

exp_{kv}

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

Jonathan P. Spratte*

2020-02-22 v0.4

Abstract

exp_{kv} 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. exp_{kv} is pretty fast, but not the fastest available $\langle key \rangle = \langle value \rangle$ 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	4
1.3.3	Comparisons	5
1.4	Examples	6
1.4.1	Standard Use-Case	6
1.4.2	An Expandable $\langle key \rangle = \langle value \rangle$ Macro Using \ekvsneak	8
1.5	Error Messages	9
1.5.1	Load Time	9
1.5.2	Defining Keys	10
1.5.3	Using Keys	10
1.6	License	11
2	Implementation	12
2.1	The L ^A T _E X Package	12
2.2	The Generic Code	12

Index	23
--------------	-----------

*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 L^AT_EX package. To use it, just use one of:

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

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

1.1 Setting up Keys

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`, 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 and must not contain a `\par` or tokens that expand to it – they must be legal inside of `\csname ... \endcsname`.

<code>\ekvdef</code>	<code>\ekvdef{\set}{\key}{\code}</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.

<code>\ekvdefNoVal</code>	<code>\ekvdefNoVal{\set}{\key}{\code}</code>
---------------------------	--

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

<code>\ekvlet</code>	<code>\ekvlet{\set}{\key}{cs}</code>
----------------------	--------------------------------------

Let the value taking $\langle key \rangle$ in $\langle set \rangle$ to $\langle cs \rangle$, there are no checks on $\langle cs \rangle$ enforced.

<code>\ekvletNoVal</code>	<code>\ekvletNoVal{\set}{\key}{cs}</code>
---------------------------	---

Let the no value taking $\langle key \rangle$ in $\langle set \rangle$ to $\langle cs \rangle$, it is not checked whether $\langle cs \rangle$ exists or that it takes no parameter.

<code>\ekvletkv</code>	<code>\ekvletkv{\set}{\key}{\set2}{\key2}</code>
------------------------	--

Let the $\langle key \rangle$ in $\langle set \rangle$ to $\langle key2 \rangle$ in $\langle set2 \rangle$, it is not checked whether that second key exists.

<code>\ekvletkvNoVal</code>	<code>\ekvletkvNoVal{\set}{\key}{\set2}{\key2}</code>
-----------------------------	---

Let the $\langle key \rangle$ in $\langle set \rangle$ to $\langle key2 \rangle$ in $\langle set2 \rangle$, 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 context¹. If you need control over the necessary steps of expansion you can use \expanded around it.

\ekvbreak and \ekvsneak 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 **expl3**, 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{\b**ekvparse**\k\kv{foo = bar, key, baz={zzz}}}

would expand to

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

¹ 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.

1.3 Miscellaneous

1.3.1 Other Macros

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

<code>\ekvVersion</code>
<code>\ekvDate</code>

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

<code>\ekvifdefined</code>
<code>\ekvifdefinedNoVal</code>

<code>\ekvifdefined{<set>}{<key>}{<true>}{<false>}</code>
<code>\ekvifdefinedNoVal{<set>}{<key>}{<true>}{<false>}</code>

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`.

<code>\ekvbreak</code>
<code>\ekvbreakPreSneak</code>
<code>\ekvbreakPostSneak</code>

<code>\ekvbreak{<after>}</code>

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.

<code>\ekvsneak</code>
<code>\ekvsneakPre</code>

<code>\ekvsneak{<after>}</code>

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 [subsection 1.4.2](#).

<code>\ekv@name</code>
<code>\ekv@name@set</code>
<code>\ekv@name@key</code>

<code>\ekv@name{<set>}{<key>}</code>
<code>\ekv@name@set{<set>}</code>
<code>\ekv@name@key{<key>}</code>

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 `bugshidden` features² 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

²Thanks, David!

