Abstract

exPkv provides a small interface for ⟨key⟩=⟨value⟩ parsing. The parsing macro is fully expandable, the ⟨code⟩ of your keys might be not. exPkv is pretty fast, but not the fastest available ⟨key⟩=⟨value⟩ solution (keyval 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 ⟨key⟩=⟨value⟩ Macro Using \ekvsneak .......... 9
  1.5 Error Messages ..................................................... 11
    1.5.1 Load Time .................................................... 11
    1.5.2 Defining Keys ................................................ 11
    1.5.3 Using Keys .................................................. 11
  1.6 License ................................................................. 12

2 Implementation .......................................................... 13
  2.1 The \LaTeX{} Package .............................................. 13
  2.2 The Generic Code ................................................ 13

Index ............................................................................. 26

*jspratte@yahoo.de
1 Documentation

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

\texttt{expkv} is usable as generic code or as a \LaTeX\ package. To use it, just use one of:
\begin{verbatim}
\usepackage{expkv} % \LaTeX
\input expkv % plainTeX
\end{verbatim}

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

In the \texttt{expkv} family are other packages contained which provide additional functionality. Those packages currently are:

\texttt{expkv-def} a key-defining frontend for \texttt{expkv} using a \texttt{⟨key⟩=⟨value⟩} syntax
\texttt{expkv-cs} define expandable \texttt{⟨key⟩=⟨value⟩} macros using \texttt{expkv}

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

1.1 Setting up Keys

\texttt{expkv} provides a rather simple approach to setting up keys, similar to keyval. However there is an auxiliary package named \texttt{expkv-def} which provides a more sophisticated interface, similar to well established packages like pgfkeys or \LaTeX{3}keys.

Keys in \texttt{expkv} (as in almost all other \texttt{⟨key⟩=⟨value⟩} implementations) belong to a set such that different sets can contain keys of the same name. Unlike many other implementations \texttt{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 \texttt{NoVal} keys by \texttt{expkv}), but both can have the same name (on the user level).

The following macros are available to define new keys. Those macros containing “\texttt{def}” in their name can be prefixed by anything allowed to prefix \texttt{\def} (but don’t use \texttt{\outer}, keys defined with it won’t ever be usable), prefixes allowed for \texttt{\let} can prefix those with “\texttt{let}” in their name, accordingly. Neither \texttt{⟨set⟩} nor \texttt{⟨key⟩} are allowed to be empty for new keys and must not contain a \texttt{\par} or tokens that expand to it – they must be legal inside of \texttt{\csname ...\endcsname}.

\begin{verbatim}
\ekvdef\ekvdefNoVal\ekvlet
\end{verbatim}

\texttt{\ekvdef} \texttt{⟨set⟩}{⟨key⟩}{⟨code⟩}

Defines a \texttt{⟨key⟩} taking a value in a \texttt{⟨set⟩} to expand to \texttt{⟨code⟩}. In \texttt{⟨code⟩} you can use \texttt{#1} to refer to the given value.

\texttt{\ekvdefNoVal} \texttt{⟨set⟩}{⟨key⟩}{⟨code⟩}

Defines a no value taking \texttt{⟨key⟩} in a \texttt{⟨set⟩} to expand to \texttt{⟨code⟩}.

\texttt{\ekvlet} \texttt{⟨set⟩}{⟨key⟩}{⟨cs⟩}

Let the value taking \texttt{⟨key⟩} in \texttt{⟨set⟩} to \texttt{⟨cs⟩}, there are no checks on \texttt{⟨cs⟩} enforced.
1.2 Parsing Keys

\ekvset \ekvset\{\{set\}\}\{\{key\}\}\{cs\}
Let the no value taking \(\{key\}\) in \(\{set\}\) to \(\{cs\}\), it is not checked whether \(\{cs\}\) exists or that it takes no parameter.

\ekvletkv \ekvletkv\{\{set\}\}\{\{key\}\}\{\{set2\}\}\{\{key2\}\}
Let the \(\{key\}\) in \(\{set\}\) to \(\{key2\}\) in \(\{set2\}\), it is not checked whether that second key exists.

