href="#">254</a> , <a href="#">261</a> , <a href="#">264</a> , <a href="#">296</a>	<code>\ekv@strip@c</code> .....	<a href="#">90</a>
		<code>\ekv@tmp</code> .....	<a href="#">1</a> , <a href="#">123</a> , <a href="#">131</a> , <a href="#">537</a>
		<code>\ekv@tmpa</code> .....	<a href="#">24</a> , <a href="#">26</a>
		<code>\ekv@tmpb</code> .....	<a href="#">25</a> , <a href="#">26</a>
		<code>\ekv@undefined@set</code> .....	..... <a href="#">117</a> , <a href="#">201</a> , <a href="#">217</a> , <a href="#">224</a> , <a href="#">397</a>
		<code>\ekv@zero</code> .....	<a href="#">104</a> , <a href="#">163</a> , <a href="#">174</a> , <a href="#">504</a> , <a href="#">510</a> , <a href="#">516</a>
		<code>\z@</code> .....	<a href="#">104</a>