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 }
```

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 `explkv` is the only fully expandable $\langle key \rangle = \langle value \rangle$ parser. I tried to compare `explkv` to every $\langle key \rangle = \langle value \rangle$ 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).

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 `explkv`. 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 **explkv**, it gets faster than **explkv** at around eight $\langle key \rangle = \langle value \rangle$ 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 **explkv** at around 10 $\langle key \rangle = \langle value \rangle$ 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 $\langle key \rangle = \langle value \rangle$ interface. The macro has a parameter, which is stored in the dimension \ourdim having a default value from its initialization. Now we want to be able to change that dimension with the width key to some specified value. For that we'd do

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	13.5	1.8	7.0	yes	yes	2014-10-28
exp _k ^v	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 9.7×10^{-3}). 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 $p_2 = 31.7$, $p_1 = 3267.4$, and $p_0 = 7177.6$.

```

\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

```

1.4.2 An Expandable $\langle key \rangle = \langle value \rangle$ Macro Using \ekvsneak

Let's set up an expandable macro, that uses a $\langle key \rangle = \langle value \rangle$ 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 $\langle key \rangle = \langle value \rangle$ 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:

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

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 ϵ -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 `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>`

`\ekv<set>(`

1.6 License

Copyright © 2020 Jonathan P. Spratte

This work may be distributed and/or modified under the conditions of the L^AT_EX 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^AT_EX Package

First we set up the L^AT_EX 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.

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

```
9 \begingroup\expandafter\expandafter\expandafter\endgroup
10 \expandafter\ifx\csname numexpr\endcsname\relax
11   \errmessage{expkv requires e-TeX}
12 \expandafter\endinput
13 \fi
14 \expandafter\ifx\csname ekvVersion\endcsname\relax
15 \else
16 \expandafter\endinput
17 \fi
```

We make sure that it's only input once:

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

```
\ekvDate
18 \def\ekvVersion{0.4}
19 \def\ekvDate{2020-02-22}
```

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

If the L^AT_EX 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
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.

<code>\@gobble</code> <code>\@firstofone</code> <code>\@firstoftwo</code> <code>\@secondoftwo</code> <code>\ekv@gobbletostop</code> <code>\ekv@fi@gobble</code> <code>\ekv@fi@secondoftwo</code>	<p>Since branching tests are often more versatile than <code>\if... \else... \fi</code> 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^AT_EX don't use the <code>ekv</code> prefix.</p> <pre> 23 \long\def\@gobble#1{} 24 \long\def\@firstofone#1{#1} 25 \long\def\@firstoftwo#1#2{#1} 26 \long\def\@secondoftwo#1#2{#2} 27 \long\def\ekv@gobbletostop#1\ekv@stop{} 28 \long\def\ekv@fi@gobble\fi\@firstofone#1{\fi} 29 \long\def\ekv@fi@secondoftwo\fi\@firstoftwo#1#2{\fi#2} </pre>
--	---

(End definition for \@gobble and others.)

As you can see `\ekv@gobbletostop` 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.

<code>\ekv@ifempty</code> <code>\ekv@ifempty@</code> <code>\ekv@ifempty@true</code> <code>\ekv@ifempty@false</code> <code>\ekv@ifempty@true@F</code>	<p>We can test for a lot of things building on an if-empty test, so let's 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.</p> <pre> 30 \long\def\ekv@ifempty#1% 31 {% 32 \ekv@ifempty@A\ekv@ifempty@B\ekv@ifempty@true 33 \ekv@ifempty@A\ekv@ifempty@B\@secondoftwo 34 } 35 \long\def\ekv@ifempty@#1\ekv@ifempty@A\ekv@ifempty@B{} 36 \long\def\ekv@ifempty@true\ekv@ifempty@A\ekv@ifempty@B\@secondoftwo#1#2{#1} 37 \long\def\ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo#1#2{#2} 38 \long\def\ekv@ifempty@true@F\ekv@ifempty@A\ekv@ifempty@B\@firstofone#1{} </pre>
--	--

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

<code>\ekv@ifblank@</code>	<p>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 <code>\ekv@mark</code> token that is to be ignored.</p>
----------------------------	--

```

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

```

(End definition for \ekv@ifblank@.)

<code>\ekv@ifdefined</code>	<p>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.</p>
-----------------------------	---

```

40 \def\ekv@ifdefined#1%
41   {%
42     \expandafter
43     \ifx\csname\ifcsname #1\endcsname #1\else relax\fi\endcsname\relax
44     \ekv@fi@secondoftwo
45   \fi

```

```

46     \@firstoftwo
47 }

```

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

`\ekv@ifdefined@pair` 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).

