

exp_{kv}

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

Jonathan P. Spratte*

2020-04-10 V1.2

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 is one and a half times as fast, but not expandable and it might strip braces it shouldn't have stripped).

Contents

1	Documentation	2
1.1	Setting up Keys	2
1.2	Parsing Keys	3
1.3	Miscellaneous	4
1.3.1	Other Macros	4
1.3.2	Bugs	5
1.3.3	Comparisons	5
1.4	Examples	7
1.4.1	Standard Use-Case	7
1.4.2	An Expandable $\langle key \rangle = \langle value \rangle$ Macro Using \ekvsneak	9
1.5	Error Messages	11
1.5.1	Load Time	11
1.5.2	Defining Keys	11
1.5.3	Using Keys	11
1.6	License	12
2	Implementation	13
2.1	The L ^A T _E X Package	13
2.2	The Generic Code	13

Index	26
--------------	-----------

*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`.

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`

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 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.

<hr/> <hr/>	<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.
<hr/> <hr/>	<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.
<hr/> <hr/>	<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

<hr/> <hr/>	<code>\ekvset{<set>}{<key>=<value>,...}</code>
	Splits $\langle key \rangle = \langle value \rangle$ pairs on commas. From both $\langle key \rangle$ and $\langle value \rangle$ up to one space is stripped from both ends, if then only a braced group remains the braces are stripped as well. So <code>\ekvset{foo}{bar=baz}</code> and <code>\ekvset{foo}{ {bar}= {baz} }</code> will both do <code>\{foobarcod\}{baz}</code> , so you can hide commas, equal signs and spaces at the ends of either $\langle key \rangle$ or $\langle value \rangle$ 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 $\langle key \rangle = \langle value \rangle$ 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. <code>\ekvset</code> should be nestable.

<code>\ekvpars</code>	<code>\ekvpars<cs1><cs2>{<key>=<value>,...}</code>
-----------------------	--

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`, `\ekvsneak`, and `\ekvchangeset` and their relatives don't work in `\ekvpars`. 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 `\ekvpars` doesn't deal with.

As a small example:

```
\ekvpars\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 `\ekvpars` (of course the names might differ). If you need the results of `\ekvpars` as the argument for another macro, you should use `\expanded` as only then the input stream will contain the output above:

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

would expand to

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

.

1.3 Miscellaneous

1.3.1 Other Macros

`expl3` provides some other macros which might be of interest.

<code>\ekvVersion</code>	These two macros store the version and date of the package.
<code>\ekvDate</code>	

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

¹ This is a change in behaviour, previously (v0.3 and before) `\ekvpars` 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.

<hr/> <code>\ekvbreak</code>	<code>\ekvbreak{<after>}</code>
<code>\ekvbreakPreSneak</code>	Gobbles the remainder of the current <code>\ekvset</code> macro and its argument list and reinserts <code><after></code> . So this can be used to break out of <code>\ekvset</code> . The first variant will also gobble anything that has been sneaked out using <code>\ekvsneak</code> or <code>\ekvsneakPre</code> , while <code>\ekvbreakPreSneak</code> will put <code><after></code> before anything that has been smuggled and <code>\ekvbreakPostSneak</code> will put <code><after></code> after the stuff that has been sneaked out.
<code>\ekvbreakPostSneak</code> <hr/>	

<hr/> <code>\ekvsneak</code>	<code>\ekvsneak{<after>}</code>
<code>\ekvsneakPre</code> <hr/>	Puts <code><after></code> after the effects of <code>\ekvset</code> . The first variant will put <code><after></code> after any other tokens which might have been sneaked before, while <code>\ekvsneakPre</code> will put <code><after></code> before other smuggled stuff. This reads and reinserts the remainder of the current <code>\ekvset</code> macro and its argument list to do its job. A small usage example is shown in subsection 1.4.2 .

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

<hr/> <code>\ekv@name</code>	<code>\ekv@name{<set>}{<key>}</code>
<code>\ekv@name@set</code>	<code>\ekv@name@set{<set>}</code>
<code>\ekv@name@key</code> <hr/>	<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 <code>\ekv@name@set</code> is “ <code>\ekv{<set>}(</code> ” and the default of <code>\ekv@name@key</code> is “ <code><key>)</code> ”. The complete name is build using <code>\ekv@name</code> which is equivalent to <code>\ekv@name@set{<set>}\ekv@name@key{<key>}</code> . For <code>NoVal</code> keys an additional <code>N</code> gets appended irrespective of these macros’ definition, so their name is <code>\ekv{<set>}(<key>)N</code> . You might redefine <code>\ekv@name@set</code> and <code>\ekv@name@key</code> locally but <i>don’t redefine</i> <code>\ekv@name</code> !

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

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 }
```

²Thanks, David!