\ekvletkvNoVal \ekvletkvNoVal\{\{set\}\}\{\{key\}\}\{\{set2\}\}\{\{key2\}\}
Let the \(\{key\}\) in \(\{set\}\) to \(\{key2\}\) in \(\{set2\}\), it is not checked whether that second key exists.

\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 \(\{foobarcod\}\{baz\}\), 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 \langle key \rangle \langle value \rangle pairs and provides those list elements which are only keys as the argument to \langle cs1 \rangle, and those which are a \langle key \rangle \langle value \rangle pair to \langle cs2 \rangle as two arguments. It is fully expandable as well and returns the parsed list in \unexpanded, which has no effect outside of an \expanded or \edef context\(^1\). If you need control over the necessary steps of expansion you can use \expanded around it.

\ekvbreak, \ekvsneak, and \ekvchangeset and their relatives don’t work in \ekvparse. It is analogue to expl3’s \keyval_parse:Nnn, but not with the same parsing rules – \keyval_parse:Nnn throws an error on multiple equal signs per \langle key \rangle \langle value \rangle pair and on empty \langle key \rangle names in a \langle key \rangle \langle value \rangle 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 \langle key \rangle. 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{\ekvparse\kv{foo = bar, key, baz={zzz}}}

would expand to
\handle\kv{foo}{bar}\kv{key}\kv{baz}{zzz}

\ekvVersion
\ekvDate

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

\ekvifdefined{\set}{\langle key \rangle}{\langle true \rangle}{\langle false \rangle}
\ekvifdefinedNoVal{\set}{\langle key \rangle}{\langle true \rangle}{\langle false \rangle}

These two macros test whether there is a \langle key \rangle in \langle set \rangle. 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) \ekvparse would expand in exactly two steps. This isn’t always necessary, but makes the parsing considerably slower. If this is necessary for your application you can put an \expanded around it and will still be faster since you need only a single \expandafter this way.
\ekvbreak \ekvbreakPreSneak \ekvbreakPostSneak

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

\ekvsneak \ekvsneakPre

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

\ekvchangeset

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

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

The names of the macros that correspond to a key in a set are build with these macros. The default definition of \ekv@name@set is “\ekv\{set\}” and the default of \ekv@name@key is “\langle key \rangle”. The complete name is build using \ekv@name which is equivalent to \ekv@name@set\{\langle set \rangle\}\ekv@name@key\{\langle key \rangle\}. For \NoVal keys an additional \textbackslash N gets appended irrespective of these macros’ definition, so their name is \ekv\{set\}\{\langle key \rangle\}\textbackslash 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, \texttt{exPkv} is bug free. But if you find \texttt{bugshidden features} you can tell me about them either via mail (see the first page) or directly on GitHub if you have an account there: \url{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 \texttt{l3benchmark} package. The key and its usage should be equivalent to

\protected\ekvdef{test}{[height]\texttt{\def\myheight{\#1}}}
\ekvset{test}{\ height = 6 \}

\textsuperscript{2}Thanks, David!
and only the usage of the key, not its definition, is benchmarked. For the impatient, the essence of these comparisons regarding speed and buggy behaviour is contained in Table 1.

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

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

\textbf{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:

\begin{verbatim}
\setkeys{foo}{bar=baz}
\setkeys{foo}{bar= \{baz\}}
\setkeys{foo}{bar=\{baz\}}
\end{verbatim}

\textbf{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 space 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.

\textbf{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 exPkv. Also, it can’t parse \texttt{\long} input, so as soon as your values contain a \texttt{\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 \url{https://tex.stackexchange.com/questions/461783}).

\textbf{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 \TeX\Live\texttt{2019} (frozen) which is a bit slower than versions starting with \TeX\Live\texttt{2020}.
\textbf{pgfkeys} is around 2.7 times slower for one key, but has an \textit{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 \texttt{\pgfkeys} by factor 8. Also I don't know whether this might introduce new bugs.

\textbf{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.

\textbf{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.

