expkv|bundle

Jonathan P. Spratte*

2023-01-23

Abstract

The expkv|bundle provides at its core a *fully expandable* ⟨*key*⟩=⟨*value*⟩ parser, that is *safe* for active commas and equals signs, *reliable* to only strip one set of braces after spaces are stripped, and blazingly *fast*, as of writing this only keyval is faster.

This parser gets augmented by a family of packages. expkv|cs allows to easily define expandable macros using a ⟨*key*⟩=⟨*value*⟩ interface, making the expandable parser available to the masses. expkv|def provides a ⟨*key*⟩=⟨*value*⟩ interface to define common ⟨*key*⟩-types. With expkv|opt you can parse package and class options. expkv|pop enables you to design your own prefix oriented parsers for interface definitions.

Contents

*jspratte@yahoo.de; Special thanks to निरंजन (Niranjan) for valuable suggestions and additions to this documentation.

Introduction

This bundle consists of different packages the base being **expkv**. Most of these packages are available for plain TEX, LATEX 2_ε, and ConTEXt. For stylistic reasons the package names are printed as **expkv**|⟨*pkg*⟩, but the files are named `expkv-`⟨*pkg*⟩ (CTAN-rules don't allow | in names), so in order to load **expkv**|cs in LATEX 2_ε you should use

```
\usepackage{expkv-cs}
```

Each section describing a package of this bundle has next to its heading the formats in which they work printed flush right. If more than a single format is supported by a package the functionality is defined by the plain TEX variant and the other variants only load the generic code in a way suitable for the format.

Terminology

This documentation uses a few terms which always mean specific things:

⟨*key*⟩=⟨*value*⟩ **pair** is one element in a comma separated list which contains at least one equals sign *not* contained in any braces, and the first such equals sign is the separator between the ⟨*key*⟩ (with an optional ⟨*expansion*⟩ prefix) and the ⟨*value*⟩.

⟨*key*⟩ means the entire left-hand side of a ⟨*key*⟩=⟨*value*⟩ pair with an optional ⟨*expansion*⟩ prefix stripped, or if =⟨*value*⟩ is omitted the complete list element, again with an ⟨*expansion*⟩ prefix stripped.

⟨*key*⟩**-name** synonymous to ⟨*key*⟩.

`Val-`⟨*key*⟩ describes a ⟨*key*⟩ in a ⟨*key*⟩=⟨*value*⟩ pair.

`NoVal-`⟨*key*⟩ describes a ⟨*key*⟩ for which =⟨*value*⟩ was or should be omitted.

⟨*value*⟩ is the right-hand side of a ⟨*key*⟩=⟨*value*⟩ pair.

⟨*key*⟩=⟨*value*⟩ **list** is a comma separated list containing ⟨*key*⟩=⟨*value*⟩ pairs and `NoVal-`⟨*key*⟩s (each possibly with an ⟨*expansion*⟩ prefix).

{⟨*key*⟩=⟨*value*⟩, ...} an argument that should get a ⟨*key*⟩=⟨*value*⟩ list.

⟨*expansion*⟩ **prefix** an optional prefix in front of ⟨*key*⟩ to specify ⟨*expansion*⟩-rules (see subsubsection 1.1.1), that prefix consists of the ⟨*expansion*⟩-rules followed by a colon immediately followed by a space.

⟨*expansion*⟩ a list of tokens specifying expansion steps for ⟨*key*⟩ and ⟨*value*⟩.

⟨*expansion*⟩**-rule** a single expansion step in the ⟨*expansion*⟩-rules.

⟨*expansion*⟩**-rules** synonymous to ⟨*expansion*⟩.

exp:NOTATION the notation of ⟨*expansion*⟩-rules in form of the ⟨*expansion*⟩ prefix.

key-code the code that is executed for a given ⟨*key*⟩.

key-macro the internal macro that stores the key-code.

Though not really terminology but more typographic representation I want to highlight a distinction between two different types of code listings in this documentation. I use the following looks to show a code example and its results:

```
\newcommand*\foo{This is an example.}
\foo
```

This is an example.

And this is how a syntax summary or a syntax example looks like (this is more abstract than an example and contains short meta-descriptions of inputs):

```
\function{⟨syntax⟩}
```

Inside such syntax summaries the following rules usually apply (and ⟨*arg*⟩ would be the meta description here):

`{`⟨*arg*⟩`}` a mandatory argument is shown in braces

⟨*arg*⟩ a mandatory argument that should be a single token is shown without additional parentheses/braces/brackets

`[`⟨*arg*⟩`]` an optional argument is shown in brackets (and should be input with brackets)

⟨`*`⟩ an optional star is shown like this

If other types of arguments are displayed the documentation will explain what they mean in this particular place.

Category Codes

Supporting different category codes of a single character code makes the programmer's life harder in TEX, but there are valid reasons to make some active, or letter. Because of this the packages in this bundle support different category codes for specific syntax relevant characters (unless otherwise documented). This doesn't mean that **expkv** changes any category codes, only that parsing is correct *if* they are changed later (the codes listed assume standard category codes of plain TEX and LATEX 2ε apply while **expkv** is loaded). The supported tokens are:

`=` $=_{12}$ and $=_{13}$

`,` $,_{12}$ and $,_{13}$

`:` (for the **exp:notation**) $:_{11}$, $:_{12}$, and $:_{13}$

`*` (for starred macros) $*_{3}$, $*_{4}$, $*_{6}$, $*_{7}$, $*_{8}$, $*_{11}$, $*_{12}$, and $*_{13}$

`[` (for `\ekvoptarg`) only $[_{12}$

`]` (for `\ekvoptarg`) only $]_{12}$

Bugs

Just like keyval, **expkv** is bug free. But if you find ~~bugs~~hidden features[1] you can tell me about them either via mail (see the first page) or directly on GitLab if you have an account there: https://gitlab.com/islandoftex/texmf/expkv-bundle

[1] Thanks, David!

expkv|bundle for the Impatient

This section gives a very brief and non-exhaustive overview over the contents of the expkv|bundle. For more information (and more functionality) you'll have to read the sections of the packages you're interested in.

expkv|bundle supports expansion control in ⟨*key*⟩=⟨*value*⟩ lists. The corresponding syntax and features are documented in subsubsection 1.1.1.