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

As far as I know `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 `\ekvpars` and `\keyval_parse:NNn` contained, as most other packages don’t provide equivalent features to my knowledge. `\ekvpars` 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:nm` of `expl3` (where the difference is much bigger).

keyval is about 1.6 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 seventeen 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 370 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 four 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 `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 around six 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. Note that this comparison uses the version contained in T_EXLive 2019 (frozen) which is a bit slower than versions starting with T_EXLive 2020.

pgfkeys is around 2.7 times slower for one key, but has an *enormous* feature set. 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 3.7 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 1.5 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) 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 almost seven 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 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 initialization. Now we want to be able to change that dimension with the width key to some specified value. For that we'd do

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

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

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

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

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	13.4	2.6	6.9	yes	yes	2014-10-28
exp _k ^v	19.7	7.3	9.6	no	no	2020-04-07
options	23.4	15.6	19.9	yes	yes	2015-03-01
pgfkeys	24.6	46.2	52.8	yes	yes	2020-01-08
kvsetkeys	*	*	39.8	no	no	2019-12-15
l3keys	108.0	56.3	50.7	no	no	2020-02-25
simplekv	149.2	25.7	8.1	yes	yes	2017-08-08
xkeyval	248.8	234.4	161.9	yes	yes	2014-12-03
yax	443.6	170.5	115.7	yes	yes	2010-01-22
ltxkeys	3516.0	5254.8	5487.0	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 $p_2 = 7.5$, $p_1 = 51.3$, and $p_0 = 49.6$. 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 3.5×10^{-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 $p_2 = 11.4$, $p_1 = 3276.7$, and $p_0 = 6132.3$.


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

Finally we can use our macro like in the following

```
\ourmacro{\par 150.pt
\ourmacro{width}\par 192.85382pt
\ourmacro{width=5pt}\par 5.pt}
```

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:

<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

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

```
\makeatletter
\ekvcSplitAndForward\sine\sine@
```

```

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

```

The resulting macro will behave just like the one previously defined, but will have an additional `unit` key, since in `expkv`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 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 subsubsection contains the errors thrown during `\ekvset`. The errors are thrown in an expandable manner by providing an undefined macro. In the following messages `<key>` gets replaced with the problematic key's name, and `<set>` with the corresponding set. If any errors during `<key>=<value>` handling are encountered, the entry in the comma separated list will be omitted after the error is thrown and the next `<key>=<value>` pair will be parsed.

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

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

If you're using a key for which only a normal version and no NoVa1 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_EX 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^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.

We make sure that it's only input once:

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

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

```
\ekvDate 18 \def\ekvVersion{1.2}
19 \def\ekvDate{2020-04-10}
```

(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
```

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.