\textbf{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:

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

\textbf{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:

\begin{verbatim}
\usepackage{yax}
\defactiveparameter yax {{\storevalue\myheight yax : height}}% key setup
\setparameterlist[yax]{{ height = 6 }}% benchmarked
\end{verbatim}

\section{1.4 Examples}

\subsection{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 \texttt{\ourdim} having a default value from its initialization. Now we want to be able to change that dimension with the \texttt{width} key to some specified value. For that we'd do

\begin{verbatim}
\newdimen\ourdim
\ourdim=150pt
\protected\ekvdef\ourdim{width}{\ourdim=\#1}\relax
\end{verbatim}

as you can see, we use the set \texttt{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 \texttt{\hsize}, so we also define

\begin{verbatim}
\protected\ekvdefNoVal\ourdim{width}{\ourdim=.9\hsize}\relax
\end{verbatim}

Now we set up our macro to use this \langle key\rangle=\langle value\rangle interface
Table 1: Comparison of \( \langle \text{key} \rangle = \langle \text{value} \rangle \) packages. The packages are ordered from fastest to slowest for one \( \langle \text{key} \rangle = \langle \text{value} \rangle \) pair. Benchmarking was done using \texttt{l3benchmark} and the scripts in the \texttt{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.

<table>
<thead>
<tr>
<th>Package</th>
<th>( p_1 )</th>
<th>( p_0 )</th>
<th>( T_0 )</th>
<th>BB</th>
<th>CF</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyval</td>
<td>13.4</td>
<td>2.6</td>
<td>6.9</td>
<td>yes</td>
<td>yes</td>
<td>2014-10-28</td>
</tr>
<tr>
<td>expkv</td>
<td>19.7</td>
<td>7.3</td>
<td>9.6</td>
<td>no</td>
<td>no</td>
<td>2020-04-07</td>
</tr>
<tr>
<td>options</td>
<td>23.4</td>
<td>15.6</td>
<td>19.9</td>
<td>yes</td>
<td>yes</td>
<td>2015-03-01</td>
</tr>
<tr>
<td>pgfkeys</td>
<td>24.6</td>
<td>46.2</td>
<td>52.8</td>
<td>yes</td>
<td>yes</td>
<td>2020-01-08</td>
</tr>
<tr>
<td>kvsetkeys</td>
<td>*</td>
<td>*</td>
<td>39.8</td>
<td>no</td>
<td>no</td>
<td>2019-12-15</td>
</tr>
<tr>
<td>l3keys</td>
<td>108.0</td>
<td>56.3</td>
<td>50.7</td>
<td>no</td>
<td>no</td>
<td>2020-02-25</td>
</tr>
<tr>
<td>simplekv</td>
<td>149.2</td>
<td>25.7</td>
<td>8.1</td>
<td>yes</td>
<td>yes</td>
<td>2017-08-08</td>
</tr>
<tr>
<td>xkeyval</td>
<td>248.8</td>
<td>234.4</td>
<td>161.9</td>
<td>yes</td>
<td>yes</td>
<td>2014-12-03</td>
</tr>
<tr>
<td>yax</td>
<td>443.6</td>
<td>170.5</td>
<td>115.7</td>
<td>yes</td>
<td>yes</td>
<td>2010-01-22</td>
</tr>
<tr>
<td>ltxkeys</td>
<td>3516.0</td>
<td>5254.8</td>
<td>5487.0</td>
<td>no</td>
<td>no</td>
<td>2012-11-17</td>
</tr>
</tbody>
</table>

*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 \( 3.5 \times 10^{-3} \)). If one extrapolates the fits for 100 \( \langle \text{key} \rangle = \langle \text{value} \rangle \) 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 \( p_0 = 6132.3 \).
Finally we can use our macro like in the following

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

The same key using \texttt{expkv-def} Using \texttt{expkv-def} we can set up the equivalent key using a \langle key \rangle = \langle value \rangle interface, after the following we could use \texttt{\ourmacro} in the same way as above. \texttt{expkv-def} will allocate and initialise \ourdim and define the \texttt{width} key protected for us, so the result will be exactly the same – with the exception that the default will use \texttt{\ourdim=0.9\hsize} instead.

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

\subsection{An Expandable \langle key \rangle = \langle value \rangle Macro Using \texttt{\ekvsetn}}

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, \texttt{\ekvset} or \texttt{\ekvparse}. For this example we also want to show the usage of \texttt{\ekvsetn}, hence we'll choose \texttt{\ekvset}. And we'll have to use \texttt{\ekvset} such that it builds a parsable list for our macro internals. To gain back control after \texttt{\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 \texttt{\ekvsetnPre} 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 \texttt{\ekvsetnPre} for the user-level keys. If we wanted some key for which the first usage should be the binding one we would use \texttt{\ekvsetn} 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 \texttt{xfp} package.

