expkv

an expandable ⟨key⟩=⟨value⟩ implementation

Jonathan P. Spratte*

2021-06-03 v1.8a

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 *swift*, close to the fastest ⟨key⟩=⟨value⟩ implementation. However it is the fastest which copes with active commas and equal signs and doesn't strip braces accidentally.

Contents

*jspratte@yahoo.de

1 Documentation

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

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

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

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

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

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

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

expkv|opt parse package and class options with **expkv**

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

A list of concise comparisons to other ⟨*key*⟩=⟨*value*⟩ packages is contained in subsection 1.7.

1.1 Setting up Keys

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

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

The following macros are available to define new keys. Those macros containing "`def`" in their name can be prefixed by anything allowed to prefix `\def` (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 ⟨*set*⟩ nor ⟨*key*⟩ are allowed to be empty for new keys. ⟨*set*⟩ will be used as is inside of `\csname ...\endcsname` and ⟨*key*⟩ will get `\detokenized`.

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

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

Example: Define `text` in `foo` to store the value inside `\foo@text`:

```
\protected\long\ekvdef{foo}{text}{\def\foo@width{#1}}
```

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

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

Example: Define `bool` in `foo` to set `\iffoo@bool` to `true`:
`\protected\ekvdefNoVal{foo}{bool}{\foo@booltrue}`

\ekvlet `\ekvlet{⟨set⟩}{⟨key⟩}⟨cs⟩`

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

Example: Let `cmd` in `foo` do the same as `\foo@cmd`:
`\ekvlet{foo}{cmd}\foo@cmd`

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

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

Example: See above.

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

Let the ⟨key⟩ in ⟨set⟩ to ⟨key2⟩ in ⟨set2⟩, it is not checked whether that second key exists (but take a look at `\ekvifdefined`).

Example: Let `B` in `bar` be an alias for `A` in `foo`:
`\ekvletkv{bar}{B}{foo}{A}`

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

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

Example: See above.

\ekvdefunknown `\ekvdefunknown{⟨set⟩}{⟨code⟩}`

By default an error will be thrown if an unknown ⟨key⟩ is encountered. With this macro you can define ⟨code⟩ that will be executed for a given ⟨set⟩ when an unknown ⟨key⟩ with a ⟨value⟩ was encountered instead of throwing an error. You can refer to the given ⟨value⟩ with `#1` and to the unknown ⟨key⟩'s name with `#2` in ⟨code⟩.[1] `\ekvdefunknown` and `\ekvredirectunknown` are mutually exclusive, you can't use both.

Example: Also search `bar` for undefined keys of set `foo`:
`\long\ekvdefunknown{foo}{\ekvset{bar}{#2={#1}}}`
This example differs from using `\ekvredirectunknown{foo}{bar}` (see below) in that also the unknown-key handler of the `bar` set will be triggered, error messages for undefined keys will look different, and this is slower than using `\ekvredirectunknown`.

\ekvdefunknownNoVal `\ekvdefunknownNoVal{⟨set⟩}{⟨code⟩}`

As already explained for `\ekvdefunknown`, expkv would throw an error when encountering an unknown ⟨key⟩. With this you can instead let it execute ⟨code⟩ if an unknown `NoVal` ⟨key⟩ was encountered. You can refer to the given ⟨key⟩ with `#1` in ⟨code⟩. `\ekvdefunknownNoVal` and `\ekvredirectunknownNoVal` are mutually exclusive, you can't use both.

[1] That order is correct, this way the code is faster.

Example: Also search bar for undefined keys of set foo:
`\ekvdefunknownNoVal{foo}{\ekvset{bar}{#1}}`

`\ekvredirectunknown` `\ekvredirectunknown{⟨set⟩}{⟨set-list⟩}`

This is a short cut to set up a special `\ekvdefunknown` for ⟨*set*⟩ that will check each set in the comma separated ⟨*set-list*⟩ for the unknown ⟨*key*⟩. You can't use prefixes (so no `\long` or `\protected`) with this macro, the resulting unknown-key handler will always be `\long`. The first set in the ⟨*set-list*⟩ has highest priority. Once the ⟨*key*⟩ is found the remaining sets are discarded, if the ⟨*key*⟩ isn't found in any set an error will be thrown eventually. Note that the error messages are affected by the use of this macro, in particular, it isn't checked whether a `NoVal` key of the same name is defined in order to throw an `unwanted value` error. `\ekvdefunknown` and `\ekvredirectunknown` are mutually exclusive, you can't use both.

Example: For every key not defined in the set foo also search the sets bar and baz:
`\ekvredirectunknown{foo}{bar, baz}`

`\ekvredirectunknownNoVal` `\ekvredirectunknownNoVal{⟨set⟩}{⟨set-list⟩}`

This behaves just like `\ekvredirectunknown` and does the same but for the `NoVal` keys. Again no prefixes are supported. Note that the error messages are affected by the use of this macro, in particular, it isn't checked whether a normal key of the same name is defined in order to throw a `missing value` error. `\ekvdefunknownNoVal` and `\ekvredirectunknownNoVal` are mutually exclusive, you can't use both.

Example: See above.

1.2 Parsing Keys

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

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

`\ekvset` is currently *not* alignment safe.[2] As a result, key names and values that contain an `&` must be wrapped in braces when `\ekvset` is used inside an alignment (like LaTeX 2ε's `tabular` environment) or you have to create a wrapper that ensures an alignment safe context.

Example: Parse `key=arg, key` in the set foo:
`\ekvset{foo}{key=arg, key}`

[2] This might change in the future, I've not decided yet.

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

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

Example: Parse `key=arg, key` in the set `foo` with `\afterwards` sneaked out:
```
\ekvsetSneaked{foo}{\afterwards}{key=arg, key}
```

\ekvsetdef `\ekvsetdef⟨cs⟩{⟨set⟩}`

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

```
\long\def⟨cs⟩#1{\ekvset{⟨set⟩}{#1}}
```

Example: Define the macro `\foosetup` to parse keys in the set `foo` and use it to parse `key=arg, key`:
```
\ekvsetdef\foosetup{foo}
\foosetup{key=arg, key}
```

\ekvsetSneakeddef `\ekvsetSneakeddef⟨cs⟩{⟨set⟩}`

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

```
\long\def⟨cs⟩#1#2{\ekvsetSneaked{⟨set⟩}{#1}{#2}}
```

Example: Define the macro `\foothings` to parse keys in the set `foo` and accept a sneaked argument, then use it to parse `key=arg, key` and sneak `\afterwards`:
```
\ekvsetSneakeddef\foothings{foo}
\foothings{\afterwards}{key=arg, key}
```

\ekvsetdefSneaked `\ekvsetdefSneaked⟨cs⟩{⟨set⟩}{⟨sneaked⟩}`

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

```
\long\def⟨cs⟩#1{\ekvsetSneaked{⟨set⟩}{⟨sneaked⟩}{#1}}
```

Example: Define the macro `\barthing` to parse keys in the set `bar` and always execute `\afterwards` afterwards, then use it to parse `key=arg, key`:
```
\ekvsetdefSneaked\barthing{bar}{\afterwards}
\barthing{key=arg, key}
```

\ekvparse `\ekvparse{⟨code1⟩}{⟨code2⟩}{⟨key⟩=⟨value⟩,...}`

This macro parses the ⟨*key*⟩=⟨*value*⟩ pairs and provides those list elements which are only keys as an argument to ⟨*code1*⟩, and those which are a ⟨*key*⟩=⟨*value*⟩ pair to ⟨*code2*⟩ as two arguments. It is fully expandable as well and returns each element of the parsed list in \unexpanded, which has no effect outside of an \expanded or \edef context. Also \ekvparse expands in exactly two steps of expansion. You can use multiple tokens in ⟨*code1*⟩ and ⟨*code2*⟩ or just a single control sequence name. In both cases the found ⟨*key*⟩ and ⟨*value*⟩ are provided as a brace group following them.