The following user interface macros (and more) are available in the different packages of the bundle:

Defining keys

- `\ekvdefinekeys{`⟨*set*⟩`}{`⟨*key*⟩`=`⟨*value*⟩`, ...}` defines the keys in the ⟨*key*⟩=⟨*value*⟩ list, many common key types are available (subsection 3.1 and for the types subsubsection 3.2.2).
- `\ekvdef{`⟨*set*⟩`}{`⟨*key*⟩`}{`⟨*code*⟩`}` defines the behaviour of a `Val-`⟨*key*⟩ (subsection 1.2).
- `\ekvdefNoVal{`⟨*set*⟩`}{`⟨*key*⟩`}{`⟨*code*⟩`}` defines the behaviour of a `NoVal-`⟨*key*⟩ (subsection 1.2).

Parsing ⟨*key*⟩=⟨*value*⟩ lists

- `\ekvset{`⟨*set*⟩`}{`⟨*key*⟩`=`⟨*value*⟩`, ...}` sets defined keys (subsection 1.5).
- `\ekvparse{`⟨*k-code*⟩`}{`⟨*kv-code*⟩`}{`⟨*key*⟩`=`⟨*value*⟩`, ...}` parses the ⟨*key*⟩=⟨*value*⟩ list and runs ⟨*k-code*⟩ or ⟨*kv-code*⟩ on the elements (subsection 1.6).

Defining expandable ⟨*key*⟩=⟨*value*⟩ macros

- `\ekvcSplit`⟨*cs*⟩`{`⟨*key*⟩`=`⟨*value*⟩`, ...}{`⟨*code*⟩`}` defines a fully expandable macro with the keys in the ⟨*key*⟩=⟨*value*⟩ list, values are accessed by `#1` to `#9` (subsubsection 2.1.2).
- `\ekvcHash`⟨*cs*⟩`{`⟨*key*⟩`=`⟨*value*⟩`, ...}{`⟨*code*⟩`}` defines a fully expandable macro with the keys in the ⟨*key*⟩=⟨*value*⟩ list, values are accessed using `\ekvcValue{`⟨*key*⟩`}{#1}` (subsubsection 2.1.3).
- `\ekvcSecondaryKeys`⟨*cs*⟩`{`⟨*key*⟩`=`⟨*value*⟩`, ...}` defines additional keys of predefined types for a ⟨*cs*⟩ defined with `\ekvcSplit` or `\ekvcHash` (subsection 2.2 and for the types subsubsection 2.2.2).

Parsing options (subsection 4.1)

- `\ekvoProcessOptions{`⟨*set*⟩`}` processes the global options, and the options given to the current and all future calls of the package.
- `\ekvoProcessGlobalOptions{`⟨*set*⟩`}` processes the global options.
- `\ekvoProcessLocalOptions{`⟨*set*⟩`}` processes the local options of a package or class.
- `\ekvoProcessFutureOptions{`⟨*set*⟩`}` processes the options of future calls of the package.

1 expkv

```
\input{expkv} % plain
\usepackage{expkv} % LaTeX
\usemodule[expkv] % ConTeXt
```

This package supports two different front ends to parse a ⟨*key*⟩=⟨*value*⟩ list. The first (`\ekvset`) is similar to keyval's `\setkeys`, it parses the list and executes defined actions based on the encountered ⟨*key*⟩s. The second (`\ekvparse`) is more versatile, it only splits the list into ⟨*key*⟩s and ⟨*value*⟩s and then runs user-provided code on the result.

The first is described in subsections 1.2 to 1.5, the latter is described in subsection 1.6.

Unlike the other packages in the bundle, if you load expkv as a LaTeX 2ε package there is a single option available:

`all` `\usepackage[all]{expkv}`

Loads all the packages of expkv|bundle.

1.1 General Parsing Rules

expkv parses a ⟨*key*⟩=⟨*value*⟩ list by first splitting the elements on commas (active or other), then looking for an equals sign (active or other). If there is one the ⟨*key*⟩=⟨*value*⟩ pair will be split at the first. From both ⟨*key*⟩ and ⟨*value*⟩ (if there was a ⟨*value*⟩) one set of outer spaces is stripped, and afterwards one set of outer braces (meaning braces which are around the complete remainder after space stripping if there are any).

So the syntax looks something like the following pseudo-input:

␣{⟨*key*⟩}␣=␣{⟨*value*⟩}␣

with the displayed spaces and braces being optional and removed if found. Note that if you want either ⟨*key*⟩ or ⟨*value*⟩ to include a comma the braces become mandatory, the same is true if ⟨*key*⟩ should contain an equals sign.

1.1.1 Expansion Control

expkv provides a mechanism to specify expansions of a ⟨*key*⟩ and/or ⟨*value*⟩. For those familiar with pgfkeys this is similar to its `.expand once` or `.expanded` handlers. This concept will be called exp:notation or ⟨*expansion*⟩ throughout this documentation.

The syntax for this notation is a leading list of ⟨*expansion*⟩-rules followed by a colon that is immediately followed by a space. Also the ⟨*expansion*⟩-rules must not contain any spaces outside of braces, and the remainder on the right hand side of the colon must not be blank, else it is not considered an exp:notation but just a weirdly formed ⟨*key*⟩-name.

The entire syntax of a ⟨*key*⟩=⟨*value*⟩ pair is

␣{⟨*expansion*⟩:␣␣{⟨*key*⟩}␣}␣=␣{⟨*value*⟩}␣

Note that the ⟨*expansion*⟩ prefix is right delimited by :␣ so the space after the colon is only optional in the sense that the entire ⟨*expansion*⟩ prefix is optional. Else all displayed spaces and braces are optional, the inner set of spaces and braces around ⟨*key*⟩ only being optional if the optional ⟨*expansion*⟩ prefix (⟨*expansion*⟩:␣) was present. If that part was present the list of ⟨*expansion*⟩-rules will be executed, which might change the contents of both ⟨*key*⟩ and ⟨*value*⟩. For `\ekvparse` this is always true, however in `\ekvset` it is only parsed for the exp:notation if there is no ⟨*key*⟩ matching the given input (so this notation doesn't impose a restriction on key names, though ⟨*key*⟩-names

actually containing what would otherwise be an ⟨*expansion*⟩ prefix should be pretty rare in practice).