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

\begin{verbatim}
\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}{\ekvset{expex}{#1, internal}{\rnd}{\deg}{d}{\f}{\sine@stop}{#2}}
\makeatother
\end{verbatim}

For the sake of simplicity we defined the macro \texttt{\sine} with two mandatory arguments, the first being the \texttt{(key)=(value)} list, the second the argument to the trigonometric function. We could've used \texttt{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.

\begin{verbatim}
\def\sine@rnd#1\rnd#2\sine@stop{\sine@deg#1\deg#2\sine@stop{#2}}
\def\sine@deg#1\deg#2\sine@stop{\sine@f#1\f#2\sine@stop{#2}}
\def\sine@f#1\f#2\sine@stop{\sine@final{#2}}
\end{verbatim}

After the macros \texttt{\sine@rnd}, \texttt{\sine@deg}, and \texttt{\sine@f} the macro \texttt{\sine@final} will see \texttt{\sine@final{\f}{degree/radian}{round}{num}}. Now \texttt{\sine@final} has to expandably deal with those arguments such that the \texttt{\fpeval} macro of \texttt{xfp} gets the correct input. Luckily this is pretty straight forward in this example. In \texttt{\fpeval} the trigonometric functions have names such as \texttt{sin} or \texttt{cos} and the degree taking variants \texttt{sind} or \texttt{cosd}.

And since the \texttt{degree} key puts a \texttt{d} in \#2 and the \texttt{radian} key leaves \#2 empty all we have to do to get the correct function name is stick the two together.

\begin{verbatim}
\newcommand{\sine@final}{\fpeval{\round(#1#2(#4),#3)}}
\makeatother
\end{verbatim}

Let's test our macro:

\begin{verbatim}
\sine{60}\par
\sine{round=10}{60}\par
\sine[f=\cos,radian]{pi}\par
\edef\myval{\sine[f=\tan]{1}}\texttt{\meaning\myval}\par
\end{verbatim}

\begin{verbatim}
0.866
0.8660254038
-1
-0.017
\end{verbatim}

\textbf{The same macro using \texttt{expvcs}} Using \texttt{expvcs} we can set up something equivalent with a bit less code. The implementation chosen in \texttt{expvcs} is more efficient than the example above and way easier to code.

\begin{verbatim}
\makeatletter
\ekvcSplitAndForward\sine\sine@
\end{verbatim}
\f=sin ,
  unit=d ,
  round=3 ,
\ekv{SecondaryKeys}{sine}
  nmeta degree={unit=d} ,
  nmeta radian={unit={}} ,
\newcommand\*{\sine[@]{}}{\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 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 \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 \ekvset{key}{value} handling are encountered, the entry in the comma separated list will be omitted after the error is thrown and the next \ekvset{key}{value} pair will be parsed.

If you're using an undefined key you'll get:
!
Undefined control sequence.
<argument> \! expkv Error:

unknown key ('<key>', set '<set>').

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

!
Undefined control sequence.
<argument> \! expkv Error:

value required ('<key>', set '<set>').

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

!
Undefined control sequence.
<argument> \! expkv Error:

value forbidden ('<key>', set '<set>').

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

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

1.6 License

Copyright © 2020 Jonathan P. Spratte

This work may be distributed and/or modified under the conditions of the LATEX 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 \LaTeX Package

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

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

2.2 The Generic Code

The rest of this implementation will be the generic code. We make sure that it’s only input once:

\begin{verbatim}
\expandafter\ifx\csname ekvVersion\endcsname\relax
\else
\expandafter\endinput
\fi
\end{verbatim}

Check whether \TeX is available – \expkv requires \TeXX.