```

\ekv@ifdefined@pair@
\ekv@ifdefined@key@
\ekv@ifdefined@key@
48 \def\ekv@ifdefined@pair#1#2%
49 {%
50     \expandafter\ekv@ifdefined@pair@
51     \csname
52         \ifcsname #1{#2}\endcsname
53         #1{#2}%
54     \else
55         relax%
56     \fi
57     \endcsname
58 }
59 \def\ekv@ifdefined@pair@#1%
60 {%
61     \ifx#1\relax
62         \ekv@fi@secondoftwo
63     \fi
64     \@firstoftwo
65     {\ekv@set@pair@#1\ekv@mark}%
66 }
67 \def\ekv@ifdefined@key#1#2%
68 {%
69     \expandafter\ekv@ifdefined@key@
70     \csname
71         \ifcsname #1{#2}N\endcsname
72         #1{#2}N%
73     \else
74         relax%
75     \fi
76     \endcsname
77 }
78 \def\ekv@ifdefined@key@#1%
79 {%
80     \ifx#1\relax
81         \ekv@fi@secondoftwo
82     \fi
83     \@firstoftwo#1%
84 }

```

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

`\ekv@name` The keys will all follow the same naming scheme, so we define it here.
`\ekv@name@set` 85 `\def\ekv@name#1#2{\ekv@name@set{#1}\ekv@name@key{#2}}`
`\ekv@name@key` 86 `\def\ekv@name@set#1{\ekv#1{}`
87 `\def\ekv@name@key#1{#1}}`

(End definition for `\ekv@name`, `\ekv@name@set`, and `\ekv@name@key`. These functions are documented on page 4.)

`\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).

```

88 \protected\def\ekv@checkvalid#1#2%
89   {%
90     \ekv@ifempty{#1}%
91     {%
92       \def\ekv@tmp{}%
93       \errmessage{expkv Error: empty set name not allowed}%
94     }%
95     {%
96       \ekv@ifempty{#2}%
97       {%
98         \def\ekv@tmp{}%
99         \errmessage{expkv Error: empty key name not allowed}%
100       }%
101       \@secondoftwo
102     }%
103     \@gobble
104   }

```

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

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

```

105 \def\ekvifdefined#1#2{\ekv@ifdefined{\ekv@name{#1}{#2}}}
106 \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` Set up the key defining macros `\ekvdef` etc.
`\ekvdefNoVal`
`\ekvlet`
`\ekvletNoVal`
`\ekvletkv`
`\ekvletkvNoVal`

```

107 \protected\long\def\ekvdef#1#2#3%
108   {%
109     \ekv@checkvalid{#1}{#2}%
110     {%
111       \expandafter\def\csname\ekv@name{#1}{#2}\endcsname##1{#3}%
112       \ekv@defset{#1}%
113     }%
114   }
115 \protected\long\def\ekvdefNoVal#1#2#3%
116   {%
117     \ekv@checkvalid{#1}{#2}%
118     {%
119       \expandafter\def\csname\ekv@name{#1}{#2}N\endcsname{#3}%
120       \ekv@defset{#1}%
121     }%
122   }
123 \protected\def\ekvlet#1#2#3%
124   {%
125     \ekv@checkvalid{#1}{#2}%
126     {%
127       \expandafter\let\csname\ekv@name{#1}{#2}\endcsname#3%
128       \ekv@defset{#1}%
129     }%
130   }
131 \protected\def\ekvletNoVal#1#2#3%

```

```

132   {%
133     \ekv@checkvalid{#1}{#2}%
134     {%
135       \expandafter\let\csname\ekv@name{#1}{#2}N\endcsname#3%
136       \ekv@defset{#1}%
137     }%
138   }
139 \protected\def\ekvletkv#1#2#3#4%
140   {%
141     \ekv@checkvalid{#1}{#2}%
142     {%
143       \expandafter\let\csname\ekv@name{#1}{#2}\expandafter\endcsname
144       \csname\ekv@name{#3}{#4}\endcsname
145       \ekv@defset{#1}%
146     }%
147   }
148 \protected\def\ekvletkvNoVal#1#2#3#4%
149   {%
150     \ekv@checkvalid{#1}{#2}%
151     {%
152       \expandafter\let\csname\ekv@name{#1}{#2}N\expandafter\endcsname
153       \csname\ekv@name{#3}{#4}N\endcsname
154       \ekv@defset{#1}%
155     }%
156   }