\ekvparse is alignment safe, meaning that you don't have to take any precautions if it is used inside an alignment context (like LaTeX 2ε's `tabular` environment) and any key or value can contain an &.

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

Example:

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

would be equivalent to

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

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

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

or

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

would both expand to

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

1.3 Other Macros

expkv provides some other macros which might be of interest.

\ekvVersion
\ekvDate

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

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

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

Example: Check whether the key `special` is already defined in set `foo`, if it isn't input a file that contains more key definitions:

```
\ekvifdefined{foo}{special}{}{\input{foo.morekeys.tex}}
```

`\ekvifdefinedset` `\ekvifdefinedset{⟨set⟩}{⟨true⟩}{⟨false⟩}`

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

Example: Check whether the set `VeRyUnLiKeLy` is already defined, if so throw an error, else do nothing:

```
\ekvifdefinedset{VeRyUnLiKeLy}
  {\errmessage{VeRyUnLiKeLy already defined}}{}
```

`\ekvbreak` `\ekvbreakPreSneak` `\ekvbreakPostSneak` `\ekvbreak{⟨after⟩}`

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

Example: Define a key `abort` that will stop key parsing inside the set `foo` and execute `\foo@aborted`, or if it got a value `\foo@aborted@with`:

```
\ekvdefNoVal{foo}{abort}{\ekvbreak{\foo@aborted}}
\ekvdef{foo}{abort}{\ekvbreak{\foo@aborted@with{#1}}}
```

`\ekvsneak` `\ekvsneakPre` `\ekvsneak{⟨after⟩}`

Puts ⟨*after*⟩ after the effects of `\ekvset`. The first variant will put ⟨*after*⟩ after any other tokens which might have been sneaked before, while `\ekvsneakPre` will put ⟨*after*⟩ before other smuggled stuff. This reads and reinserts the remainder of the current `\ekvset` macro and its argument list to do its job. After `\ekvset` has parsed the entire ⟨*key*⟩=⟨*value*⟩ list everything that has been `\ekvsneaked` will be left in the input stream. A small usage example is shown in subsubsection 1.4.3.

Example: Define a key `secret` in the set `foo` that will sneak out `\foo@secretly@sneaked`:

```
\ekvdefNoVal{foo}{secret}{\ekvsneak{\foo@secretly@sneaked}}
```

`\ekvchangeset` `\ekvchangeset{⟨new-set⟩}`

Replaces the current set with ⟨*new-set*⟩, so for the rest of the current `\ekvset` call, that call behaves as if it was called with `\ekvset{⟨new-set⟩}`. It is comparable to using ⟨*key*⟩`/.cd` in pgfkeys.

Example: Define a key `cd` in set `foo` that will change to another set as specified in the value, if the set is undefined it'll stop the parsing and throw an error as defined in the macro `\foo@cd@error`:

```
\ekvdef{foo}{cd}
  {\ekvifdefinedset{#1}{\ekvchangeset{#1}}{\ekvbreak{\foo@cd@error}}}
```

\ekvoptarg `\ekvoptarg{⟨next⟩}{⟨default⟩}`

This macro will check for a following optional argument in brackets (`[]`) expandably. After the optional argument there has to be a mandatory one. The code in ⟨*next*⟩ should expect two arguments (the processed optional argument and the mandatory one). If there was an optional argument the result will be ⟨*next*⟩{⟨*optional*⟩}⟨*mandatory*⟩ (so the optional argument will be wrapped in braces, the mandatory argument will be untouched). If there was no optional argument the result will be ⟨*next*⟩{⟨*default*⟩}{⟨*mandatory*⟩} (so the default will be used and the mandatory argument will be wrapped in braces).

`\ekvoptarg` expands in exactly two steps, grabs all the arguments only at the second expansion step, and is alignment safe. It has its limitations however. It can't tell the difference between `[` and `{[}`, so it doesn't work if the mandatory argument is a single bracket. Also if the optional argument should contain a nested closing bracket, the optional argument has to use nested braces like so: `[{arg]ument}]`.

Example: Say we have a macro that should take an optional argument defaulting to 1:

```
\newcommand\foo{\ekvoptarg\@foo{1}}
\newcommand\@foo[2]{Mandatory: #2\par Optional: #1}
```

\ekvoptargTF `\ekvoptargTF{⟨true⟩}{⟨false⟩}`

This macro is similar to `\ekvoptarg`, but will result in ⟨*true*⟩{⟨*optional*⟩}⟨*mandatory*⟩ or ⟨*false*⟩{⟨*mandatory*⟩} instead of placing a default value.

`\ekvoptargTF` expands in exactly two steps, grabs all the arguments only at the second expansion step, and is alignment safe. It has the same limitations as `\ekvoptarg`.

Example: Say we have a macro that should behave differently depending on whether there was an optional argument or not. This could be done with:

```
\newcommand\foo{\ekvoptargTF\foo@a\foo@b}
\newcommand\foo@a[2]{Mandatory: #2\par Optional: #1}
\newcommand\foo@b[1]{Mandatory: #1\par No optional.}
```

\ekverr `\ekverr{⟨package⟩}{⟨message⟩}`

This macro will throw an error fully expandably.[3] The error length is limited to a total length of 69 characters, and since ten characters will be added for the formatting (`!␣` and `␣Error:␣`) that leaves us with a total length for ⟨*package*⟩ plus ⟨*message*⟩ of 59 characters. If the message gets longer TEX will only display the first 69 characters and append `\ETC.` to the end.

Neither ⟨*package*⟩ nor ⟨*message*⟩ expand any further. Also ⟨*package*⟩ must not contain an explicit `\par` token or the token `\thanks@jfbu`. No such restriction applies to ⟨*message*⟩.

If `^^J` is set up as the `\newlinechar` (which is the case in LATEX 2_ε but not in plain TEX by default) you can use that to introduce line breaks in your error message. However that doesn't change the message length limit.

After your own error message some further text will be placed. The formatting of that text will look good if `^^J` is the `\newlinechar`, else not so much. That text will read:

```
! Paragraph ended before \<an-expandable-macro>
completed due to above exception.  If the error
summary is  not comprehensible  see the package
```

[3] The used mechanism was to the best of my knowledge first implemented by Jean-François Burnol.

```
documentation.
I will try to recover now.  If you're in inter-
active mode hit <return>  at the ? prompt and I
continue hoping recovery was complete.
```

Any clean up has to be done by you, `\ekverr` will expand to nothing after throwing the error message.

Example: Say we set up a small calculation which works with user input. In our calculation we need a division, so have to watch out for division by zero. If we detect such a case we throw an error and do the recovery by using the biggest integer allowed in TEX as the result.

```
\newcommand*\mydivision[2]
  {%
    \number\numexpr
      \ifnum\numexpr#2=0 % space here on purpose
        \ekverr{my}{division by 0. Setting result to 2147483647.}%
        2147483647%
      \else
        (#1)/(#2)%
      \fi
    \relax
  }
$(10+5)/(3-3)\approx\mydivision{10+5}{3-3}$
```

If that code gets executed the following will be the terminal output

```
Runaway argument?
! my Error: division by 0. Setting result to 2147483647.
! Paragraph ended before \<an-expandable-macro>
completed due to above exception.  If the error
summary is  not comprehensible  see the package
documentation.
I will try to recover now.  If you're in inter-
active mode hit <return>  at the ? prompt and I
continue hoping recovery was complete.
<to be read again>
                   \par
1.15 $(10+5)/(3-3)\approx\mydivision{10+5}{3-3}
                                                  $
?
```

and the output would contain $(10+5)/(3-3)\approx 2147483647$ if we continued the TEX run at the prompt.