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

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

\begin{verbatim}
\def\ekvVersion{1.2}
\def\ekvDate{2020-04-10}
\end{verbatim}

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

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

\begin{verbatim}
\csname ekv@tmp\endcsname
\end{verbatim}

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

\begin{verbatim}
\expandafter\chardef\csname ekv@tmp\endcsname=\catcode\@=11
\end{verbatim}

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

\begin{verbatim}
\long\def\@gobble#1{}
\long\def\@firstofone#1{#1}
\long\def\@firstoftwo#1#2{#1}
\long\def\@secondoftwo#1#2{#2}
\long\def\ekv@fi@secondoftwo\fi\@firstoftwo#1#2{\fi#2}
\long\def\ekv@gobbleto@stop#1\ekv@stop{}
\def\ekv@gobble@mark\ekv@mark{}
\long\def\ekv@gobble@from@mark@to@stop\ekv@mark#1\ekv@stop{}
\end{verbatim}

(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.

\begin{verbatim}
\long\def\ekv@ifempty#1{%
\ekv@ifempty@\ekv@ifempty@A#1\ekv@ifempty@B\ekv@ifempty@true
\ekv@ifempty@A\ekv@ifempty@B\@secondoftwo
\}
\long\def\ekv@ifempty@#1\ekv@ifempty@A\ekv@ifempty@B{\ekv@ifempty@true@F\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo#1#2{#1}}
\long\def\ekv@ifempty@true@F@gobble\ekv@ifempty@A\ekv@ifempty@B\@firstofone#1#2{#1}{\ekv@stop}
\end{verbatim}

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

\begin{verbatim}
\long\def\ekv@ifblank@\ekv@mark#1{\ekv@ifempty@\ekv@ifempty@A}\ekv@ifempty@true@F@gobbletwo
\ekv@ifempty@true@F@gobbletwo\ekv@ifempty@true@F@gobbletwo
\end{verbatim}

(End definition for \ekv@ifblank.)

\begin{verbatim}
\long\def\ekv@ifdefined\ekv@mark#1{%#1}{\ekv@ifempty@\ekv@ifempty@A}\ekv@ifempty@true@F@gobbletwo
\ekv@ifempty@true@F@gobbletwo\ekv@ifempty@true@F@gobbletwo
\end{verbatim}

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

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

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

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

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

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.

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

(End definition for \ekv@checkvalid.)
And provide user-level macros to test whether a key is defined.

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

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

Set up the key defining macros `\ekvdef` etc.

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

16
In order to enhance the speed the set name given to \ekvset will be turned into a control sequence pretty early, so we have to define that control sequence.

\protected\def\ekv@defset#1#2{\ifx#1\relax\edef#1##1{\ekv@name@set{#2}\ekv@name@key{##1}}\fi}

(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 \emph{,}, and #2 will be a \emph{=}.

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

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

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

\long\def\ekv@endset{\ekv@stop\ekv@set@other##1,\ekv@stop,\ekv@set##2\ekv@mark
##3}{##3}
Splitting at equal signs will be done in a way that checks whether there is an equal sign and splits at the same time. This gets quite messy and the code might look complicated, but this is pretty fast (faster than first checking for an equal sign and splitting if one is found). The splitting code will be adapted for \ekvset and \ekvparse to get the most speed, but some of these macros don’t require such adaptions. \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\TeX’s argument grabbing rules and we don’t have to split after the branching if the equal sign was there.

\long\def\ekv@eq@other##1=##2\ekv@mark##3##4\ekv@stop
\{{%##3##1\ekv@stop\ekv@mark##2%\}}
\long\def\ekv@eq@active##1#2##2\ekv@mark##3##4\ekv@stop
\{{%##3##1\ekv@stop\ekv@mark##2%\}}

The macro \ekv@set@other is guaranteed to get only single \langle key\rangle=\langle value\rangle pairs.
\long\def\ekv@set@other##1##2,\%
\{{%First we test whether we’re done.
\ekv@gobble@from@mark@to@stop##2\ekv@endset@other\ekv@stop
If not we split at the equal sign of category other.
\ekv@eq@other##2\ekv@nil\ekv@mark\ekv@set@eq@other@active@a=\ekv@mark\ekv@set@eq@active\ekv@stop
And insert the set name and the next recursion step of \ekv@set@other.
##1%\ekv@set@other##1\ekv@mark
\}}

\ekv@set@eq@other@a\ekv@set@eq@other@b
The first of these two macros runs the split-test for equal signs of category active. It will 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.
\long\def\ekv@set@eq@other@0\ekv@set@eq@other@a##1\ekv@stop
\{{%
\ekv@eq@active##1\ekv@nil\ekv@mark\ekv@set@eq@other@active@a#2\ekv@mark\ekv@set@eq@other@0b\ekv@stop
\}}

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.
\long\def\ekv@set@eq@other@b##1\ekv@nil\ekv@mark\ekv@set@eq@other@active@a\ekv@stop\ekv@mark
\{{%
\ekv@strip{##1}\ekv@set@pair
}

(End definition for `\ekv@set@eq@other@0a` and `\ekv@set@eq@other@0b`.)