```

(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.

```

157 \protected\def\ekv@defset#1%
158   {%
159     \expandafter\edef\csname\ekv@name@set{#1}\endcsname##1%
160     {\ekv@name@set{#1}\ekv@name@key{##1}}%
161   }

```

(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.

```

162 \def\ekvset#1#2{%
163   \endgroup
164   \long\def\ekvset##1##2%
165     {%
166       \expandafter\ekv@set\csname\ekv@name@set{##1}\endcsname
167       \ekv@mark##2#1\ekv@stop#1}%
168   }

```

(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.

```
169 \long\def\ekv@set##1##2#1%
170 {%
```

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

```
171 \ekv@ifstop##2\ekv@endset\ekv@mark\ekv@stop
```

else go on with other commas,

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

and get the next active comma delimited $\langle key \rangle = \langle value \rangle$ pair.

```
173 \ekv@set##1\ekv@mark
174 }
```

(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.

```
175 \long\def\ekv@endset
176 \ekv@mark\ekv@stop\ekv@set@other##1,\ekv@stop,\ekv@set##2\ekv@mark
177 ##3%
178 {##3}
```

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

`\ekv@set@other` The macro `\ekv@set@other` is guaranteed to get only single $\langle key \rangle = \langle value \rangle$ 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.

```
179 \long\def\ekv@set@other##1##2,%
180 {%
181 \ekv@ifstop##2\ekv@endset@other\ekv@mark\ekv@stop
182 \ekv@ifhas@eq@other##2=\ekv@ifempty@B\ekv@ifempty@false
183 \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
184 {\ekv@set@eq@other##1##2\ekv@stop}%
185 {%
186 \ekv@ifhas@eq@active##2\ekv@ifempty@B\ekv@ifempty@false
187 \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
188 {\ekv@set@eq@active##1##2\ekv@stop}%
189 {%
190 \ekv@ifblank@##2\ekv@nil\ekv@ifempty@B\ekv@ifempty@true@F
191 \ekv@ifempty@A\ekv@ifempty@B\@firstofone
192 {\ekv@strip{##2}\ekv@set@key##1}%
193 }%
194 }%
195 \ekv@set@other##1\ekv@mark%
196 }
```

(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`.

```
197 \long\def\ekv@set@eq@other##1##2=%
```

```

198 {%
199   \ekv@ifhas@eq@active##2#2\ekv@ifempty@B\ekv@ifempty@false
200   \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
201   {\ekv@set@eq@active##1##2=}%
202   {\ekv@strip{##2}\ekv@set@pair##1}%
203 }

```

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

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

```

204 \long\def\ekv@set@eq@active##1##2#2%
205 {%
206   \ekv@strip{##2}\ekv@set@pair##1%
207 }

```

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

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

```

\ekv@ifhas@eq@active
\ekv@endset@other
208 \long\def\ekv@ifhas@eq@other\ekv@mark##1={\ekv@ifempty@\ekv@ifempty@A}
209 \long\def\ekv@ifhas@eq@active\ekv@mark##1#2{\ekv@ifempty@\ekv@ifempty@A}
210 \long\def\ekv@endset@other
211   \ekv@mark\ekv@stop
212   \ekv@ifhas@eq@other##1=\ekv@ifempty@B\ekv@ifempty@false
213   \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
214   ##2%
215   \ekv@set@other##3\ekv@mark
216 {}

```

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

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

```

\ekvbreakPreSneak
\ekvbreakPostSneak
217 \long\def\ekvbreak##1##2\ekv@stop#1##3{##1}
218 \long\def\ekvbreakPreSneak ##1##2\ekv@stop#1##3{##1##3}
219 \long\def\ekvbreakPostSneak##1##2\ekv@stop#1##3{##3##1}

```

(End definition for \ekvbreak, \ekvbreakPreSneak, and \ekvbreakPostSneak. These functions are documented on page 4.)