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

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

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

Example: Execute the callback of the NoVal key key in set foo:
\csname\ekv@name{foo}{key}N\endcsname

1.4 Examples

1.4.1 Standard Use-Case

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

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

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

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

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

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

Finally we can use our macro like in the following

```
\ourmacro{}\par
\ourmacro{width}\par
\ourmacro{width=5pt}\par
```

```
150.0pt
145.08086pt
5.0pt
```

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

```
\input expkv-def                    % or \usepackage{expkv-def}
\ekvdefinekeys{our}
  {
```

```
    dimen     width = \ourdim,
    qdefault  width = .9\hsize,
    initial   width = 150pt
  }
```

1.4.2 A Macro to Draw Rules

Another small example could be a ⟨*key*⟩=⟨*value*⟩ driven `\rule` alternative, because I keep forgetting the correct order of its arguments. First we define the keys (and initialize the macros used to store the keys):

```
\makeatletter
\newcommand*\myrule@ht{1ex}
\newcommand*\myrule@wd{0.1em}
\newcommand*\myrule@raise{\z@}
\protected\ekvdef{myrule}{ht}{\def\myrule@ht{#1}}
\protected\ekvdef{myrule}{wd}{\def\myrule@wd{#1}}
\protected\ekvdef{myrule}{raise}{\def\myrule@raise{#1}}
\protected\ekvdef{myrule}{lower}{\def\myrule@raise{-#1}}
```

Then we define a macro to change the defaults outside of `\myrule` and `\myrule` itself:

```
\ekvsetdef\myruleset{myrule}
\newcommand*\myrule[1][]
  {\begingroup\myruleset{#1}\myrule@out\endgroup}
```

And finally the output:

```
\newcommand*\myrule@out{\rule[\myrule@raise]\myrule@wd\myrule@ht}
\makeatother
```

And we can use it:

```
a\myrule\par
a\myrule[ht=2ex,lower=.5ex]\par
\myruleset{wd=5pt}
a\myrule
```

a
a
a

1.4.3 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. The user macro `\sine` uses `\ekvoptargTF` to check for the optional argument short cutting to the final step if no optional argument was found. This way we safe some time in this case, though we have to specify the default values twice.

```
\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{\ekvoptargTF\sine@args{\sine@final{sin}{d}{3}}}
\newcommand*\sine@args[2]
  {\ekvset{expex}{#1,internal}\rnd{3}\deg{d}\f{sin}\sine@stop{#2}}
```

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

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

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

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

Let's test our macro:

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

```
0.866
0.8660254038
-1
macro:->0.017
```

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

```
\makeatletter
\newcommand*\sine{\ekvoptargTF\sine@a{\sine@b{sin}{d}{3}}}
\ekvcSplitAndForward\sine@a\sine@b
  {
    f=sin,
    unit=d,
    round=3,
  }
\ekvcSecondaryKeys\sine@a
  {
    nmeta degree={unit=d},
    nmeta radian={unit={}},
  }
\newcommand*\sine@b[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 expkvics 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 and the `\expanded` primitive are available. If it isn't, an error will be thrown using `\errmessage`:

```
! expkv Error: e-TeX and \expanded 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 using `\ekverr`. 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:

Runaway argument?
! expkv Error: unknown key '⟨key⟩' in 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:

Runaway argument?
! expkv Error: missing value for '⟨key⟩' in 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:

Runaway argument?
! expkv Error: unwanted value for '⟨key⟩' in set '⟨set⟩'

If you're using an undefined key in a set for which `\ekvredirectunknown` was used, and the key isn't found in any of the other sets as well, you'll get:

Runaway argument?
! expkv Error: no key '⟨key⟩' in sets {⟨set1⟩}{⟨set2⟩}...

If you're using an undefined `NoVal` key in a set for which `\ekvredirectunknownNoVal` was used, and the key isn't found in any of the other sets as well, you'll get:

Runaway argument?
! expkv Error: no NoVal key '⟨key⟩' in sets {⟨set1⟩}{⟨set2⟩}...

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 Bugs

Just like keyval, expkv is bug free. But if you find ~~bugs~~hidden features[4] 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

[4] Thanks, David!

1.7 Comparisons

This subsection makes some basic comparison between **expkv** and other ⟨*key*⟩=⟨*value*⟩ packages. The comparisons are really concise, regarding speed, feature range (without listing the features of each package), and bugs and misfeatures.

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

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

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

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

In this subsection is no benchmark of `\ekvparse` and `\keyval_parse:NNn` contained, as most other packages don't provide equivalent features to my knowledge. `\ekvparse` is slightly faster than `\ekvset`, but keep in mind that it does less. The same is true for `\keyval_parse:NNn` compared to `\keys_set:nn` of expl3 (where the difference is much bigger). Comparing just the two, `\ekvparse` is a tad faster than `\keyval_parse:NNn` because of the two tests (for empty key names and only a single equal sign) which are omitted.

keyval is about 30 % to 40 % faster and has a comparable feature set (actually a bit smaller since **expkv** supports unknown-key handlers and redirection to other sets) 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}}  % should be ' baz'
\setkeys{foo}{bar={{baz}}} % should be '{baz}'
```

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

ltxkeys is no longer compatible with the LaTeX kernel starting with the release 2020-10-01. It is over 380 times slower – which is funny, because it aims to be "[...] faster [...] than these earlier packages [referring to keyval and xkeyval]." It needs more time to parse zero keys than five of the packages in this comparison need to parse 100 keys. Since it aims to have a bigger feature set than xkeyval, it most definitely also has a bigger feature set than **expkv**. Also, it can't parse `\long` input, so as soon as your values contain a `\par`, it'll throw errors. Furthermore, ltxkeys doesn't strip outer braces at all by design, which, imho, is a weird design choice. In addition ltxkeys loads catoptions which is known to introduce bugs (e.g., see https://tex.stackexchange.com/questions/461783). Because it is no longer compatible with the kernel, I stop benchmarking it (so the numbers listed here and in Table 1 regarding ltxkeys were last updated on 2020-10-05).

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

pgfkeys is around 2.7 times slower for one key if one uses the `/⟨path⟩/.cd` syntax and almost 20 % slower if one uses `\pgfqkeys`, but has an *enormous* feature set. To get the best performance `\pgfqkeys` was used in the benchmark. This reduces the overhead for setting the base directory of the benchmark keys by about 43 ops (so both p_0 and T_0 would be about 43 ops bigger if `\pgfkeys{⟨path⟩/.cd,⟨keys⟩}` was used instead). It has the same or a very similar bug keyval has. The brace bug (and also the category fragility) can be fixed by pgfkeyx, but this package was last updated in 2012 and it slows down `\pgfkeys` by factor 8. Also pgfkeyx is no longer compatible with versions of pgfkeys newer than 2020-05-25.

kvsetkeys with kvdefinekeys is about 4.4 times slower, but it works even if commas and equals have category codes different from 12 (just as some other packages in this list). Else the features of the keys are equal to those of keyval, the parser has more features, though.

options is 1.7 times slower for only a single value. It has a much bigger feature set. Unfortunately it also suffers from the premature unbracing bug keyval has.

simplekv is hard to compare because I don't speak French (so I don't understand the documentation). There was an update released on 2020-04-27 which greatly improved the package's performance and adds functionality so that it can be used more like most of the other ⟨*key*⟩=⟨*value*⟩ packages. It has problems with stripping braces and spaces in a hard to predict manner just like keyval. Also, while it tries to be robust against category code changes of commas and equal signs, the used mechanism fails if the ⟨*key*⟩=⟨*value*⟩ list already got tokenised. Regarding unknown keys it got a very interesting behaviour. It doesn't throw an error, but stores the ⟨*value*⟩ in a new entry accessible with `\useKV`. Also if you omit ⟨*value*⟩ it stores `true` for that ⟨*key*⟩. For up to three keys, **expkv** is a bit faster, for more keys simplekv takes the lead.

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

Package	p_1	p_0	T_0	BB	CF	Date
keyval	13.7	1.5	7.3	yes	yes	2014-10-28
expkv	19.7	2.2	6.6	no	no	2020-10-10
simplekv	18.3	7.0	17.7	yes	yes	2020-04-27
pgfkeys	24.3	1.7	10.7	yes	yes	2020-09-05
options	23.6	15.6	20.8	yes	yes	2015-03-01
kvsetkeys	*	*	40.3	no	no	2019-12-15
l3keys	71.3	33.1	31.6	no	no	2020-09-24
xkeyval	253.6	202.2	168.3	yes	yes	2014-12-03
YAX	421.9	157.0	114.7	yes	yes	2010-01-22
ltxkeys	3400.1	4738.0	5368.0	no	no	2012-11-17

*For kvsetkeys the linear model used for the other packages is a poor fit, kvsetkeys seems to have approximately quadratic run-time, the coefficients of the second degree polynomial fit are $p_2 = 8.2$, $p_1 = 44.9$, and $p_0 = 60.8$. Of course the other packages might not really have linear run-time, but at least from 1 to 20 keys the fits don't seem too bad. If one extrapolates the fits for 100 ⟨*key*⟩=⟨*value*⟩ pairs one finds that most of them match pretty well, the exception being ltxkeys, which behaves quadratic as well with $p_2 = 23.5$, $p_1 = 2906.6$, and $p_0 = 6547.5$.

YAX is over twenty times slower. It has a pretty strange syntax for the TEX-world, imho, and again a direct equivalent is hard to define (don't understand me wrong, I don't say I don't like the syntax, it's just atypical). 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 } % setup
\setparameterlist{yax}{ height = 6 }                       % benchmark
```