`\ekv@set@eq@other@active@0a` will be called if the `(key)=(value)` 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@0b` which calls `\ekv@set@pair` on the then correctly split `(key)=(value)` pair.

\long\def\ekv@set@eq@other@active@0a##1\ekv@stop##2\ekv@nil\ekv@mark{
\ekv@set@eq@other@active@0b{##1}##2=%}

(End definition for `\ekv@set@eq@other@active@0a` and `\ekv@set@eq@other@active@0b`.)

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

\long\def\ekv@set@eq@active\ekv@stop\ekv@mark{\ekv@eq@active\ekv@mark}\ekv@set@eq@active@#2\ekv@stop}

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

\long\def\ekv@set@eq@active@##1\ekv@stop{\ekv@strip{##1}\ekv@set@pair}

(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`.

\long\def\ekv@set@noeq##1\ekv@stop\ekv@mark{\ekv@ifblank{##1}\ekv@nil\ekv@mark\ekv@set@noeq\ekv@mark}{\ekv@strip{##1}\ekv@set@key}\

(End definition for `\ekv@set@noeq`.)
All that’s left for \texttt{\ekv@endset@other} is the macro which breaks the recursion loop at the end. This is done by gobbling all the remaining tokens.

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

(End definition for \texttt{\ekv@endset@other}.)

Provide macros that can completely stop the parsing of \texttt{\ekvset}, who knows what it’ll be useful for.

\begin{verbatim}
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{verbatim}

(End definition for \texttt{\ekvbreak}, \texttt{\ekvbreakPreSneak}, and \texttt{\ekvbreakPostSneak}. These functions are documented on page 5.)

One last thing we want to do for \texttt{\ekvset} is to provide macros that just smuggle stuff after \texttt{\ekvset}’s effects.

\begin{verbatim}
210 \long\def\ekvsneak##1##2\ekv@stop#1##3{%
211 \texttt{##2}\texttt{\ekv@stop#1}{##3##1}%
212 }
213 \long\def\ekvsneakPre##1##2\ekv@stop#1##3{%
214 \texttt{##2}\texttt{\ekv@stop#1}{##1##3}%
215 }
\end{verbatim}

(End definition for \texttt{\ekvsneak} and \texttt{\ekvsneakPre}. These functions are documented on page 5.)

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

\begin{verbatim}
218 \long\def\ekvparsen##1##2##3{%
219 \texttt{\ekvparse##1##2}{\ekv@mark##3}\ekv@stop#1%
220 }
\end{verbatim}

(End definition for \texttt{\ekvparsen}. This function is documented on page 4.)
\texttt{\textbackslash ekv@endparse}
\begin{verbatim}
\long\def\ekv@endparse
\ekv@stop\ekv@parse@other##1,\ekv@stop,\ekv@parse##2\ekv@mark
{}
\end{verbatim}
\textit{(End definition for} \texttt{\textbackslash ekv@endparse}).

\texttt{\textbackslash ekv@parse@other}
\begin{verbatim}
\long\def\ekv@parse@other##1##2##3,%
{%
\ekv@gobble@from@mark@to@stop##3\ekv@endparse@other\ekv@stop
\ekv@parse@eq@other##3\ekv@nil\ekv@mark\ekv@parse@eq@other@a
=\ekv@mark\ekv@parse@eq@active\ekv@stop
##1##2,%
\ekv@parse@other##1##2\ekv@mark
}
\end{verbatim}
\textit{(End definition for} \texttt{\textbackslash ekv@parse@other}).