```

\@gobble
\@firstofone
\@firstoftwo
\@secondoftwo
\ekv@gobbleto@stop
\ekv@fi@secondoftwo
\ekv@gobble@mark
\ekv@gobble@from@mark@to@stop
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@fi@secondoftwo\fi\@firstoftwo#1#2{\fi#2}
28 \long\def\ekv@gobbleto@stop#1\ekv@stop{}
29 \def\ekv@gobble@mark\ekv@mark{}
30 \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
\ekv@ifempty@
\ekv@ifempty@true
\ekv@ifempty@false
\ekv@ifempty@true@F
\ekv@ifempty@true@F@gobble
\ekv@ifempty@true@F@gobbletwo
23 \long\def\ekv@ifempty#1%
24 {%
25 \ekv@ifempty@\ekv@ifempty@A#1\ekv@ifempty@B\ekv@ifempty@true
26 \ekv@ifempty@A\ekv@ifempty@B\@secondoftwo
27 }
28 \long\def\ekv@ifempty@#1\ekv@ifempty@A\ekv@ifempty@B{}
29 \long\def\ekv@ifempty@true\ekv@ifempty@A\ekv@ifempty@B\@secondoftwo#1#2{#1}
30 \long\def\ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo#1#2{#2}
31 \long\def\ekv@ifempty@true@F\ekv@ifempty@A\ekv@ifempty@B\@firstofone#1{}
32 \long\def\ekv@ifempty@true@F@gobble\ekv@ifempty@A\ekv@ifempty@B\@firstofone#1#2%
33 {}
34 \long\def\ekv@ifempty@true@F@gobbletwo
35 \ekv@ifempty@A\ekv@ifempty@B\@firstofone#1#2#3%
36 {}
37
38
39
40
41
42
43
44

```

(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.

```

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

```

(End definition for \ekv@ifblank@.)

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

```

46 \def\ekv@ifdefined#1%
47   {%
48     \expandafter
49     \ifx\csname\ifcsname #1\endcsname #1\else relax\fi\endcsname\relax
50     \ekv@fi@secondoftwo
51     \fi
52     \@firstoftwo
53   }

```

(End definition for \ekv@ifdefined.)

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

```

\ekv@name@set 54 \def\ekv@name#1#2{\ekv@name@set{#1}\ekv@name@key{#2}}
\ekv@name@key 55 \def\ekv@name@set#1{\ekv#1{}}
56 \def\ekv@name@key#1{#1}}

```

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

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

```

57 \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.

```

58 \protected\def\ekv@checkvalid#1#2%
59   {%
60     \ekv@ifempty{#1}%
61     {%
62       \def\ekv@tmp{}%
63       \errmessage{expkv Error: empty set name not allowed}%
64     }%
65     {%
66       \ekv@ifempty{#2}%
67       {%
68         \def\ekv@tmp{}%
69         \errmessage{expkv Error: empty key name not allowed}%
70       }%
71       \@secondoftwo
72     }%
73     \@gobble
74   }

```

(End definition for \ekv@checkvalid.)

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

`\ekvifdefinedNoVal`

```

75 \def\ekvifdefined#1#2{\ekv@ifdefined{\ekv@name{#1}{#2}}}
76 \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`

```

77 \protected\long\def\ekvdef#1#2#3%
78   {%
79     \ekv@checkvalid{#1}{#2}%
80     {%
81       \expandafter\def\csname\ekv@name{#1}{#2}\endcsname##1{#3}%
82       \expandafter\ekv@defset\csname\ekv@undefined@set{#1}\endcsname{#1}%
83     }%
84   }
85 \protected\long\def\ekvdefNoVal#1#2#3%
86   {%
87     \ekv@checkvalid{#1}{#2}%
88     {%
89       \expandafter\def\csname\ekv@name{#1}{#2}N\endcsname{#3}%
90       \expandafter\ekv@defset\csname\ekv@undefined@set{#1}\endcsname{#1}%
91     }%
92   }
93 \protected\def\ekvlet#1#2#3%
94   {%
95     \ekv@checkvalid{#1}{#2}%
96     {%
97       \expandafter\let\csname\ekv@name{#1}{#2}\endcsname#3%
98       \expandafter\ekv@defset\csname\ekv@undefined@set{#1}\endcsname{#1}%
99     }%
100  }
101 \protected\def\ekvletNoVal#1#2#3%
102   {%
103     \ekv@checkvalid{#1}{#2}%
104     {%
105       \expandafter\let\csname\ekv@name{#1}{#2}N\endcsname#3%
106       \expandafter\ekv@defset\csname\ekv@undefined@set{#1}\endcsname{#1}%
107     }%
108  }
109 \protected\def\ekvletkv#1#2#3#4%
110   {%
111     \ekv@checkvalid{#1}{#2}%
112     {%
113       \expandafter\let\csname\ekv@name{#1}{#2}\expandafter\endcsname
114       \csname\ekv@name{#3}{#4}\endcsname
115       \expandafter\ekv@defset\csname\ekv@undefined@set{#1}\endcsname{#1}%
116     }%
117  }
118 \protected\def\ekvletkvNoVal#1#2#3#4%
119   {%
120     \ekv@checkvalid{#1}{#2}%
121     {%
122       \expandafter\let\csname\ekv@name{#1}{#2}N\expandafter\endcsname
123       \csname\ekv@name{#3}{#4}N\endcsname

```



```

124         \expandafter\ekv@defset\csname\ekv@undefined@set{#1}\endcsname{#1}%
125     }%
126 }

```

(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.