1.8 License

2 Implementation

2.1 The LaTeX Package

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

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

2.2 The Generic Code

The rest of this implementation will be the generic code.

We make sure that it's only input once:

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

Check whether ε-TeX and \expanded are available – expkv requires ε-TeX.

```
\begingroup
  \edef\ekvtmpa{\string\expanded}
  \edef\ekvtmpb{\meaning\expanded}
  \expandafter
\endgroup
\ifx\ekvtmpa\ekvtmpb
\else
  \errmessage{expkv Error: e-TeX and \noexpand\expanded required}
  \expandafter\endinput
\fi
```

\ekvVersion
\ekvDate

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

```
\def\ekvVersion{1.8a}
\def\ekvDate{2021-06-03}
```

(*End definition for* \ekvVersion *and* \ekvDate*. These functions are documented on page 6.*)

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

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

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

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

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

\ekv@if@lastnamedcs

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

```
\begingroup
  \edef\ekv@tmpa{\string \lastnamedcs}
  \edef\ekv@tmpb{\meaning\lastnamedcs}
  \ifx\ekv@tmpa\ekv@tmpb
    \def\ekv@if@lastnamedcs{\long\def\ekv@if@lastnamedcs##1##2{##1}}
  \else
    \def\ekv@if@lastnamedcs{\long\def\ekv@if@lastnamedcs##1##2{##2}}
  \fi
  \expandafter
\endgroup
\ekv@if@lastnamedcs
```

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

\ekv@empty

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

```
\def\ekv@empty{}
```

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

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

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. Not all of the ones defined here are really needed by **expkv** but are provided because packages like **expkv**|DEF or **expkv**|OPT need them (and I don't want to define them in each package which might need them).

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

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

```
\long\def\ekv@ifempty#1%
```

```
  {%
    \ekv@ifempty@\ekv@ifempty@A#1\ekv@ifempty@B\ekv@ifempty@true
      \ekv@ifempty@A\ekv@ifempty@B\@secondoftwo
  }
\long\def\ekv@ifempty@#1\ekv@ifempty@A\ekv@ifempty@B{}
\long\def\ekv@ifempty@true\ekv@ifempty@A\ekv@ifempty@B\@secondoftwo#1#2{#1}
\long\def\ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo#1#2{#2}
\long\def\ekv@ifempty@true@F\ekv@ifempty@A\ekv@ifempty@B\@firstofone#1{}
\long\def\ekv@ifempty@true@F@gobble\ekv@ifempty@A\ekv@ifempty@B\@firstofone#1#2%
  {}
\long\def\ekv@ifempty@true@F@gobbletwo
    \ekv@ifempty@A\ekv@ifempty@B\@firstofone#1#2#3%
  {}
```

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

`\ekv@ifblank` `\ekv@ifblank@`

The obvious test that can be based on an if-empty is if-blank, meaning a test checking whether the argument is empty or consists only of spaces. Our version here will be tweaked a bit, as we want to check this, but with one leading `\ekv@mark` token that is to be ignored. The wrapper `\ekv@ifblank` will not be used by **expkv** for speed reasons but **expkv|opt** uses it.

```
\long\def\ekv@ifblank#1%
  {%
    \ekv@ifblank@#1\ekv@nil\ekv@ifempty@B\ekv@ifempty@true
      \ekv@ifempty@A\ekv@ifempty@B\@secondoftwo
  }
\long\def\ekv@ifblank@\ekv@mark#1{\ekv@ifempty@\ekv@ifempty@A}
```

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

`\ekv@ifdefined`

We'll need to check whether something is defined quite frequently, so why not define a macro that does this. The following test is expandable and pretty fast. The version with `\lastnamedcs` is the fastest version to test for an undefined macro I know of (that considers both undefined macros and those with the meaning `\relax`).

```
\ekv@if@lastnamedcs
  {%
    \def\ekv@ifdefined#1{\ifcsname#1\endcsname\ekv@ifdef@\fi\@secondoftwo}
    \def\ekv@ifdef@\fi\@secondoftwo
      {%
        \fi
        \expandafter\ifx\lastnamedcs\relax
          \ekv@fi@secondoftwo
        \fi
        \@firstoftwo
      }
  }
  {%
    \def\ekv@ifdefined#1%
      {%
        \ifcsname#1\endcsname\ekv@ifdef@\fi\ekv@ifdef@false#1\endcsname\relax
          \ekv@fi@secondoftwo
        \fi
        \@firstoftwo
      }
```

```
    \def\ekv@ifdef@\fi\ekv@ifdef@false{\fi\expandafter\ifx\csname}
    \long\def\ekv@ifdef@false
        #1\endcsname\relax\ekv@fi@secondoftwo\fi\@firstoftwo#2#3%
      {#3}
  }
```

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

`\ekv@strip` `\ekv@strip@a` `\ekv@strip@b` `\ekv@strip@c`

We borrow some ideas of expl3's l3tl to strip spaces from keys and values. This `\ekv@strip` also strips one level of outer braces *after* stripping spaces, so an input of `{abc}` becomes abc after stripping. It should be used with `#1` prefixed by `\ekv@mark`. Also this implementation at most strips *one* space from both sides (which should be fine most of the time, since TEX reads consecutive spaces as a single one during tokenisation).

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

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

`\ekv@exparg` `\ekv@exparg@` `\ekv@expargtwice` `\ekv@expargtwice@` `\ekv@zero`

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

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

```
\let\ekv@zero\z@
\long\def\ekv@exparg#1#2{\expandafter\ekv@exparg@\expandafter{#2}{#1}}
\long\def\ekv@exparg@#1#2{#2{#1}}%
\long\def\ekv@expargtwice#1#2{\expandafter\ekv@expargtwice@\expandafter{#2}{#1}}
\def\ekv@expargtwice@{\expandafter\ekv@exparg@\expandafter}
```

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

`\ekv@csv@loop` `\ekv@csv@loop@do` `\ekv@csv@loop@end`

This is just a very simple loop over a list of comma separated values, leaving each element as the argument to a specified function inside of `\unravel`. It should be used as `\ekv@csv@loop{`⟨*function*⟩`}\ekv@mark`⟨*csv-list*⟩`,\ekv@stop,`. We use some `\expandafter` chain to preexpand `\ekv@strip` here.

```
\ekv@exparg{\long\def\ekv@csv@loop#1#2,}%
  {%
    \expandafter
    \ekv@gobble@from@mark@to@stop
      \expandafter#\expandafter2\expandafter\ekv@csv@loop@end\expandafter
    \ekv@stop
```

```
    \ekv@strip{#2}{\ekv@csv@loop@do{#1}}%
    \ekv@csv@loop{#1}\ekv@mark
  }
\long\def\ekv@csv@loop@do#1#2{\unexpanded{#1{#2}}}
\long\expandafter\def\expandafter\ekv@csv@loop@end
    \expandafter\ekv@stop
    \ekv@strip{#1}#2%
    \ekv@csv@loop#3\ekv@mark
  {}
```

(*End definition for* `\ekv@csv@loop`, `\ekv@csv@loop@do`, *and* `\ekv@csv@loop@end`.)