\texttt{\textbackslash ekv@parse@eq@other@a}
\texttt{\textbackslash ekv@parse@eq@other@b}
\begin{verbatim}
\long\def\ekv@parse@eq@other@a##1\ekv@stop
{%
\ekv@eq@active##1\ekv@nil\ekv@mark\ekv@parse@eq@other@active@a
##2\ekv@mark\ekv@parse@eq@other@b\ekv@stop
}
\end{verbatim}
\textit{(End definition for} \texttt{\textbackslash ekv@parse@eq@other@a} \textit{and} \texttt{\textbackslash ekv@parse@eq@other@b}).

\texttt{\textbackslash ekv@parse@eq@other@active@a}
\texttt{\textbackslash ekv@parse@eq@other@active@b}
\begin{verbatim}
\long\def\ekv@parse@eq@other@active@a##1\ekv@stop##2\ekv@nil\ekv@mark
{%
\ekv@parse@eq@other@active@b{##1}##2=%
}
\end{verbatim}
\textit{(End definition for} \texttt{\textbackslash ekv@parse@eq@other@active@a} \textit{and} \texttt{\textbackslash ekv@parse@eq@other@active@b}).

\texttt{\textbackslash ekv@parse@eq@active}
\texttt{\textbackslash ekv@parse@eq@active@0}
\begin{verbatim}
\long\def\ekv@parse@eq@active
##1\ekv@nil\ekv@mark\ekv@parse@eq@other@0\ekv@stop\ekv@mark
{%
\ekv@parse@eq@active##1\ekv@nil\ekv@mark\ekv@parse@eq@active@0
##2\ekv@mark\ekv@parse@noeq\ekv@stop
}
\end{verbatim}
\textit{(End definition for} \texttt{\textbackslash ekv@parse@eq@active} \textit{and} \texttt{\textbackslash ekv@parse@eq@active@0}).
Finally really setting things up with \ekvset’s temporary meaning:

```latex
\begin{group}
\catcode'\,=13
\catcode'=13
\ekvset,=
```
\texttt{\ekvchangeset} Provide a macro that is able to switch out the current \texttt{⟨set⟩} in \texttt{\ekvset}. This operation is slow (by comparison, it should be slightly faster than \texttt{\ekvsneak}), but allows for something similar to pgfkeys’s \texttt{⟨key⟩}/.\texttt{cd} mechanism. However this operation is more expensive than /.\texttt{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 \texttt{\ekvsneak} macros reading and reinserting the remainder of the \texttt{(key)=(value)} list.

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

(End definition for \texttt{\ekvchangeset}. This function is documented on page \pageref{page5}.)

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

\begin{verbatim}
\longdef\ekv@changeset#1#2\ekv@set@other#3#4\ekv@set#5\
    {\ekv@gobble@mark#2\ekv@set@other#1#4\ekv@set#1}
\end{verbatim}

(End definition for \texttt{\ekvchangeset}.)

\texttt{\ekv@set@pair} \texttt{\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. \texttt{\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 \texttt{\ekv@set@pair}.

\begin{verbatim}
\longdef\ekv@set@pair#1#2\ekv@nil#3\
    {\expandafter\ekv@set@pair\csname\
        \ifcsname \ekv@undefined{#3{#1}}\endcsname
        \ekv@undefined{#3{#1}}\else\relax\fi
        \endcsname\ekv@ifdefined\ekv@err@noarg\ekv@err@unknown
        \ekv@undefined{#3{#1}}}
\end{verbatim}

\begin{verbatim}
\longdef\ekv@set@pair@#1#2\
    {\ifx#1\relax\ekv@fi@secondoftwo\fi\@firstoftwo}
\end{verbatim}
\ekvsetkey  \Analogous to \ekvsetpair, \ekvsetkey builds the \NoVal key-macro and provides an error-branch. \ekvsetkey will test whether the key-macro is defined and if so call it, else the errors are thrown.

\begin{verbatim}
\long\def\ekvsetkey#1#2\% {
  \expandafter\ekvsetkey@
  \csname \ifcsname #2{#1}N\endcsname #2{#1}N\%
  \else
    \relax\%
  \fi
  \endcsname {%
    \ekv@ifdefined{#2{#1}}%
    \ekv@err@reqval
    \ekv@err@unknown
    #2{#1}%
  }%
}
\def\ekvsetkey@#1\% {
  \ifx#1\relax
    \ekv@fi@secondoftwo
  \fi
  \@firstoftwo#1\%
}\end{verbatim}