```

127 \protected\def\ekv@defset#1#2%
128 {%
129     \ifx#1\relax
130         \edef#1##1{\ekv@name@set{#2}\ekv@name@key{##1}}%
131     \fi
132 }

```

(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`.

```

133 \def\ekvset#1#2{%
134 \endgroup
135 \long\def\ekvset##1##2%
136 {%
137     \expandafter\ekv@set\csname\ekv@undefined@set{##1}\endcsname
138     \ekv@mark##2#1\ekv@stop#1}%
139 }

```

(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.

```

140 \long\def\ekv@set##1##2#1%
141 {%

```

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

```

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

```

else go on with other commas,

```

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

```

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

```

144     \ekv@set##1\ekv@mark
145 }

```

(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.

```

146 \long\def\ekv@endset
147     \ekv@stop\ekv@set@other##1,\ekv@stop,\ekv@set##2\ekv@mark
148     ##3%
149     {##3}

```

(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
\ekv@eq@active 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_EX's argument grabbing rules and we don't have to split after the branching if the equal
sign was there.

```

150 \long\def\ekv@eq@other##1=##2\ekv@mark##3##4\ekv@stop
151     {%
152         ##3##1\ekv@stop\ekv@mark##2%
153     }
154 \long\def\ekv@eq@active##1#2##2\ekv@mark##3##4\ekv@stop
155     {%
156         ##3##1\ekv@stop\ekv@mark##2%
157     }

```

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

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

```

158 \long\def\ekv@set@other##1##2,%
159     {%

```

First we test whether we're done.

```

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

```

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

```

161     \ekv@eq@other##2\ekv@nil\ekv@mark\ekv@set@eq@other@a
162     =\ekv@mark\ekv@set@eq@active\ekv@stop

```

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

```

163     ##1%
164     \ekv@set@other##1\ekv@mark
165 }

```

(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
\ekv@set@eq@other@b only be inserted if the $\langle key \rangle = \langle value \rangle$ pair contained at least one equal sign of category
other and ##1 will contain everything up to that equal sign.

```

166 \long\def\ekv@set@eq@other@a##1\ekv@stop
167     {%
168     \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@set@eq@other@active@a
169     #2\ekv@mark\ekv@set@eq@other@b\ekv@stop
170 }

```

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.

```

171 \long\def\ekv@set@eq@other@b
172     ##1\ekv@nil\ekv@mark\ekv@set@eq@other@active@a\ekv@stop\ekv@mark
173     {%

```

```

174 \ekv@strip{##1}\ekv@set@pair
175 }

```

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

\ekv@set@eq@other@active@a \ekv@set@eq@other@active@a 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 passes things on to \ekv@set@eq@other@active@b which calls \ekv@set@pair on the then correctly split $\langle key \rangle = \langle value \rangle$ pair.

```

176 \long\def\ekv@set@eq@other@active@a##1\ekv@stop##2\ekv@nil\ekv@mark
177 {%
178 \ekv@set@eq@other@active@b{##1}##2=%
179 }
180 \long\def\ekv@set@eq@other@active@b##1%
181 {%
182 \ekv@strip{##1}\ekv@set@pair
183 }

```

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

\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.

```

184 \long\def\ekv@set@eq@active
185 ##1\ekv@nil\ekv@mark\ekv@set@eq@other@a\ekv@stop\ekv@mark
186 {%
187 \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@set@eq@active@
188 #2\ekv@mark\ekv@set@noeq\ekv@stop
189 }

```

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.

```

190 \long\def\ekv@set@eq@active##1\ekv@stop
191 {%
192 \ekv@strip{##1}\ekv@set@pair
193 }

```

(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.