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

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

```
\def\ekv@name@set#1{ekv#1(}
\def\ekv@name@key#1{#1)}
\edef\ekv@name
  {%
    \unexpanded\expandafter{\ekv@name@set{#1}}%
    \unexpanded\expandafter{\ekv@name@key{\detokenize{#2}}}%
  }
\ekv@exparg{\def\ekv@name#1#2}{\ekv@name}
```

(*End definition for* `\ekv@name`, `\ekv@name@set`, *and* `\ekv@name@key`. *These functions are documented on page* 10.)

`\ekv@undefined@set`

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

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

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

`\ekv@checkvalid`

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

```
\edef\ekv@checkvalid
  {%
    \unexpanded\expandafter{\ekv@ifempty{#1}}%
    \unexpanded
      {{%
        \def\ekv@tmp{}%
        \errmessage{expkv Error: empty set name not allowed}%
      }}%
      {%
        \unexpanded\expandafter{\ekv@ifempty{#2}}%
        \unexpanded
          {%
            {%
              \def\ekv@tmp{}%
```

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

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

`\ekvifdefined` `\ekvifdefinedNoVal`

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

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

(*End definition for* `\ekvifdefined` *and* `\ekvifdefinedNoVal`. *These functions are documented on page* 6.)

`\ekvdef` `\ekvdefNoVal` `\ekvlet` `\ekvletNoVal` `\ekvletkv` `\ekvletkvNoVal` `\ekvdefunknown` `\ekvdefunknownNoVal`

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

```
\def\ekvdef#1#2#3#4%
  {%
    \protected\long\def\ekvdef##1##2##3%
      {#1{\expandafter\def\csname#2\endcsname####1{##3}#3}}%
    \protected\long\def\ekvdefNoVal##1##2##3%
      {#1{\expandafter\def\csname#2N\endcsname{##3}#3}}%
    \protected\def\ekvlet##1##2##3%
      {#1{\expandafter\let\csname#2\endcsname##3#3}}%
    \protected\def\ekvletNoVal##1##2##3%
      {#1{\expandafter\let\csname#2N\endcsname##3#3}}%
    \ekv@expargtwice{\protected\long\def\ekvdefunknown##1##2}%
      {%
        \romannumeral
        \expandafter\ekv@exparg@\expandafter
          {%
            \expandafter\expandafter\expandafter
            \def\expandafter\csname\ekv@name{##1}{}u\endcsname####1####2{##2}%
            #3%
          }%
          {\ekv@zero\ekv@checkvalid{##1}.}%
      }%
    \ekv@expargtwice{\protected\long\def\ekvdefunknownNoVal##1##2}%
      {%
        \romannumeral
        \expandafter\ekv@exparg@\expandafter
          {%
            \expandafter\expandafter\expandafter
            \def\expandafter\csname\ekv@name{##1}{}uN\endcsname####1{##2}%
            #3%
          }%
          {\ekv@zero\ekv@checkvalid{##1}.}%
      }%
    \protected\def\ekvletkv##1##2##3##4%
```

```
      {%
        #1%
          {%
            \expandafter\let\csname#2\expandafter\endcsname
            \csname#4\endcsname
            #3%
          }%
      }%
    \protected\def\ekvletkvNoVal##1##2##3##4%
      {%
        #1%
          {%
            \expandafter\let\csname#2N\expandafter\endcsname
            \csname#4N\endcsname
            #3%
          }%
      }%
  }
\edef\ekvdefNoVal
  {%
    {\unexpanded\expandafter{\ekv@checkvalid{#1}{#2}}}%
    {\unexpanded\expandafter{\ekv@name{#1}{#2}}}%
    {%
      \unexpanded{\expandafter\ekv@defsetmacro\csname}%
      \unexpanded\expandafter{\ekv@undefined@set{#1}\endcsname{#1}}%
    }%
    {\unexpanded\expandafter{\ekv@name{#3}{#4}}}%
  }
\expandafter\ekvdef\ekvdefNoVal
```

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

`\ekvredirectunknown`
`\ekvredirectunknownNoVal`
`\ekv@defredirectunknown`
`\ekv@redirectunknown@aux`
`\ekv@redirectunknownNoVal@aux`

The redirection macros prepare the unknown function by looping over the provided list of sets and leaving a `\ekv@redirect@kv` or `\ekv@redirect@k` for each set. Only the first of these internals will receive the ⟨*key*⟩ and ⟨*value*⟩ as arguments.

```
\protected\def\ekvredirectunknown
  {%
    \ekv@defredirectunknown
      \ekv@redirect@kv
      \ekv@err@redirect@kv@notfound
      {\long\ekvdefunknown}%
      \ekv@redirectunknown@aux
  }
\protected\def\ekvredirectunknownNoVal
  {%
    \ekv@defredirectunknown
      \ekv@redirect@k
      \ekv@err@redirect@k@notfound
      \ekvdefunknownNoVal
      \ekv@redirectunknownNoVal@aux
  }
\protected\def\ekv@defredirectunknown#1#2#3#4#5#6%
  {%
    \begingroup
```

```
    \edef\ekv@tmp
      {%
        \ekv@csv@loop#1\ekv@mark#6,\ekv@stop,%
        \unexpanded{#2}%
        {\ekv@csv@loop{}\ekv@mark#5,#6,\ekv@stop,}%
      }%
    \ekv@expargtwice
      {\endgroup#3{#5}}%
      {\expandafter#4\ekv@tmp\ekv@stop}%
  }
\def\ekv@redirectunknown@aux#1{#1{##1}{##2}}
\def\ekv@redirectunknownNoVal@aux#1{#1{##1}}
```

(*End definition for* `\ekvredirectunknown` *and others. These functions are documented on page 4.*)

`\ekv@redirect@k` `\ekv@redirect@k@a` `\ekv@redirect@k@a@` `\ekv@redirect@k@b` `\ekv@redirect@k@c` `\ekv@redirect@k@d` `\ekv@redirect@kv` `\ekv@redirect@kv@a` `\ekv@redirect@kv@a@` `\ekv@redirect@kv@b` `\ekv@redirect@kv@c` `\ekv@redirect@kv@d`

The redirect code works by some simple loop over all the sets, which we already preprocessed in `\ekv@defredirectunknown`. For some optimisation we blow this up a bit code wise, essentially, all this does is `\ekvifdefined` or `\ekvifdefinedNoVal` in each set, if there is a match gobble the remainder of the specified sets and execute the key macro, else go on with the next set (to which the ⟨*key*⟩ and ⟨*value*⟩ are forwarded).