\end{definition}

\ekverr  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.

\begin{verbatim}
\begingroup
\edef\ekverr {\
  \unexpanded{\long\def\ekverr##1\% {
    \unexpanded{\expandafter\ekverr@\@firstofone}
      \unexpanded{\expandafter{\csname \ifcsname \expkv Error:\endcsname##1.}-%\ekv@stop}}%\ekv@err
  \def\ekverr@{\expandafter\ekv@gobbleto@stop}
\end{verbatim}

\end{definition}

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

\begin{verbatim}
\long\def\ekverrcommon #1#2{\expandafter\ekverr@common\string#2{#1}}
\long\def\ekverr@common@#1#2#3#4#5{\ekverr@4 {#5, set '#2')}\}
\end{verbatim}

24
Finally we borrow some ideas of expl3's \texttt{ltl} to strip spaces from keys and values. This \texttt{ekv@strip} also strips one level of outer braces \textit{after} stripping spaces, so an input of \texttt{\{abc\}} becomes \texttt{abc} after stripping. It should be used with \texttt{#1} prefixed by \texttt{\ekv@mark}.

Also this implementation at most strips \textit{one} space from both sides.

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

Now everything that's left is to reset the category code of \texttt{@}.
### 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ekvbreak</td>
<td>207</td>
</tr>
<tr>
<td>\ekvbreakPostSneak</td>
<td>207</td>
</tr>
<tr>
<td>\ekvbreakPreSneak</td>
<td>207</td>
</tr>
<tr>
<td>\ekvchangeset</td>
<td>298</td>
</tr>
<tr>
<td>\ekvDate</td>
<td>4, 8, 18</td>
</tr>
<tr>
<td>\ekvdef</td>
<td>77</td>
</tr>
<tr>
<td>\ekvdefNoVal</td>
<td>77</td>
</tr>
<tr>
<td>\ekvinfos</td>
<td>75</td>
</tr>
<tr>
<td>\ekvinfosNoVal</td>
<td>75</td>
</tr>
<tr>
<td>\ekvlet</td>
<td>277</td>
</tr>
<tr>
<td>\ekvletkv</td>
<td>77</td>
</tr>
<tr>
<td>\ekvletkvNoVal</td>
<td>77</td>
</tr>
<tr>
<td>\ekvparse</td>
<td>428</td>
</tr>
<tr>
<td>\ekvsno</td>
<td>210</td>
</tr>
<tr>
<td>\ekvsnoPre</td>
<td>210</td>
</tr>
<tr>
<td>\ekvVersion</td>
<td>4, 8, 18</td>
</tr>
</tbody>
</table>

### \TeX and \LaTeX commands:

- \@firstofone 23, 39, 40, 43, 197, 271, 362
- \@firstoftwo 23, 38, 52, 329, 354
- \@gobble 73
- \@secondoftwo 23, 34, 37, 71
- \ekv@changeset 300, 302
- \ekv@checkvalid
- \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@equals@active 150, 168, 187, 241, 260
- \ekv@equals@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, 360
- \ekv@err@unknown 320, 345, 369
- \ekv@fi@secondoftwo 23, 50, 327, 352
- \ekv@gobble@frommark@to@stop
- \ekv@gobble@mark 23, 304
- \ekv@gobbleto@stop 23, 368
- \ekv@ifblank 46, 75, 76, 318, 343
- \ekv@ifempty 31, 60, 66
- \ekv@ifndef@false 31
- \ekv@ifndef@true@false 31
- \ekv@ifempty@true@F@gobble 31, 196
- \ekv@name 54, 75
- \ekv@name@key 76, 81, 89, 97, 105, 113, 114, 122, 123
- \ekv@name@set 54, 130
- \ekv@nil 161, 168, 172, 176, 185
- \ekv@parse 220, 222, 229
- \ekv@parse@eq@active 235, 257, 277
- \ekv@parse@eq@active@false 257, 268
- \ekv@parse@eq@other 234, 239, 258, 276
- \ekv@parse@eq@other@active@a
- \ekv@parse@eq@other@active@b 241, 245, 249
- \ekv@parse@eq@other@b 249
- \ekv@parse@eq@other@b 239
- \ekv@parse@key 272, 280