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

```

\ekvsneakPre
220 \long\def\ekvsneak##1##2\ekv@stop#1##3%
221 {%
222   ##2\ekv@stop#1{##3##1}%
223 }
224 \long\def\ekvsneakPre##1##2\ekv@stop#1##3%
225 {%
226   ##2\ekv@stop#1{##1##3}%
227 }

```

(End definition for \ekvsneak and \ekvsneakPre. These functions are documented on page 4.)

\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.

```

228 \long\def\ekvparse##1##2##3%
229 {%
230   \ekv@parse##1##2\ekv@mark##3#1\ekv@stop#1%
231 }

```

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

\ekv@parse

```

232 \long\def\ekv@parse##1##2##3#1%
233 {%
234   \ekv@ifstop##3\ekv@endparse\ekv@mark\ekv@stop
235   \ekv@parse@other##1##2##3,\ekv@stop,%
236   \ekv@parse##1##2\ekv@mark
237 }

```

(End definition for \ekv@parse.)

\ekv@endparse

```

238 \long\def\ekv@endparse
239   \ekv@mark\ekv@stop\ekv@parse@other##1,\ekv@stop,\ekv@parse##2\ekv@mark
240 {}

```

(End definition for \ekv@endparse.)

\ekv@parse@other

```

241 \long\def\ekv@parse@other##1##2##3,%
242 {%
243   \ekv@ifstop##3\ekv@endparse@other\ekv@mark\ekv@stop
244   \ekv@ifhas@eq@other##3=\ekv@ifempty@B\ekv@ifempty@false
245   \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
246   {\ekv@parse@eq@other##3\ekv@stop##2}%
247   {%
248     \ekv@ifhas@eq@active##3#2\ekv@ifempty@B\ekv@ifempty@false
249     \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
250     {\ekv@parse@eq@active##3\ekv@stop##2}%
251     {%
252       \ekv@ifblank###3\ekv@nil\ekv@ifempty@B\ekv@ifempty@true@F
253       \ekv@ifempty@A\ekv@ifempty@B\@firstofone
254       {\ekv@strip{##3}\ekv@parse@key##1}%
255     }%
256   }%
257   \ekv@parse@other##1##2\ekv@mark
258 }

```

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

\ekv@parse@eq@other

```

259 \long\def\ekv@parse@eq@other##1=%
260 {%
261   \ekv@ifhas@eq@active##1#2\ekv@ifempty@B\ekv@ifempty@false

```

```

262     \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
263     {\ekv@parse@eq@active##1=}%
264     {\ekv@strip{##1}\ekv@parse@pair\ekv@mark}%
265 }

```

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

\ekv@parse@eq@active

```

266 \long\def\ekv@parse@eq@active##1#2%
267 {%
268     \ekv@strip{##1}\ekv@parse@pair\ekv@mark
269 }

```

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

\ekv@endparse@other

```

270 \long\def\ekv@endparse@other
271     \ekv@mark\ekv@stop
272     \ekv@ifhas@eq@other##1=\ekv@ifempty@B\ekv@ifempty@false
273     \ekv@ifempty@A\ekv@ifempty@B\@firstoftwo
274     ##2%
275     \ekv@parse@other##3\ekv@mark
276 {}

```

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

\ekv@parse@pair

\ekv@parse@pair@

```

277 \long\def\ekv@parse@pair##1##2\ekv@stop
278 {%
279     \ekv@strip{##2}\ekv@parse@pair@{##1}%
280 }
281 \long\def\ekv@parse@pair@##1##2##3%
282 {%
283     \unexpanded{##3{##2}{##1}}%
284 }

```

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

\ekv@parse@key

```

285 \long\def\ekv@parse@key##1##2%
286 {%
287     \unexpanded{##2{##1}}%
288 }

```

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

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

```

289 }
290 \begingroup
291 \catcode'\,=13
292 \catcode'\==13
293 \ekvset,=

```

\ekv@ifstop

The \ekv@ifstop test works similar to our if-empty test, but instead of using tokens which are used nowhere else (\ekv@ifempty@A and \ekv@ifempty@B) we use \ekv@mark and \ekv@stop.

```

294 \long\def\ekv@ifstop#1\ekv@mark\ekv@stop{}

```

(End definition for \ekv@ifstop.)