First we set up some code which is different depending on `\lastnamedcs` being available or not. All this is stored in a temporary macro to have pre-expanded `\ekv@name` constellations ready.

```
\def\ekv@redirect@k#1#2#3#4%
  {%
    \ekv@if@lastnamedcs
      {%
        \def\ekv@redirect@k##1##2##3%
          {%
            \ifcsname#1\endcsname\ekv@redirect@k@a\fi
            ##3{##1}%
          }%
        \def\ekv@redirect@k@a\fi{\fi\expandafter\ekv@redirect@k@b\lastnamedcs}%
        \long\def\ekv@redirect@kv##1##2##3##4%
          {%
            \ifcsname#2\endcsname\ekv@redirect@kv@a\fi\@gobble{##1}%
            ##4{##1}{##2}%
          }
        \def\ekv@redirect@kv@a\fi\@gobble
          {\fi\expandafter\ekv@redirect@kv@b\lastnamedcs}%
      }
      {%
        \def\ekv@redirect@k##1##2##3%
          {%
            \ifcsname#1\endcsname\ekv@redirect@k@a\fi\ekv@redirect@k@a@
              #1\endcsname
            ##3{##1}%
          }%
        \def\ekv@redirect@k@a@#3\endcsname{}%
        \def\ekv@redirect@k@a\fi\ekv@redirect@k@a@
          {\fi\expandafter\ekv@redirect@k@b\csname}%
        \long\def\ekv@redirect@kv##1##2##3##4%
          {%
            \ifcsname#2\endcsname\ekv@redirect@kv@a\fi\ekv@redirect@kv@a@
```

```
              #2\endcsname{##1}%
            ##4{##1}{##2}%
          }
        \long\def\ekv@redirect@kv@a@#4\endcsname##3{}%
        \def\ekv@redirect@kv@a\fi\ekv@redirect@kv@a@
          {\fi\expandafter\ekv@redirect@kv@b\csname}%
      }
  }
```

The key name given to this loop will already be `\detokenized` by `\ekvset`, so we can safely remove the `\detokenize` here for some performance gain.

```
\def\ekv@redirect@kv#1\detokenize#2#3\ekv@stop{\unexpanded{#1#2#3}}
\edef\ekv@redirect@kv
  {%
    {\expandafter\ekv@redirect@kv\ekv@name{#2}{#1}N\ekv@stop}%
    {\expandafter\ekv@redirect@kv\ekv@name{#3}{#2}\ekv@stop}%
    {\expandafter\ekv@redirect@kv\ekv@name{#1}{#2}N\ekv@stop}%
    {\expandafter\ekv@redirect@kv\ekv@name{#1}{#2}\ekv@stop}%
  }
```

Everything is ready to make the real definitions.

```
\expandafter\ekv@redirect@k\ekv@redirect@kv
```

The remaining macros here are independent on `\lastnamedcs`, starting from the `@b` we know that there is a hash table entry, and get the macro as a parameter. We still have to test whether the macro is `\relax`, depending on the result of that test we have to either remove the remainder of the current test, or the remainder of the set list and invoke the macro.

```
\def\ekv@redirect@k@b#1%
  {\ifx\relax#1\ekv@redirect@k@c\fi\ekv@redirect@k@d#1}
\def\ekv@redirect@k@c\fi\ekv@redirect@k@d#1{\fi}
\def\ekv@redirect@k@d#1#2\ekv@stop{#1}
\def\ekv@redirect@kv@b#1%
  {\ifx\relax#1\ekv@redirect@kv@c\fi\ekv@redirect@kv@d#1}
\long\def\ekv@redirect@kv@c\fi\ekv@redirect@kv@d#1#2{\fi}
\long\def\ekv@redirect@kv@d#1#2#3\ekv@stop{#1{#2}}
```

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

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

```
\edef\ekv@defsetmacro
  {%
    \unexpanded{\ifx#1\relax\edef#1##1}%
      {%
        \unexpanded\expandafter{\ekv@name@set{#2}}%
        \unexpanded\expandafter{\ekv@name@key{##1}}%
      }%
    \unexpanded{\fi}%
  }
\ekv@exparg{\protected\def\ekv@defsetmacro#1#2}{\ekv@defsetmacro}
```

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

\ekvifdefinedset

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

(*End definition for* `\ekvifdefinedset`. *This function is documented on page* 7.)

\ekvset Set up `\ekvset`, which should not be affected by active commas and equal signs. The equal signs are a bit harder to cope with and we'll do that later, but the active commas can be handled by just doing two comma-splitting loops one at actives one at others. 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 $=_{13}$.

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

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

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

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

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

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

else go on with other commas.

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

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

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

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

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

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

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

`\ekv@set@other` The macro `\ekv@set@other` is guaranteed to get only single ⟨*key*⟩=⟨*value*⟩ pairs.

```
\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@a
      =\ekv@mark\ekv@set@eq@active
```

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

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

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

`\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 ⟨*key*⟩=⟨*value*⟩ 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@a##1\ekv@stop
  {%
    \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@set@eq@other@active
      #2\ekv@mark\ekv@set@eq@other@b
  }
```

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 ⟨*key*⟩=⟨*value*⟩ pair to `\ekv@set@pair`. Normally we would have to also gobble an additional `\ekv@mark` after `\ekv@stop`, but this mark is needed to delimit `\ekv@set@pair`'s argument anyway, so we just leave it there.

```
\ekv@exparg
  {%
    \long\def\ekv@set@eq@other@b
        ##1\ekv@nil\ekv@mark\ekv@set@eq@other@active\ekv@stop\ekv@mark
        ##2\ekv@nil=\ekv@mark\ekv@set@eq@active
  }%
  {\ekv@strip{##1}{\expandafter\ekv@set@pair\detokenize}\ekv@mark##2\ekv@nil}
```

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

`\ekv@set@eq@other@active` `\ekv@set@eq@other@active` will be called if the ⟨*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 forward things to `\ekv@set@pair`.

```
\ekv@exparg
  {%
    \long\def\ekv@set@eq@other@active
        ##1\ekv@stop##2\ekv@nil#2\ekv@mark
        \ekv@set@eq@other@b\ekv@mark##3=\ekv@mark\ekv@set@eq@active
  }%
  {\ekv@strip{##1}{\expandafter\ekv@set@pair\detokenize}\ekv@mark##2=##3}
```

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

`\ekv@set@eq@active` `\ekv@set@eq@active@`

`\ekv@set@eq@active` 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
    ##1\ekv@nil\ekv@mark\ekv@set@eq@other@a\ekv@stop\ekv@mark
  {%
    \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@set@eq@active@
      #2\ekv@mark\ekv@set@noeq
  }
```

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`.

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

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

`\ekv@set@noeq`

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

```
\edef\ekv@set@noeq
  {%
    \unexpanded
      {%
        \ekv@ifblank@##1\ekv@nil\ekv@ifempty@B\ekv@set@was@blank
          \ekv@ifempty@A\ekv@ifempty@B
      }%
    \unexpanded\expandafter
      {\ekv@strip{##1}{\expandafter\ekv@set@key\detokenize}\ekv@mark}%
  }
\ekv@exparg
  {%
    \long\def\ekv@set@noeq
        ##1\ekv@nil\ekv@mark\ekv@set@eq@active@\ekv@stop\ekv@mark
  }%
  {\ekv@set@noeq}
\expandafter\def\expandafter\ekv@set@was@blank
    \expandafter\ekv@ifempty@A\expandafter\ekv@ifempty@B
    \ekv@strip{\ekv@mark##1}##2\ekv@mark
  {\ekv@set@other}
```

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

`\ekv@endset@other`

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

```
\long\def\ekv@endset@other
    \ekv@stop
    \ekv@eq@other\ekv@mark\ekv@stop\ekv@nil\ekv@mark\ekv@set@eq@other@a
    =\ekv@mark\ekv@set@eq@active
  {\ekv@set}
```

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

`\ekvbreak` `\ekvbreakPreSneak` `\ekvbreakPostSneak`

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

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

(*End definition for* `\ekvbreak`, `\ekvbreakPreSneak`, *and* `\ekvbreakPostSneak`. *These functions are documented on page* 7.)

`\ekvsneak` `\ekvsneakPre`

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

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

(*End definition for* `\ekvsneak` *and* `\ekvsneakPre`. *These functions are documented on page* 7.)

`\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. The `\unexpanded\expanded{{...}}` ensures that the material is in an alignment safe group at all time, and that it doesn't expand any further in an `\edef` or `\expanded` context.