All packages in expkv|bundle support this notation (most of them internally use `\ekvset` or `\ekvparse`). Please note however that while expkv|opt fully supports them, reinsertion via the `\r` ⟨*expansion*⟩-rule might affect the unused global options list if used in the class options.

An ⟨*expansion*⟩-rule consists of a single token. In a `Val`-⟨*key*⟩ they work on the ⟨*value*⟩ (but you can use the `\key` rule to also affect the ⟨*key*⟩ there) while in a `NoVal`-⟨*key*⟩ they work on the ⟨*key*⟩. The following rules are available (those familiar with expl3 will notice that the first six are identical to its argument types):

`o` Expands the first token once.

`e` Expands the entire ⟨*value*⟩ inside of `\expanded`.

`c` Builds a `\csname` from the contents.

`f` Expands the contents until a space or an unexpandable token is found (the space would be removed).

`V` The ⟨*value*⟩ should be a single token, either defined as a parameterless macro or as a register (via `\newcount` etc.). This expands to the value of the register or the macro's replacement text. If the token in ⟨*value*⟩ has the `\meaning` of `\relax` an error is thrown and the result is empty.

`v` This is a combination of `c` and `V`, meaning the ⟨*value*⟩ is turned into a single control sequence via `\csname`, and then expanded to its value. The control sequence will only be built if it's defined.

Example: Say we want to hand the contents of a macro as the value to our key, but the actual macro name depends on user input. For this we have two options which behave slightly different. One is to use `v` the other is to combine the `co` ⟨*expansion*⟩-rules. The following demonstrates both (I modified the way errors are thrown to instead output them in red for this; you'll learn about `\ekvparse` in a few pages, for now just stick with me):

```
\newcommand\mypair[2]{Arg: `\detokenize{#2}'. }%
\newcommand\myvalue{Value}%
\ekvparse\@firstofone\mypair
  {
     co: key = myvalue, v: key = myvalue, \par
    ,co: key = myValue, v: key = myValue, \par
  }
```

```
Arg: `Value'. Arg: `Value'.
Arg: `\myValue '. ! expkv Error: Erroneous variable `\myValue' usedArg: ``.
```

The difference is that in `co` the variable is implicitly initialised as `\relax` by `c` if it doesn't exist and then doesn't expand in `o`. On the other hand `v` will check whether the variable would exist and throw an error if it doesn't (and will not set it to `\relax` by blindly using `\csname`).

`s` Strips one set of outer spaces and outer braces.

`b` Adds one set of outer braces.

`p` `p{`⟨*contents*⟩`}`

Places ⟨*contents*⟩ before the ⟨*value*⟩.

`P` `P{`⟨*contents*⟩`}`

Places ⟨*contents*⟩ after the ⟨*value*⟩.

`g` Gobbles the first token or balanced group on the left (leads to a low-level TEX-error if the ⟨*value*⟩ is empty).

`\r` In a `Val-`⟨*key*⟩ reinserts the contents of ⟨*value*⟩ after all the ⟨*expansion*⟩-rules were executed (the ⟨*key*⟩-name needs to be empty). In a `NoVal-`⟨*key*⟩ the contents of ⟨*key*⟩ are reinserted after all the ⟨*expansion*⟩-rules were executed (the ⟨*value*⟩ needs to be empty, which is an easy to fulfil rule as there was no ⟨*value*⟩). Normal ⟨*key*⟩=⟨*value*⟩ parsing is aborted afterwards for the current ⟨*key*⟩=⟨*value*⟩ list element.

Example: Say we want to store a list of common settings in a macro, then we want to parse a few keys, insert the contents of the macro, and parse a few more keys. The following does exactly that (`\ekvset` is analogue to `\setkeys` of the keyval package if you're familiar with it, else you'll learn about `\ekvset` a few pages down the road so be patient):

```
\newcommand*\mykeylist{color=red,height=5cm}
\ekvset{mypkg}{key=value, o\r: \mykeylist, other key=other value}
```

You could also use the following with the same outcome, but this looks more complicated so the other form should be preferred:

```
\ekvset{mypkg}{key=value, o\r: {}=\mykeylist, other key=other value}
```

`\key` `\key{`⟨*expansion*⟩`}`

This is the only supported way to change the contents of ⟨*key*⟩ for a `Val-`⟨*key*⟩ in the **exp:**NOTATION. All the rules in ⟨*expansion*⟩ are applied to ⟨*key*⟩ instead of ⟨*value*⟩.

`R` This is the same as if you used `V\r`. So it expects a single token, retrieves its value, and reinserts this as additional ⟨*key*⟩=⟨*value*⟩ input.

`r` This is the same as if you used `v\r`. So it builds a `\csname` if that is defined, retrieves its value, and reinserts this as additional ⟨*key*⟩=⟨*value*⟩ input.

Example: Now that we also know the `R` and `r` rule, the example above can be input even simpler:

```
\ekvset{mypkg}{key=value, R: \mykeylist, other key=other value}
```

or

```
\ekvset{mypkg}{key=value, r: mykeylist, other key=other value}
```

1.2 Setting up Keys

expkv provides a rather simple approach to setting up keys, similar to keyval. If you're looking for a more sophisticated interface similar to those of l3keys or pgfkeys take a look at expkv|def described in section 3 or for a simple interface that defines expandable macros at expkv|cs described in section 2.

Keys in expkv (as in many other ⟨*key*⟩=⟨*value*⟩ implementations) belong to a *set*, so 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 a value (we call those `Val-`⟨*key*⟩) and keys that don't (which are called `NoVal-`⟨*key*⟩ by expkv), but both can share the same name on the user level, the only difference for the user is whether =⟨*value*⟩ was used or not.

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). And prefixes allowed for `\let` can prefix those macros 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`. Also ⟨*set*⟩ should not contain an explicit `\par` token.