\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.

```
295 \long\def\ekv@set@pair#1#2%
296   {%
297     \ekv@ifdefined@pair#2{#1}%
298     {%
```

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.

```
299     \ekv@ifdefined{#2{#1}N}%
300     \ekv@err@noarg
301     \ekv@err@unknown
302     #2{#1}%
303     \ekv@gobbletostop
304   }%
305 }
```

\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.

```
306 \long\def\ekv@set@pair@#1#2\ekv@stop
307   {%
308     \ekv@strip{#2}#1%
309   }
```

(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.

```
310 \long\def\ekv@set@key#1#2%
311   {%
312     \ekv@ifdefined@key#2{#1}%
313     {%
314       \ekv@ifdefined{#2{#1}}%
315       \ekv@err@reqval
316       \ekv@err@unknown
317       #2{#1}%
318     }%
319   }
```

(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.

```
320 \begingroup
321 \edef\ekv@err
322   {%
323   \endgroup
```

```

324 \unexpanded{\long\def\ekv@err}##1%
325 {%
326 \unexpanded{\expandafter\ekv@err@\@firstofone}%
327 {\expandafter\noexpand\csname ! expkv Error:\endcsname ##1.}%
328 \unexpanded{\ekv@stop}%
329 }%
330 }
331 \ekv@err
332 \def\ekv@err@{\expandafter\ekv@gobbletostop}

```

(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 instead of the full strings.

```

\ekv@err@common@ 333 \long\def\ekv@err@common #1#2{\expandafter\ekv@err@common@\string#2{#1}}
\ekv@err@unknown 334 \long\def\ekv@err@common@#1#2#3#4#5(#6#7{\ekv@err{#6 ('#7', set '#5')}}}
\ekv@err@noarg 335 \long\def\ekv@err@unknown#1#2{\ekv@err@common{unknown key}#1{#2}}
\ekv@err@reqval 336 \long\def\ekv@err@noarg #1#2{\ekv@err@common{value forbidden}#1{#2}}
337 \long\def\ekv@err@reqval #1#2{\ekv@err@common{value required}#1{#2}}

```

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

\ekv@strip 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.

```

338 \def\ekv@strip#1%
339 {%
340 \long\def\ekv@strip##1%
341 {%
342 \ekv@strip@a
343 ##1%
344 \ekv@nil
345 \ekv@mark#1%
346 #1\ekv@nil{}%
347 \ekv@stop
348 }%
349 \long\def\ekv@strip@a##1\ekv@mark#1##2\ekv@nil##3%
350 {%
351 \ekv@strip@b##3##1##2\ekv@nil
352 }%
353 \long\def\ekv@strip@b##1#1\ekv@nil
354 {%
355 \ekv@strip@c##1\ekv@nil
356 }%
357 \long\def\ekv@strip@c\ekv@mark##1\ekv@nil##2\ekv@stop##3%
358 {%
359 ##3{##1}%
360 }%
361 }
362 \ekv@strip{ }