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

(*End definition for* `\ekvparse`. *This function is documented on page* 6.)

`\ekv@parse`

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

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

`\ekv@endparse`

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

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

`\ekv@parse@other`

```
\long\def\ekv@parse@other##1##2##3,%
  {%
    \ekv@gobble@from@mark@to@stop##3\ekv@endparse@other\ekv@stop
    \ekv@eq@other##3\ekv@nil\ekv@mark\ekv@parse@eq@other@a
      =\ekv@mark\ekv@parse@eq@active
    {##1}{##2}%
    \ekv@mark
  }
```

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

`\ekv@parse@eq@other@a`
`\ekv@parse@eq@other@b`

```
\long\def\ekv@parse@eq@other@a##1\ekv@stop
  {%
    \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@parse@eq@other@active
      #2\ekv@mark\ekv@parse@eq@other@b
  }
\ekv@exparg
  {%
    \long\def\ekv@parse@eq@other@b
        ##1\ekv@nil\ekv@mark\ekv@parse@eq@other@active\ekv@stop\ekv@mark
        ##2\ekv@nil=\ekv@mark\ekv@parse@eq@active
  }%
  {\ekv@strip{##1}\ekv@parse@pair##2\ekv@nil}
```

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

`\ekv@parse@eq@other@active`

```
\ekv@exparg
  {%
    \long\def\ekv@parse@eq@other@active
        ##1\ekv@stop##2\ekv@nil#2\ekv@mark
        \ekv@parse@eq@other@b\ekv@mark##3=\ekv@mark\ekv@parse@eq@active
  }%
  {\ekv@strip{##1}\ekv@parse@pair##2=##3}
```

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

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

```
\long\def\ekv@parse@eq@active
    ##1\ekv@nil\ekv@mark\ekv@parse@eq@other@a\ekv@stop\ekv@mark
  {%
    \ekv@eq@active##1\ekv@nil\ekv@mark\ekv@parse@eq@active@
      #2\ekv@mark\ekv@parse@noeq
  }
\ekv@exparg
  {\long\def\ekv@parse@eq@active@##1\ekv@stop##2#2\ekv@mark\ekv@parse@noeq}%
  {\ekv@strip{##1}\ekv@parse@pair##2}
```

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

`\ekv@parse@noeq`

```
\edef\ekv@parse@noeq
  {%
    \unexpanded
      {%
        \ekv@ifblank@##1\ekv@nil\ekv@ifempty@B\ekv@parse@was@blank
          \ekv@ifempty@A\ekv@ifempty@B
      }%
      \unexpanded\expandafter{\ekv@strip{##1}\ekv@parse@key}%
  }
\ekv@exparg
  {%
    \long\def\ekv@parse@noeq
```

```
          ##1\ekv@nil\ekv@mark\ekv@parse@eq@active@\ekv@stop\ekv@mark
  }%
  {\ekv@parse@noeq}
\expandafter\def\expandafter\ekv@parse@was@blank
    \expandafter\ekv@ifempty@A\expandafter\ekv@ifempty@B
    \ekv@strip{\ekv@mark##1}\ekv@parse@key
  {\ekv@parse@other}
```

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

`\ekv@endparse@other`

```
\long\def\ekv@endparse@other
    \ekv@stop
    \ekv@eq@other\ekv@mark\ekv@stop\ekv@nil\ekv@mark\ekv@parse@eq@other@a
    =\ekv@mark\ekv@parse@eq@active
  {\ekv@parse}
```

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

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

```
\ekv@exparg{\long\def\ekv@parse@pair##1##2\ekv@nil}%
  {\ekv@strip{##2}\ekv@parse@pair@{##1}}
\long\def\ekv@parse@pair@##1##2##3##4%
  {%
    \unexpanded{##4{##2}{##1}}%
    \ekv@parse@other{##3}{##4}%
  }
```

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

`\ekv@parse@key`

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

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

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

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

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

```
\edef\ekvsetSneaked
  {%
    \unexpanded{\ekvsneak{#2}}%
    \unexpanded\expandafter{\ekvset{#1}{#3}}%
  }
\ekv@expargtwice{\long\def\ekvsetSneaked#1#2#3}{\ekvsetSneaked}
```

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

\ekvchangeset Provide a macro that is able to switch out the current ⟨*set*⟩ in \ekvset. This operation allows something similar to pgfkeys's ⟨*key*⟩/.cd mechanism. However this operation can be more expensive than /.cd as we can't just redefine some token to reflect this, but have to switch out the set expandably, so this works similar to the \ekvsneak macros reading and reinserting things, but it only has to read and reinsert the remainder of the current key's replacement code.

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

(*End definition for* \ekvchangeset*. This function is documented on page* 7.)

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

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

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

\ekv@set@pair
\ekv@set@pair@a
\ekv@set@pair@b
\ekv@set@pair@c
\ekv@set@pair@d
\ekv@set@pair@e

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

```
\ekv@if@lastnamedcs
  {%
    \long\def\ekv@set@pair#1\ekv@mark#2\ekv@nil#3%
      {%
        \ifcsname #3{#1}\endcsname\ekv@set@pair@a\fi\@secondoftwo
          {#2}%
          {%
            \ifcsname #3{}u\endcsname\ekv@set@pair@a\fi\@secondoftwo
              {#2}%
              {%
                \ekv@ifdefined{#3{#1}N}%
                  \ekv@err@noarg
                  \ekv@err@unknown
                  #3%
              }%
              {#1}%
          }%
        \ekv@set@other#3%
      }
    \def\ekv@set@pair@a\fi\@secondoftwo
      {\fi\expandafter\ekv@set@pair@b\lastnamedcs}
  }
  {%
    \long\def\ekv@set@pair#1\ekv@mark#2\ekv@nil#3%
```

```
      {%
        \ifcsname #3{#1}\endcsname
            \ekv@set@pair@a\fi\ekv@set@pair@c#3{#1}\endcsname
          {#2}%
          {%
            \ifcsname #3{}u\endcsname
                \ekv@set@pair@a\fi\ekv@set@pair@c#3{}u\endcsname
              {#2}%
              {%
                \ekv@ifdefined{#3{#1}N}%
                  \ekv@err@noarg
                  \ekv@err@unknown
                  #3%
              }%
              {#1}%
          }%
        \ekv@set@other#3%
      }
    \def\ekv@set@pair@a\fi\ekv@set@pair@c{\fi\expandafter\ekv@set@pair@b\csname}
    \long\def\ekv@set@pair@c#1\endcsname#2#3{#3}
  }
\long\def\ekv@set@pair@b#1%
  {%
    \ifx#1\relax
      \ekv@set@pair@e
    \fi
    \ekv@set@pair@d#1%
  }
\ekv@exparg{\long\def\ekv@set@pair@d#1#2#3}{\ekv@strip{#2}#1}
\long\def\ekv@set@pair@e\fi\ekv@set@pair@d#1#2#3{\fi#3}
```

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

`\ekv@set@key` `\ekv@set@key@a` `\ekv@set@key@b` `\ekv@set@key@c`

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.

```
\ekv@if@lastnamedcs
  {%
    \long\def\ekv@set@key#1\ekv@mark#2%
      {%
        \ifcsname #2{#1}N\endcsname\ekv@set@key@a\fi\@firstofone
          {%
            \ifcsname #2{}uN\endcsname\ekv@set@key@a\fi\@firstofone
              {%
                \ekv@ifdefined{#2{#1}}%
                  \ekv@err@reqval
                  \ekv@err@unknown
                  #2%
              }%
              {#1}%
          }%
        \ekv@set@other#2%
      }
    \def\ekv@set@key@a\fi\@firstofone{\fi\expandafter\ekv@set@key@b\lastnamedcs}
```

```
  }
  {%
    \long\def\ekv@set@key#1\ekv@mark#2%
      {%
        \ifcsname #2{#1}N\endcsname
            \ekv@set@key@a\fi\ekv@set@key@c#2{#1}N\endcsname
          {%
            \ifcsname #2{}uN\endcsname
                \ekv@set@key@a\fi\ekv@set@key@c#2{}uN\endcsname
              {%
                \ekv@ifdefined{#2{#1}}%
                  \ekv@err@reqval
                  \ekv@err@unknown
                  #2%
              }%
              {#1}%
          }%
        \ekv@set@other#2%
      }
    \def\ekv@set@key@a\fi\ekv@set@key@c{\fi\expandafter\ekv@set@key@b\csname}
    \long\def\ekv@set@key@c#1N\endcsname#2{#2}
  }
\long\def\ekv@set@key@b#1%
  {%
    \ifx#1\relax
      \ekv@fi@secondoftwo
    \fi
    \@firstoftwo#1%
  }
```