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

Defines a `Val-`⟨*key*⟩ 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@text{#1}}
```

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

Defines a `NoVal-`⟨*key*⟩ in ⟨*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 `Val-`⟨*key*⟩ in ⟨*set*⟩ to ⟨*cs*⟩. There are no checks on ⟨*cs*⟩ enforced, but the code should expect the ⟨*value*⟩ as a single braced argument directly following it.

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

```
\ekvlet{foo}{cmd}\foo@cmd
```

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

Let the `NoVal-⟨key⟩` in ⟨*set*⟩ to ⟨*cs*⟩. Again no checks on ⟨*cs*⟩ are done. It shouldn't expect any provided argument.

Example: See above.

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

Copies the definition such that `Val-⟨key⟩` in ⟨*set*⟩ behaves like ⟨*key2*⟩ of ⟨*set2*⟩. It is not checked whether that second key exists!

Example: Let `B` in `bar` do the same as `A` in `foo`:

```
\ekvletkv{bar}{B}{foo}{A}
```

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

And this lets the `NoVal-⟨key⟩` in ⟨*set*⟩ to the definition of the `NoVal-⟨key2⟩` in ⟨*set2*⟩. Again, it is not checked whether the second key exists.

Example: See above.

1.3 Handle Unknown Keys

By default expkv throws an error message if it encounters an undefined ⟨*key*⟩. You can change this behaviour with the macros listed here. Just like in the section above, prefixes for `\def` are allowed if the macro has `def` in its name, and `\let` prefixes are allowed if the macro is named something with `let`.

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

Execute ⟨*code*⟩ if an undefined `Val-⟨key⟩` is encountered while parsing in ⟨*set*⟩. You can refer to the given ⟨*value*⟩ with `#1`, the unknown ⟨*key*⟩'s name with `#2` (will be `\detokenized`), and to the ⟨*key*⟩'s name without `\detokenize` applied with `#3` in ⟨*code*⟩ (this order is chosen for performance reasons).

`\ekvdefunknown` and `\ekvredirectunknown` are mutually exclusive, you can't use both.

Example: Also search `bar` for undefined keys of set `foo` (and use the not yet `\detokenized` ⟨*key*⟩'s name in case the undefined key handler of `bar` needs that):

```
\long\ekvdefunknown{foo}{\ekvset{bar}{{#3}={#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⟩}`

With this you can let expkv execute ⟨*code*⟩ if an unknown `NoVal-⟨key⟩` was encountered. You can refer to the given ⟨*key*⟩ with `#1` (will be `\detokenized`), and to the not `\detokenized` ⟨*key*⟩'s name with `#2`.

`\ekvdefunknownNoVal` and `\ekvredirectunknownNoVal` are mutually exclusive, you can't use both.

Example: Adding to the above also handling of `NoVal-⟨key⟩`s in `foo`:

```
\ekvdefunknownNoVal{foo}{\ekvset{bar}{{#2}}}
```

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

This is a short cut to set up a special \ekvdefunknown-rule for ⟨set⟩ that will check each set in the comma separated ⟨set-list⟩ for an unknown Val-⟨key⟩. The resulting unknown-key handler will always be \long and *not* \protected. The first set in ⟨set-list⟩ has highest priority, once the Val-⟨key⟩ is found in one of the sets the remainder of the list is discarded. If ⟨key⟩ isn't found in any of the sets an error will be thrown eventually. Note that the error message looks different than a normal key-not-found error, in particular no unwanted-value message can be thrown (it will not be checked if a NoVal-⟨key⟩ of the same name does exist), and the error message will contain all sets.

\ekvdefunknown and \ekvredirectunknown are mutually exclusive, you can't use both.

Example: For every undefined Val-⟨key⟩ in foo also search the sets bar and baz:

```
\ekvredirectunknown{foo}{bar, baz}
```

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

This behaves just like \ekvredirectunknown, it does the same but for NoVal-⟨key⟩s. Again no prefixes are supported (the result will neither be \long nor \protected). Note that the error messages will not check whether a missing-value error should be thrown.

\ekvdefunknownNoVal and \ekvredirectunknownNoVal are mutually exclusive, you can't use both.

Example: See above.

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

This lets the handler for unknown Val-⟨key⟩s to ⟨cs⟩. ⟨cs⟩ should expect three arguments, the first will be the ⟨value⟩ the second the \detokenized ⟨key⟩-name, the third the unprocessed ⟨key⟩-name. No conditions on ⟨cs⟩ are enforced.

Example: Let the set foo do the same as the macro \foo@unknown whenever an unknown Val-⟨key⟩ is encountered:

```
\ekvletunknown{foo}\foo@unknown
```

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

This does the same as \ekvletunknown but for NoVal-⟨key⟩s. The ⟨cs⟩ should expect two arguments, namely the \detokenized ⟨key⟩ and the unprocessed ⟨key⟩.

Example: Let the set foo ignore unknown NoVal-⟨key⟩s by gobbling the ⟨key⟩-name:

```
\ekvletunknownNoVal{foo}\@gobbletwo
```

1.4 Helpers in Actions

`\ekvifdefined` `\ekvifdefinedNoVal`

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

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

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}}{}
```

`\ekvsneak` `\ekvsneakPre`

`\ekvsneak{`⟨*after*⟩`}`

Puts ⟨*after*⟩ after the effects of `\ekvset` (without cancelling the current `\ekvset` call). 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. After `\ekvset` has parsed the entire ⟨*key*⟩=⟨*value*⟩ list everything that has been `\ekvsneaked` will be left in the input stream.

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

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

A more elaborate usage example is shown in subsubsection 1.9.2.

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

`\ekvbreak{`⟨*after*⟩`}`

Gobbles the remainder of the current `\ekvset` call and its argument list and inserts ⟨*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}}}
```

`\ekvmorekv`

`\ekvmorekv{`⟨*key*⟩=⟨*value*⟩`, ...}`

Adds the contents of the ⟨*key*⟩=⟨*value*⟩ list to the list processed by the current call of `\ekvset`.