```

194 \long\def\ekv@set@noeq##1\ekv@nil\ekv@mark\ekv@set@eq@active@\ekv@stop\ekv@mark
195 {%
196 \ekv@ifblank{##1\ekv@nil\ekv@ifempty@B\ekv@ifempty@true@F@gobble
197 \ekv@ifempty@A\ekv@ifempty@B}@firstofone
198 {\ekv@strip{##1}\ekv@set@key}%
199 }

```

(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.

```

200 \long\def\ekv@endset@other
201   \ekv@stop
202   \ekv@eq@other##1\ekv@nil\ekv@mark\ekv@set@eq@other@a
203   =\ekv@mark\ekv@set@eq@active\ekv@stop
204   ##2%
205   \ekv@set@other##3\ekv@mark
206   {}

```

(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`

```

207 \long\def\ekvbreak##1##2\ekv@stop#1##3{##1}
208 \long\def\ekvbreakPreSneak ##1##2\ekv@stop#1##3{##1##3}
209 \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`

```

210 \long\def\ekvsneak##1##2\ekv@stop#1##3%
211   {%
212     ##2\ekv@stop#1{##3##1}%
213   }
214 \long\def\ekvsneakPre##1##2\ekv@stop#1##3%
215   {%
216     ##2\ekv@stop#1{##1##3}%
217   }

```

(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.

```

218 \long\def\ekvparse##1##2##3%
219   {%
220     \ekv@parse##1##2\ekv@mark##3#1\ekv@stop#1%
221   }

```

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

`\ekv@parse`

```

222 \long\def\ekv@parse##1##2##3#1%
223   {%
224     \ekv@gobble@from@mark@to@stop##3\ekv@endparse\ekv@stop
225     \ekv@parse@other##1##2##3,\ekv@stop,%
226     \ekv@parse##1##2\ekv@mark
227   }

```

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

\ekv@endparse

```
228 \long\def\ekv@endparse
229     \ekv@stop\ekv@parse@other##1,\ekv@stop,\ekv@parse##2\ekv@mark
230 }
```

(End definition for \ekv@endparse.)

\ekv@parse@other

```
231 \long\def\ekv@parse@other##1##2##3,%
232 {%
233     \ekv@gobble@from@mark@to@stop##3\ekv@endparse@other\ekv@stop
234     \ekv@eq@other##3\ekv@nil\ekv@mark\ekv@parse@eq@other@a
235     =\ekv@mark\ekv@parse@eq@active\ekv@stop
236     ##1##2%
237     \ekv@parse@other##1##2\ekv@mark
238 }
```

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

\ekv@parse@eq@other@a

\ekv@parse@eq@other@b

```
239 \long\def\ekv@parse@eq@other@a##1\ekv@stop
240 {%
241     \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@parse@eq@other@active@a
242     #2\ekv@mark\ekv@parse@eq@other@b\ekv@stop
243 }
244 \long\def\ekv@parse@eq@other@b
245     ##1\ekv@nil\ekv@mark\ekv@parse@eq@other@active@a\ekv@stop\ekv@mark
246 {%
247     \ekv@strip{##1}\ekv@parse@pair
248 }
```

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

\ekv@parse@eq@other@active@a

\ekv@parse@eq@other@active@b

```
249 \long\def\ekv@parse@eq@other@active@a##1\ekv@stop##2\ekv@nil\ekv@mark
250 {%
251     \ekv@parse@eq@other@active@b{##1}##2=%
252 }
253 \long\def\ekv@parse@eq@other@active@b##1%
254 {%
255     \ekv@strip{##1}\ekv@parse@pair
256 }
```

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

\ekv@parse@eq@active

\ekv@parse@eq@active@

```
257 \long\def\ekv@parse@eq@active
258     ##1\ekv@nil\ekv@mark\ekv@parse@eq@other@a\ekv@stop\ekv@mark
259 {%
260     \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@parse@eq@active@
261     #2\ekv@mark\ekv@parse@noeq\ekv@stop
262 }
263 \long\def\ekv@parse@eq@active@##1\ekv@stop
264 {%
265     \ekv@strip{##1}\ekv@parse@pair
266 }
```