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

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

```
\ekv@expargtwice{\protected\def\ekvsetdef#1#2}%
  {%
    \romannumeral
    \ekv@exparg{\ekv@zero\ekv@exparg{\long\def#1##1}}%
      {\ekvset{#2}{##1}}%
  }
```

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

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

```
\ekv@expargtwice{\protected\def\ekvsetSneakeddef#1#2}%
  {%
    \romannumeral
    \ekv@exparg{\ekv@zero\ekv@exparg{\long\def#1##1##2}}%
      {\ekvsetSneaked{#2}{##1}{##2}}%
  }
\ekv@expargtwice{\protected\def\ekvsetdefSneaked#1#2#3}%
```

```
  {%
    \romannumeral
    \ekv@exparg{\ekv@zero\ekv@exparg{\long\def#1##1}}%
      {\ekvsetSneaked{#2}{#3}{##1}}%
  }
```

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

`\ekv@alignsafe` `\ekv@endalignsafe`

These macros protect the usage of ampersands inside of alignment contexts.

```
\begingroup
\catcode`\^^@=2
\@firstofone{\endgroup
  \def\ekv@alignsafe{\romannumeral\iffalse{\fi`^^@ }
}
\def\ekv@endalignsafe{\ifnum`{=\ekv@zero}\fi}
```

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

`\ekvoptarg` `\ekvoptargTF`

Provide macros to expandably collect an optional argument in brackets. The macros here are pretty simple in nature compared to xparse's possibilities (they don't care for nested bracket levels).

We start with a temporary definition to pre-expand `\ekv@alignsafe` (will be `#1`) and `\ekv@endalignsafe` (will be `#2`).

```
\begingroup
\def\ekvoptarg#1#2{%
\endgroup
```

The real definition starts an expansion context and afterwards grabs the arguments. `#1` will be the next step, `#2` the default value, and `#3` might be an opening bracket, or the mandatory argument. We check for the opening bracket, if it is found grab the optional argument, else leave `#1{#2}` in the input stream after ending the expansion context.

```
\def\ekvoptarg{\romannumeral#1\ekv@optarg@a}
\long\def\ekv@optarg@a##1##2##3%
  {%
    \ekv@optarg@if\ekv@mark##3\ekv@mark\ekv@optarg@b\ekv@mark[\ekv@mark
    #2%
    \@firstofone{\ekv@zero##1}{##2}{##3}%
  }%
```

The other variant of this will do roughly the same. Here, `#1` will be the next step if an optional argument is found, `#2` the next step else, and `#3` might be the opening bracket or mandatory argument.

```
\def\ekvoptargTF{\romannumeral#1\ekv@optargTF@a}
\long\def\ekv@optargTF@a##1##2##3%
  {%
    \ekv@optarg@if\ekv@mark##3\ekv@mark\ekv@optargTF@b{##1}\ekv@mark[\ekv@mark
    #2%
    \@firstofone{\ekv@zero##2}{##3}%
  }
```

The two macros to grab the optional argument have to remove the remainder of the test and the wrong next step as well as grabbing the argument.

```
\long\def\ekv@optarg@b\ekv@mark[\ekv@mark\ifnum`##1\fi\@firstofone##2##3##4##5]%
  {#2##2{##5}}
\long\def\ekv@optargTF@b
```

```
    ##1\ekv@mark[\ekv@mark\ifnum`##2\fi\@firstofone##3##4##5]%
  {#2\ekv@zero##1{##5}}
}
```

Do the definitions and add the test macro.

```
\ekv@exparg{\expandafter\ekvoptarg\expandafter{\ekv@alignsafe}}\ekv@endalignsafe
\long\def\ekv@optarg@if#1\ekv@mark[\ekv@mark{}
```

(*End definition for* `\ekvoptarg` *and* `\ekvoptargTF`*. These functions are documented on page 8.*)

`\ekverr` `\ekv@err@collect` `\ekv@err@cleanup`

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 a runaway argument (to my knowledge the first version of this method was implemented by Jean-François Burnol, many thanks to him). The first step is to ensure that the second argument (which might contain user input) doesn't contain tokens we use as delimiters (in this case `\par`), this will be done by the front facing macro `\ekverr`. But first we set some other things up.

We use a temporary definition for `\ekverr` to get multiple consecutive spaces. Then we set up the macro that will collect the error and the macro that will throw the error. The latter will have an unreasonable long name. This way we can convey more information. Though the information in the macro name is static and has to be somewhat general to fit every occurence. The important bit is that the long named macro has a delimited argument and is short which will throw the error at the `\par` at the end of `\ekv@err@collect`. This macro has the drawback that it will only print nicely if the `\newlinechar` is `^^J`.

```
\def\ekv@err@cleanup\par{}
\def\ekv@err@collect#1%
  {%
    \def\ekv@err@collect##1\par##2%
      {%
        \expandafter
        \ekv@err@cleanup
        #1! ##2 Error: ##1\par
      }%
    \def#1##1\thanks@jfbu{}%
  }
\def\ekverr{ }
\expandafter\ekv@err@collect\csname <an-expandable-macro>^^J%
  completed due to above exception. \ekverr If the error^^J%
  summary is \ekverr not comprehensible \ekverr see the package^^J%
  documentation.^^J%
  I will try to recover now. \ekverr If you're in inter-^^J%
  active mode hit <return> \ekverr at the ? prompt and I^^J%
  continue hoping recovery\endcsname
\long\def\ekverr#1#2{\expandafter\ekv@err@collect\detokenize{#2}\par{#1}}
```

(*End definition for* `\ekverr`*,* `\ekv@err@collect`*, and* `\ekv@err@cleanup`*. These functions are documented on page 8.*)

`\ekv@err`

We define a shorthand to throw errors in expkv.

```
\ekv@exparg{\long\def\ekv@err#1}{\ekverr{expkv}{#1}}
```

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

\ekv@err@common
\ekv@err@common@
\ekv@err@unknown
\ekv@err@noarg
\ekv@err@reqval

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

```
\long\def\ekv@err@common #1#2{\expandafter\ekv@err@common@\string#2{#1}}
\ekv@exparg{\long\def\ekv@err@common@#1‘#2’ #3.#4#5}%
  {\ekv@err{#4 ‘#5’ in set ‘#2’}}
\ekv@exparg{\long\def\ekv@err@unknown#1}{\ekv@err@common{unknown key}{#1}}
\ekv@exparg{\long\def\ekv@err@noarg  #1}{\ekv@err@common{unwanted value for}{#1}}
\ekv@exparg{\long\def\ekv@err@reqval #1}{\ekv@err@common{missing value for}{#1}}
\ekv@exparg{\long\def\ekv@err@redirect@kv@notfound#1#2#3\ekv@stop}%
  {\ekv@err{no key ‘#2’ in sets #3}}
\ekv@exparg{\def\ekv@err@redirect@k@notfound#1#2\ekv@stop}%
  {\ekv@err{no NoVal key ‘#1’ in sets #2}}
```

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

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

```
\catcode‘\@=\ekv@tmp
```

Index

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