```

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

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

```

363 \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` 4, 217
`\ekvbreakPostSneak` 4, 217
`\ekvbreakPreSneak` 4, 217
`\ekvDate` 4, 4, 8, 18
`\ekvdef` 2, 107
`\ekvdefNoVal` 2, 107
`\ekvifdefined` 4, 105
`\ekvifdefinedNoVal` 4, 105
`\ekvlet` 2, 107
`\ekvletkv` 2, 107
`\ekvletkvNoVal` 2, 107
`\ekvletNoVal` 2, 107
`\ekvparse` 3, 228
`\ekvset` 3, 162, 293
`\ekvsneak` 4, 220
`\ekvsneakPre` 4, 220
`\ekvVersion` 4, 4, 8, 18

T

T_EX and L^AT_EX 2_ε commands:

`\@firstofone` 23, 38, 191, 253, 326
`\@firstoftwo` 23, 37, 46, 64, 83,
183, 187, 200, 213, 245, 249, 262, 273
`\@gobble` 23, 103
`\@secondoftwo` 23, 33, 36, 101
`\ekv@checkvalid`
..... 88, 109, 117, 125, 133, 141, 150
`\ekv@defset`
..... 112, 120, 128, 136, 145, 154, 157
`\ekv@endparse` 234, 238
`\ekv@endparse@other` 243, 270
`\ekv@endset` 171, 175
`\ekv@endset@other` 181, 208
`\ekv@err` 320, 334
`\ekv@err@` 320
`\ekv@err@common` 333
`\ekv@err@common@` 333
`\ekv@err@noarg` 300, 333
`\ekv@err@reqval` 315, 333
`\ekv@err@unknown` 301, 316, 333
`\ekv@fi@gobble` 23
`\ekv@fi@secondoftwo` 23, 44, 62, 81

`\ekv@gobbletostop` 23, 303, 332
`\ekv@ifblank@` 39, 190, 252
`\ekv@ifdefined` .. 40, 105, 106, 299, 314
`\ekv@ifdefined@key` 48, 312
`\ekv@ifdefined@key@` 48
`\ekv@ifdefined@pair` 48, 297
`\ekv@ifdefined@pair@` 48
`\ekv@ifempty` 30, 90, 96
`\ekv@ifempty@` 30, 39, 208, 209
`\ekv@ifempty@A` 32, 33,
35, 36, 37, 38, 39, 183, 187, 191, 200,
208, 209, 213, 245, 249, 253, 262, 273
`\ekv@ifempty@B` 32, 33, 35,
36, 37, 38, 182, 183, 186, 187, 190,
191, 199, 200, 212, 213, 244, 245,
248, 249, 252, 253, 261, 262, 272, 273
`\ekv@ifempty@false` 30,
182, 186, 199, 212, 244, 248, 261, 272
`\ekv@ifempty@true` 30
`\ekv@ifempty@true@F` 30, 190, 252
`\ekv@ifhas@eq@active`
..... 186, 199, 208, 248, 261
`\ekv@ifhas@eq@other` 182, 208, 244, 272
`\ekv@ifstop` 171, 181, 234, 243, 294
`\ekv@mark` 39, 65, 167, 171,
173, 176, 181, 195, 208, 209, 211,
215, 230, 234, 236, 239, 243, 257,
264, 268, 271, 275, 294, 345, 349, 357
`\ekv@name` 4, 85, 105, 106,
111, 119, 127, 135, 143, 144, 152, 153
`\ekv@name@key` 4, 85, 160
`\ekv@name@set` 4, 85, 159, 160, 166
`\ekv@nil` 190,
252, 344, 346, 349, 351, 353, 355, 357
`\ekv@parse` 230, 232, 239
`\ekv@parse@eq@active` ... 250, 263, 266
`\ekv@parse@eq@other` 246, 259
`\ekv@parse@key` 254, 285
`\ekv@parse@other` .. 235, 239, 241, 275
`\ekv@parse@pair` 264, 268, 277
`\ekv@parse@pair@` 277
`\ekv@set` 166, 169, 176
`\ekv@set@eq@active` 188, 201, 204
`\ekv@set@eq@other` 184, 197

\ekv@set@key	192, <u>310</u>	250, 271, 277, 294, 306, 328, 347, 357	
\ekv@set@other	172, 176, 179, 215	\ekv@strip	192,
\ekv@set@pair	202, <u>206</u> , <u>295</u>		202, 206, 254, 264, 268, 279, 308, <u>338</u>
\ekv@set@pair@	65, <u>295</u>	\ekv@strip@a	<u>338</u>
\ekv@stop	27,	\ekv@strip@b	<u>338</u>
	167, 171, 172, 176, 181, 184, 188,	\ekv@strip@c	<u>338</u>
	211, 217, 218, 219, 220, 222, 224,	\ekv@tmp	1, 92, 98, 363
	226, 230, 234, 235, 239, 243, 246,		