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

\ekv@parse@noeq

```

267 \long\def\ekv@parse@noeq
268   ##1\ekv@nil\ekv@mark\ekv@parse@eq@active@\ekv@stop\ekv@mark
269   {%
270     \ekv@ifblank@##1\ekv@nil\ekv@ifempty@B\ekv@ifempty@true@F@gobbletwo
271     \ekv@ifempty@A\ekv@ifempty@B\@firstofone
272     {\ekv@strip{##1}\ekv@parse@key}%
273   }

```

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

\ekv@endparse@other

```

274 \long\def\ekv@endparse@other
275   \ekv@stop
276   \ekv@eq@other##1\ekv@nil\ekv@mark\ekv@parse@eq@other@a
277   =\ekv@mark\ekv@parse@eq@active\ekv@stop
278   ##2%
279   \ekv@parse@other##3\ekv@mark
280   {}

```

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

\ekv@parse@pair

\ekv@parse@pair@

```

281 \long\def\ekv@parse@pair##1##2\ekv@nil
282   {%
283     \ekv@strip{##2}\ekv@parse@pair@{##1}%
284   }
285 \long\def\ekv@parse@pair@##1##2##3##4%
286   {%
287     \unexpanded{##4{##2}{##1}}%
288   }

```

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

\ekv@parse@key

```

289 \long\def\ekv@parse@key##1##2##3%
290   {%
291     \unexpanded{##2{##1}}%
292   }

```

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

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

```

293 }
294 \begingroup
295 \catcode'\,=13
296 \catcode'\==13
297 \ekvset,=

```

\ekvchangeset Provide a macro that is able to switch out the current $\langle set \rangle$ in $\backslash ekvset$. This operation is slow (by comparison, it should be slightly faster than $\backslash ekvsneak$), but allows for something similar to pgfkeys's $\langle key \rangle/.cd$ 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 $\backslash ekvsneak$ macros reading and reinserting the remainder of the $\langle key \rangle=\langle value \rangle$ list.

```

298 \def\ekvchangeset#1%
299   {%
300     \expandafter\ekv@changeset\csname\ekv@undefined@set{#1}\endcsname\ekv@mark
301   }

```

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

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

```

302 \long\def\ekv@changeset#1#2\ekv@set@other#3#4\ekv@set#5%
303   {%
304     \ekv@gobble@mark#2\ekv@set@other#1#4\ekv@set#1%
305   }

```

(End definition for $\backslash ekv@changeset$.)

\ekv@set@pair $\backslash ekv@set@pair$ gets invoked with the space and brace stripped key-name as its first argument, the value as the second argument, and the set name as the third argument. It builds the key-macro name and provides everything to be able to throw meaningful error messages if it isn't defined. $\backslash ekv@set@pair@$ will space and brace strip the value if the macro is defined and call the key-macro. Else it'll branch into the error messages provided by $\backslash ekv@set@pair$.

```

306 \long\def\ekv@set@pair#1#2\ekv@nil#3%
307   {%
308     \expandafter\ekv@set@pair@
309     \csname
310       \ifcsname #3{#1}\endcsname
311       #3{#1}%
312     \else
313       relax%
314     \fi
315     \endcsname
316     {#2}%
317     {%
318       \ekv@ifdefined{#3{#1}N}%
319       \ekv@err@noarg
320       \ekv@err@unknown
321       #3{#1}%
322     }%
323   }
324 \long\def\ekv@set@pair@#1#2%
325   {%
326     \ifx#1\relax
327       \ekv@fi@secondoftwo
328     \fi
329     \@firstoftwo
330     {\ekv@strip{#2}#1}%
331   }

```

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

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

```

332 \long\def\ekv@set@key#1#2%
333   {%
334     \expandafter\ekv@set@key@
335     \csname
336       \ifcsname #2{#1}N\endcsname
337       #2{#1}N%
338     \else
339       relax%
340     \fi
341     \endcsname
342   {%
343     \ekv@ifdefined{#2{#1}}%
344     \ekv@err@reqval
345     \ekv@err@unknown
346     #2{#1}%
347   }%
348 }
349 \def\ekv@set@key@#1%
350   {%
351     \ifx#1\relax
352     \ekv@fi@secondoftwo
353     \fi
354     \@firstoftwo#1%
355   }

```

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

\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.

```

356 \begingroup
357 \edef\ekv@err
358   {%
359     \endgroup
360     \unexpanded{\long\def\ekv@err}##1%
361     {%
362       \unexpanded{\expandafter\ekv@err@\@firstofone}%
363       {\unexpanded\expandafter{\csname ! expkv Error:\endcsname}##1.}%
364       \unexpanded{\ekv@stop}%
365     }%
366   }
367 \ekv@err
368 \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 instead of the full strings.