Example: Define a `NoVal-`⟨*key*⟩ `style` that sets the keys `border`, `width`, and `height` as a shortcut:

```
\ekvdefNoVal{foo}{style}{\ekvmorekv{border, width=2cm, height=1.5ex}}
```

`\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}}}
```

1.5 Parsing Keys in Sets

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

This macro parses the ⟨*key*⟩=⟨*value*⟩ list and checks for defined ⟨*key*⟩s that are in ⟨*set*⟩. Unlike the generic `\ekvparse` this macro uses `\detokenize` on the ⟨*key*⟩ before checking whether it is a defined key.

`\ekvset` is nestable, and fully expandable. But it is *not* alignment safe. As a result ⟨*key*⟩ names and ⟨*value*⟩s that contain an `&` must be wrapped in braces if `\ekvset` is used inside an alignment (like LaTeX 2ε's `tabular` environment) or alternatively you have to create a wrapper that ensurs an alignment safe context.

Example: Parse `key=arg, key` in set `foo`:

```
\ekvset{foo}{key=arg, key}
```

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

This behaves like `\ekvset` in which `\ekvsneak` was immediately called.

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

```
\ekvsetSneaked{foo}{\afterwards}{key=arg, key}
```

`\ekvsetdef` `\ekvsetdef`⟨*cs*⟩`{`⟨*set*⟩`}`

Defines the macro ⟨*cs*⟩ to be a shortcut for `\ekvset{`⟨*set*⟩`}`. You can use any TeX-prefix allowed to prefix `\def` for `\ekvsetdef` (so `\long`, `\protected`, or `\global` – don't use `\outer`). The resulting macro is faster than but else equivalent to the idiomatic definition:

```
\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 will make it a shorthand for `\ekvsetSneaked`, meaning ⟨*cs*⟩ will take two arguments (first the `\ekvsneak` argument, then the ⟨*key*⟩=⟨*value*⟩ list). Hence the result is a faster version of:

```
\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⟩}`

This macro behaves like `\ekvsetSneakeddef`, but with a fixed ⟨*sneaked*⟩ argument. So the resulting macro is faster than but else equivalent to

```
\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}
```

\ekvcompile `\ekvcompile⟨*⟩⟨cs⟩⟨parameters⟩{⟨set⟩}{⟨key⟩=⟨value⟩, ...}`

This macro defines ⟨*cs*⟩ to be a *fast* way to set the given ⟨*key*⟩=⟨*value*⟩ list in ⟨*set*⟩. The meaning of the keys is frozen if you don't give the optional `*` (if the star is present the stored content will be the key-macros and later redefinitions of keys will affect them, otherwise the key-macros are expanded once, hence the key-code is stored). This does support the unknown key handlers set up with `\ekvdefunknown` and `\ekvdefunknownNoVal` and also the redirection of unknown keys (the latter will not be expanded exhaustively though, so the key-search is done on every later call of ⟨*cs*⟩). Any prefix allowed for `\def` might prefix `\ekvcompile`. The list is not entirely fixed, as you might use ⟨*parameters*⟩ in a ⟨*value*⟩ (this is not a single token but a parameter text as you'd use it with `\def`). They can not be part of a ⟨*key*⟩-name (the names are indeed fixed). If you need a `#` in a ⟨*value*⟩ you'll need to double it just as you'd do in `\def`. Internally `\ekvcompile` uses `\ekvparse` and no `\ekvset` variant, because of this the **exp:NOTATION** is handled slightly differently; in case you're using a ⟨*key*⟩-name that starts with something that looks like **exp:NOTATION** you'll have to explicitly add an empty ⟨*expansion*⟩ prefix.

Example: Define the macro `\foo` to set some keys in the set `foo`. Since one key has a strange name we need to add an empty ⟨*expansion*⟩ prefix. Also we'd like `\foo` to take one parameter which is part of the ⟨*value*⟩ of `bar` (since the list is parsed now and not when `\foo` is used we don't need to put braces around that value, even if at use time `#1` contains commas):

```
\ekvcompile\foo#1{foo}
  {
     bar = #1baz
    ,: part-of-key: name = strange
    ,NoVal
  }
```

After this using `\foo{VAL}` will be the same as but faster than

```
\ekvset{foo}{bar={VALbaz},part-of-key: name=strange,NoVal}
```

1.6 Generic Key Parsing

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

This macro parses the ⟨*key*⟩=⟨*value*⟩ list and provides `NoVal-`⟨*key*⟩s to ⟨*code1*⟩ as a single argument and `Val-`⟨*key*⟩s with their corresponding ⟨*value*⟩ as two arguments to ⟨*code2*⟩.

`\ekvparse` is fully expandable and alignment safe, meaning that you don't have to take any extra precautions if it is used inside an alignment context (like LaTeX 2ε's `tabular` environment) and any ⟨*key*⟩ or ⟨*value*⟩ can contain an `&`. `\ekvparse` expands in exactly two steps, the result is provided inside `\unexpanded` (so doesn't expand further in an `\edef` or `\expanded` context).

`\ekvbreak`, `\ekvsneak`, `\ekvmorekv`, *etc.* don't work in `\ekvparse`. `\ekvparse` does not throw an error if multiple unprotected equals signs are found (it just splits at the first), and doesn't throw an error if a ⟨*key*⟩ is empty. If something looks like exp:NOTATION (has a colon followed but not preceded by a space and with non-blank material following it) it'll be parsed as such (which might throw errors due to undefined ⟨*expansion*⟩-rules if that wasn't the intended input). If you for some reason need to input a ⟨*key*⟩-name that would match that pattern you'll need to precede it by `:␣` (an empty ⟨*expansion*⟩ prefix).

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 keys. No such macros are contained in expkv, but I hope you get the idea. Because it expands in two steps and doesn't expand any further both

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

and

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

expand to

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

1.7 Other Useful Macros

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

This macro will expandably check for a following optional argument in brackets (`[]`). After the optional argument there has to be a mandatory one (or else this might have unwanted side effects). The code in ⟨*next*⟩ should expect two arguments (or more), namely the processed optional argument and the mandatory one that followed it. 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 after it was read once – if it was already wrapped it is effectively unchanged).

`\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 it has to be nested in braces like so: `[{arg[u]ment}]` (or else the result would be `arg[u` with a trailing `ment]`).

Example: Say we have a macro that should take an optional argument defaulting to `1`, we could program it like this:

```
\newcommand\foo{\ekvoptarg\@foo{1}}
\newcommand\@foo[2]{Mandatory: #2\par Optional: #1}
\foo{5}\par
\foo[4]{5}\par
```

Mandatory: 5
Optional: 1
Mandatory: 5
Optional: 4

\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.}
\foo{5}\par
\foo[4]{5}\par
```

Mandatory: 5
No optional.
Mandatory: 5
Optional: 4

\ekvcsvloop `\ekvcsvloop{⟨code⟩}{⟨csv-list⟩}`

This loops over the comma separated items in ⟨*csv-list*⟩ and, after stripping spaces from either end of ⟨*item*⟩ and removing at most one set of outer braces, leaves `\unexpanded{⟨code⟩{⟨item⟩}}` for each list item in the input stream. Blank elements are ignored (if you need a blank element it should be given as {␣}). It supports both active commas and commas of category other. `\ekvcsvloop` is not alignment safe, but you could make it so by nesting it in `\expanded` (since the braces around the argument of `\expanded` will hide alignment characters from TEX's parsing).

Example: The following splits a comma separated list and prints it in a typewriter font with parentheses around each element:

```
\newcommand*\myprocessor[1]{\texttt{(#1)}}
\ekvcsvloop\myprocessor{abc,def,ghi}\par
\ekvcsvloop\myprocessor{1,,2,,3,,4}\par
```

```
(abc)(def)(ghi)
(1)(2)(3)(4)
```

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

This macro will throw an error fully expandably.[2] 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 of ⟨*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 2ε 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
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.

In ConTEXt this macro works differently. While still being fully expandable, it doesn't have the character count limitation and doesn't impose restrictions on ⟨*package*⟩. It will not display the additional text and adding line breaks is not possible.

Example: Say we set up a macro that takes as mandatory argument a simple equation which must not be empty and if it's not empty it displays it and calculates the result:

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

```
\newcommand*\mycalc[1]
  {%
    \the\numexpr
    \if\relax\detokenize{#1}\relax
      \ekverr{my}{Empty equation not allowed, leaving -2147483647}%
      -2147483647%
    \else
      #1%
    \fi
    \relax
  }
Using \mycalc{} wrong.
```

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

```
Runaway argument?
! my Error: Empty equation not allowed, leaving -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
1.17 Using \mycalc{}
                       wrong.
?
```

and the output would contain `Using -2147483647 wrong` if we continued the TEX run at the prompt.

1.8 Other Macros

`\ekvDate` `\ekvVersion`

These two macros store expkv's date and version.

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

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

The names of the macros storing the code of `Val-`⟨*key*⟩s are stored in are built with these macros. The name is built from two blocks, one that is formatting the ⟨*set*⟩ name, and on for formatting the ⟨*key*⟩ name. To get the actual name the argument to `\ekv@name@key` must be `\detokenize`d. Both blocks are put together (with the necessary `\detokenize`) by `\ekv@name`. For `NoVal-`⟨*key*⟩s an additional `N` gets appended, 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 key-macro of the `NoVal-`⟨*key*⟩ named `bar` in set `foo`:

```
\csname\ekv@name{foo}{bar}N\endcsname
```

1.9 Examples

1.9.1 Standard Use-Case

Example: Because I keep forgetting the correct order of LaTeX 2ε's `\rule` command I want to create a ⟨*key*⟩=⟨*value*⟩ interface to it. For this I define the keys `ht` to specify the rule's height, `wd` to specify its width, and to give a displacement I use two keys (because who can remember whether the rule is moved upwards or downwards?).

First the internals storing the values are initialised

```
\makeatletter
\newcommand*\myrule@ht{1ex}
\newcommand*\myrule@wd{0.1em}
\newcommand*\myrule@raise{\z@}
```

then the keys are defined. We could use `\dimen` registers instead of defining macros, but macros have the advantage that the font dependent dimensions are evaluated at use time.

```
\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}}
```

We also want a way to change the initial values without outputting a rule (since there are unexpandable keys involved it's a good idea to define this `\protected`)

```
\protected\ekvsetdef\myruleset{myrule}
```

and we need an actual frontend that does the job:

```
\newcommand*\myrule[1][]
  {%
    \begingroup
      \myruleset{#1}%
      \rule[\myrule@raise]{\myrule@wd}{\myrule@ht}%
    \endgroup
  }
\makeatother
```

Now we can use it:

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

1.9.2 An Expandable ⟨*key*⟩=⟨*value*⟩ Macro Using \ekvsneak

Example: 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 we'll use \ekvsneakPre for the real values as well. If for some reason we wanted a key for which the first usage was the binding one we'd use \ekvsneak for that one.

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 our end marker.

Ordering the keys can be done simply by searching for a specific token for each argument (that token acts as a flag), so our sneaked out values will include these specific tokens acting as markers.

Now we got an answer to each of our initial problems. Everything that's left is deciding what our macro should actually do. For this example we'll define a macro that calculates the sine of a number rounded to a specified precision. The macro should also understand input in radian and degree, and we could also decide to evaluate a different function. For the real hard part of this (expandably calculating trigonometric functions) we'll use xfp.

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 default values right before it. The user macro \sine uses \ekvoptargTF to check for the optional argument short cutting a bit if no optional argument was found. If you'd so prefer you could use ltcmd's \NewExpandableDocumentCommand to expandably get an optional argument as well.

```
\RequirePackage{xfp}
\makeatletter
\ekvdef{sine}{f}{\ekvsneakPre{\f{#1}}}
\ekvdef{sine}{round}{\ekvsneakPre{\rnd{#1}}}
\ekvdefNoVal{sine}{degree}{\ekvsneakPre{\deg{d}}}
\ekvdefNoVal{sine}{radian}{\ekvsneakPre{\deg{}}}
\ekvdefNoVal{sine}{internal}{\ekvsneakPre{\sine@rnd}}
\newcommand*\sine{\ekvoptargTF\sine@args{\sine@final{sin}{d}{3}}}
\newcommand*\sine@args[2]
  {\ekvset{sine}{#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 argument after the first matching flag). 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 part is pretty easy after the build up we've done until now. In `\fpeval` the trigonometric functions have names such as `sin` or `cos`, and the degree taking alternatives just have an appended `d` (so `sind` or `cosd`). So putting ⟨*f*⟩ and ⟨*degree/radian*⟩ together will form the correct names.

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

Let's give our macro a test:

```
\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
```

Please note that setting this up a lot more user friendly is easily possible by utilizing expkvics (see section 2).