```

\ekv@err@common@
\ekv@err@unknown 369 \long\def\ekv@err@common #1#2{\expandafter\ekv@err@common@\string#2{#1}}
\ekv@err@noarg    370 \long\def\ekv@err@common@#1'#2' #3.#4#5{\ekv@err{#4 ('#5', set '#2')}}
\ekv@err@reqval

```



```

371 \long\def\ekv@err@unknown#1#2{\ekv@err@common{unknown key}#1{#2}}
372 \long\def\ekv@err@noarg #1#2{\ekv@err@common{value forbidden}#1{#2}}
373 \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@a \ekv@strip also strips one level of outer braces *after* stripping spaces, so an input of
\ekv@strip@b {abc} becomes abc after stripping. It should be used with #1 prefixed by \ekv@mark.
\ekv@strip@c Also this implementation at most strips *one* space from both sides.

```

374 \def\ekv@strip#1%
375 {%
376   \long\def\ekv@strip##1%
377   {%
378     \ekv@strip@a
379     ##1%
380     \ekv@nil
381     \ekv@mark#1%
382     #1\ekv@nil{}%
383     \ekv@stop
384   }%
385   \long\def\ekv@strip@a##1\ekv@mark#1##2\ekv@nil##3%
386   {%
387     \ekv@strip@b##3##1##2\ekv@nil
388   }%
389   \long\def\ekv@strip@b##1#1\ekv@nil
390   {%
391     \ekv@strip@c##1\ekv@nil
392   }%
393   \long\def\ekv@strip@c\ekv@mark##1\ekv@nil##2\ekv@stop##3%
394   {%
395     ##3{##1}%
396   }%
397 }
398 \ekv@strip{ }

```

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

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

```

399 \catcode'\@=\ekv@tmp

```

Index

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

E

`\ekvbreak` 5, 207
`\ekvbreakPostSneak` 5, 207
`\ekvbreakPreSneak` 5, 207
`\ekvchangeset` 5, 298
`\ekvDate` 4, 4, 8, 18
`\ekvdef` 2, 77
`\ekvdefNoVal` 2, 77
`\ekvifdefined` 4, 75
`\ekvifdefinedNoVal` 4, 75
`\ekvlet` 2, 77
`\ekvletkv` 3, 77
`\ekvletkvNoVal` 3, 77
`\ekvletNoVal` 3, 77
`\ekvparse` 4, 218
`\ekvset` 3, 133, 297
`\ekvsneak` 5, 210
`\ekvsneakPre` 5, 210
`\ekvVersion` 4, 4, 8, 18

T

TeX and L^AT_EX 2_ε commands:

`\@firstofone` 23, 39, 40, 43, 197, 271, 362
`\@firstoftwo` 23, 38, 52, 329, 354
`\@gobble` 23, 73
`\@secondoftwo` 23, 34, 37, 71
`\ekv@changeset` 300, 302
`\ekv@checkvalid`
..... 58, 79, 87, 95, 103, 111, 120
`\ekv@defset` 82, 90, 98, 106, 115, 124, 127
`\ekv@endparse` 224, 228
`\ekv@endparse@other` 233, 274
`\ekv@endset` 142, 146
`\ekv@endset@other` 160, 200
`\ekv@eq@active` . 150, 168, 187, 241, 260
`\ekv@eq@other` .. 150, 161, 202, 234, 276
`\ekv@err` 356, 370
`\ekv@err@` 356
`\ekv@err@common` 369
`\ekv@err@common@` 369
`\ekv@err@noarg` 319, 369
`\ekv@err@reqval` 344, 369

`\ekv@err@unknown` 320, 345, 369
`\ekv@fi@secondoftwo` .. 23, 50, 327, 352
`\ekv@gobble@from@mark@to@stop` ..
..... 23, 142, 160, 224, 233
`\ekv@gobble@mark` 23, 304
`\ekv@gobbleto@stop` 23, 368
`\ekv@ifblank@` 45, 196, 270
`\ekv@ifdefined` ... 46, 75, 76, 318, 343
`\ekv@ifempty` 31, 60, 66
`\ekv@ifempty@` 31, 45
`\ekv@ifempty@A` 33,
34, 36, 37, 38, 39, 40, 43, 45, 197, 271
`\ekv@ifempty@B` 33, 34,
36, 37, 38, 39, 40, 43, 196, 197, 270, 271
`\ekv@ifempty@false` 31
`\ekv@ifempty@true` 31
`\ekv@ifempty@true@F` 31
`\ekv@ifempty@true@F@gobble` .. 31, 196
`\ekv@ifempty@true@F@gobbletwo` 31, 270
`\ekv@mark` 29,
30, 45, 138, 144, 147, 150, 152, 154,
156, 161, 162, 164, 168, 169, 172,
176, 185, 187, 188, 194, 202, 203,
205, 220, 226, 229, 234, 235, 237,
241, 242, 245, 249, 258, 260, 261,
268, 276, 277, 279, 300, 381, 385, 393
`\ekv@name` 5, 54, 75,
76, 81, 89, 97, 105, 113, 114, 122, 123
`\ekv@name@key` 5, 54, 130
`\ekv@name@set` 5, 54, 130
`\ekv@nil` ... 161, 168, 172, 176, 185,
187, 194, 196, 202, 234, 241, 245,
249, 258, 260, 268, 270, 276, 281,
306, 380, 382, 385, 387, 389, 391, 393
`\ekv@parse` 220, 222, 229
`\ekv@parse@eq@active` ... 235, 257, 277
`\ekv@parse@eq@active@` 257, 268
`\ekv@parse@eq@other@a`
..... 234, 239, 258, 276
`\ekv@parse@eq@other@active@a` ...
..... 241, 245, 249
`\ekv@parse@eq@other@active@b` ... 249
`\ekv@parse@eq@other@b` 239
`\ekv@parse@key` 272, 289

\ekv@parse@noeq	261, <u>267</u>	\ekv@set@pair	174, 182, 192, <u>306</u>
\ekv@parse@other ..	225, 229, <u>231</u> , 279	\ekv@set@pair@	308, <u>324</u>
\ekv@parse@pair ...	247, 255, <u>265</u> , <u>281</u>	\ekv@stop	28, 30, 138, 142, 143,
\ekv@parse@pair@	<u>281</u>		147, 150, 152, 154, 156, 160, 162,
\ekv@set	137, <u>140</u> , 147, 302, 304		166, 169, 172, 176, 185, 188, 190,
\ekv@set@eq@active	162, <u>184</u> , 203		194, 201, 203, 207, 208, 209, 210,
\ekv@set@eq@active@	<u>184</u> , 194		212, 214, 216, 220, 224, 225, 229,
\ekv@set@eq@other@a	161, <u>166</u> , 185, 202		233, 235, 239, 242, 245, 249, 258,
\ekv@set@eq@other@active@a			261, 263, 268, 275, 277, 364, 383, 393
	168, 172, <u>176</u>	\ekv@strip	174, 182, 192,
\ekv@set@eq@other@active@b	<u>176</u>		198, 247, 255, 265, 272, 283, 330, <u>374</u>
\ekv@set@eq@other@b	<u>166</u>	\ekv@strip@a	<u>374</u>
\ekv@set@key	198, <u>332</u>	\ekv@strip@b	<u>374</u>
\ekv@set@key@	334, 349	\ekv@strip@c	<u>374</u>
\ekv@set@noeq	188, <u>194</u>	\ekv@tmp	1, 62, 68, 399
\ekv@set@other		\ekv@undefined@set	<u>57</u> ,
	143, 147, <u>158</u> , 205, 302, 304		82, 90, 98, 106, 115, 124, 137, 300