```
\input{expkv-cs} % plain
\usepackage{expkv-cs} % LaTeX
\usemodule[expkv-cs] % ConTeXt
```

expkv-cs aids in creating fully expandable macros that take a ⟨*key*⟩=⟨*value*⟩ argument. It implements somewhat efficient solutions to expandable ⟨*key*⟩=⟨*value*⟩ parsing without the user having to worry too much about the details.

The package supports two main approaches for this. The first is splitting the keys up into individual arguments, the second preparses the ⟨*key*⟩=⟨*value*⟩ list into a single argument in which accessing the value of individual keys is fast. The behaviour of the second type is similar to a hash table, so we call that variant `Hash`, the first type is called `Split`. Both these variants support a number of so called *primary keys* (a primary key matches an argument, roughly speaking).

In addition to these methods there is a structured way to define additional keys which might build upon the primary keys but not directly relate to an argument. These keys are called *secondary keys*. Primary and secondary keys belong to a specific macro (the macro name serves as the *set*).

A word of advice you should consider: Macros defined with expkv-cs are simple to create, and there might be good use cases for them (for instance since they don't work by assignments but only by argument forwarding logic they have no issues with implicit or explicit groups whatsoever). But they don't scale as well as established ⟨*key*⟩=⟨*value*⟩ interfaces (think of the idiomatic key definitions with keyval, or l3keys, or expkv with or without expkv-def), and they are slower than idiomatic key definitions in packages with fast ⟨*key*⟩=⟨*value*⟩ parsing.

2.1 Defining Macros and Primary Keys

All macros defined with expkv-cs have to be previously undefined (or have the `\meaning` of `\relax`). There is no way to automatically undefine keys once they are set up – so to make sure there are no conflicts only new definitions are allowed. The *set* name (as used by `\ekvset`) will be `\string`\⟨*macro*⟩.

2.1.1 Primary Keys

The notion of primary keys needs a bit of explanation, or better, the input syntax for the argument ⟨*primary keys*⟩ in the following explanations. The ⟨*primary keys*⟩ argument should be a ⟨*key*⟩=⟨*value*⟩ list in which each ⟨*key*⟩ will be one primary key and ⟨*value*⟩ the initial value of said ⟨*key*⟩ (and that value is mandatory, even if you leave it blank that's fine, but you have to explicitly state it). By default all keys are defined short, but you can define `\long` keys by prefixing ⟨*key*⟩ with `long` (*e.g.*, `long name=Jonathan P. Spratte` to define a `\long` key called `name`). Due to some internal implementations it's worth noting that `\long` keys are a microscopic grain faster. The ⟨*cs*⟩ will only be defined `\long` if at least one of the keys was `\long`. For obvious reasons there is no interface in place to define something as `\protected`.

To allow keys to start with the word `long` even if you don't want them to be `\long` you can also prefix them with `short`. The first found prefix of the two will stop the parsing for prefixes and what remains becomes the ⟨*key*⟩.

These rules culminate in the following:

```
\ekvcSplit\foo
  {
    long short = abc\par
```

```
    ,short long = def
  }
  {#1#2}
```

will define a macro \foo that knows two primary keys, short which is defined \long (so will accept explicit \par tokens inside its value at use time), and long which doesn't accept explicit \par tokens (leading to a low level TEX error). The description of \ekvcSplit follows shortly.

There is one exception to the rule that each ⟨*key*⟩ in ⟨*primary keys*⟩ needs to get a value: If you include a key named ... without a value this will be a primary key in which every unknown key will be collected – and ⟨*cs*⟩ will be defined \long. The unknown keys will be stored in a way that *most* ⟨*key*⟩=⟨*value*⟩ parsers will parse them correctly (but this is no general guarantee, for instance pgfkeys can accidentally strip multiple sets of braces at the wrong moment). See some examples in subsection 2.4.

At the moment expkvics doesn't require any internal keys, but I can't foresee whether this will be the case in the future as well, as it might turn out that some features I deem useful can't be implemented without such internal keys. Because of this, please don't use key names starting with EKVC| as that should be the private name space.

2.1.2 Split

The split variants will provide the key values as separate arguments. This limits the number of keys for which this is truly useful.

\ekvcSplit `\ekvcSplit⟨cs⟩{⟨primary keys⟩}{⟨definition⟩}`

This defines ⟨*cs*⟩ to be a macro taking one mandatory argument which should contain a ⟨*key*⟩=⟨*value*⟩ list. The ⟨*primary keys*⟩ will be defined for this macro (see subsubsection 2.1.1). The ⟨*definition*⟩ is the code that will be executed. You can access the ⟨*value*⟩ of a ⟨*key*⟩ by using a macro parameter from #1 to #9. The order of the macro parameters will be the order provided in the ⟨*primary keys*⟩ list (so #1 is the ⟨*value*⟩ of the ⟨*key*⟩ defined first). With \ekvcSplit you can define macros using at most nine primary keys.

Example: The following defines a macro \foo that takes the keys a and b and outputs their values in a textual form:

```
\ekvcSplit\foo{a=a,b=b}{a is #1.\par b is #2.\par}
\foo{}
\foo{b=e}
```

a is a.
b is b.
a is a.
b is e.

\ekvcSplitAndForward `\ekvcSplitAndForward⟨cs⟩{⟨after⟩}{⟨primary keys⟩}`

This defines ⟨*cs*⟩ to be a macro taking one mandatory argument which should contain a ⟨*key*⟩=⟨*value*⟩ list. You can use as many primary keys as you want with this. The primary keys will be forwarded to ⟨*after*⟩ as braced arguments (as many as necessary for your primary keys). The order of the braced arguments will be the order of your primary key definitions. In ⟨*after*⟩ you can use just a single control sequence, or some arbitrary stuff which will be left in the input stream before your braced values (with one set of braces stripped from ⟨*after*⟩), so both of the following would be fine:

```
\ekvcSplitAndForward\foo\foo@aux{keyA = A, keyB = B}
\ekvcSplitAndForward\foo{\foo@aux{more args}}{keyA = A, keyB = B}
```

In the first case `\foo@aux` should take at least two arguments (`keyA` and `keyB`), in the second case at least three (`more args`, `keyA`, and `keyB`).

\ekvcSplitAndUse `\ekvcSplitAndUse⟨cs⟩{⟨primary keys⟩}`

This will roughly do the same as `\ekvcSplitAndForward`, but instead of specifying what will be used after splitting the keys, ⟨*cs*⟩ will use what follows the ⟨*key*⟩=⟨*value*⟩ list. So its syntax will be

⟨*cs*⟩{⟨*key*⟩=⟨*value*⟩, ...}{⟨*after*⟩}

and the code in `after` should expect at least as many arguments as the number of keys defined for ⟨*cs*⟩.

2.1.3 Hash

The hash variants will provide the key values as a single argument in which you can access specific values using a special macro. The implementation might be more convenient and scale better, *but* it is slower (for a rudimentary macro with a single key benchmarking was almost 1.7 times slower, the root of which being the key access with `\ekvcValue`, not the parsing, and for a key access using `\ekvcValueFast` it was still about 1.2 times slower). Still to be future proof, considering the hash variants is a good idea, and to get best performance but less maintainable code you can resort to the split approach.

\ekvcHash `\ekvcHash⟨cs⟩{⟨primary keys⟩}{⟨definition⟩}`

This defines ⟨*cs*⟩ to be a macro taking one mandatory argument which should contain a ⟨*key*⟩=⟨*value*⟩ list. You can use as many primary keys as you want. The primary keys will be forwarded as a single argument containing every key to the underlying macro. The underlying macro is defined as ⟨*definition*⟩, in which you can access the ⟨*value*⟩ of a ⟨*key*⟩ by using `\ekvcValue{⟨key⟩}{#1}` (or similar).

Example: This defines an equivalent macro to the `\foo` defined with `\ekvcSplit` earlier:

```
\ekvcHash\foo{a=a,b=b}{a is \ekvcValue{a}{#1}.\par
                       b is \ekvcValue{b}{#1}.\par}
\foo{}
\foo{b=e}
```

a is a.
b is b.
a is a.
b is e.

\ekvcHashAndForward `\ekvcHashAndForward⟨cs⟩{⟨after⟩}{⟨primary keys⟩}`

This defines ⟨*cs*⟩ to be a macro taking one mandatory argument which should contain a ⟨*key*⟩=⟨*value*⟩ list. You can use as many primary keys as you want. The primary keys will be forwarded as a single argument containing every key to ⟨*after*⟩. You can use a single macro for ⟨*after*⟩ or use some arbitrary stuff, which will be left in the input stream before the hashed ⟨*key*⟩=⟨*value*⟩ list with one set of braces stripped. In the macro called in ⟨*after*⟩ you can access the ⟨*value*⟩ of a ⟨*key*⟩ by using `\ekvcValue{⟨key⟩}{#1}` (or whichever argument the hashed ⟨*key*⟩=⟨*value*⟩ list will be in).

Example: This defines a macro `\foo` processing two keys, and passing the result to `\foobar`:

```
\ekvcHashAndForward\foo\foobar{a=a,b=b}
\newcommand*\foobar[1]{a is \ekvcValue{a}{#1}.\par
                       b is \ekvcValue{b}{#1}.\par}
\foo{}
\foo{b=e}
```

a is a.
b is b.
a is a.
b is e.

\ekvcHashAndUse `\ekvcHashAndUse⟨cs⟩{⟨primary keys⟩}`

This will roughly do the same as `\ekvcHashAndForward`, but instead of specifying what will be used after hashing the keys during the definition, ⟨*cs*⟩ will use what follows the ⟨*key*⟩=⟨*value*⟩ list. So its syntax will be

⟨*cs*⟩{⟨*key*⟩=⟨*value*⟩, ...}{⟨*after*⟩}

\ekvcValue `\ekvcValue{⟨key⟩}{⟨key list⟩}`

This is a safe way to access your keys in a hash variant. ⟨*key*⟩ is the key which's ⟨*value*⟩ you want to use out of the ⟨*key list*⟩. ⟨*key list*⟩ should be the key list argument forwarded to your underlying macro by `\ekvcHash`, `\ekvcHashAndForward`, or `\ekvcHashAndUse`. It will be tested whether the hash function to access that ⟨*key*⟩ exists, the ⟨*key*⟩ argument is not empty, and that the ⟨*key list*⟩ really contains a ⟨*value*⟩ of that ⟨*key*⟩. This macro needs exactly two steps of expansion and if used inside of an `\edef` or `\expanded` context will protect the ⟨*value*⟩ from further expanding.

\ekvcValueFast `\ekvcValueFast{⟨key⟩}{⟨key list⟩}`

This behaves similar to `\ekvcValue`, but *without any* safefy tests. As a result this is about 1.4 times faster *but* will throw low level TEX errors eventually if the hash function isn't defined or the ⟨*key*⟩ isn't part of the ⟨*key list*⟩ (*e.g.*, because it was defined as a key for another macro – all macros share the same hash function per ⟨*key*⟩ name). Note that this will not only throw low level errors but result in undefined behaviour as well! This macro needs exactly three steps of expansion in the no-error case.

\ekvcValueSplit `\ekvcValueSplit{⟨key⟩}{⟨key list⟩}{⟨next⟩}`

If you need a specific ⟨*key*⟩ from a ⟨*key list*⟩ more than once, it'll be a good idea to only extract it once and from then on keep it as a separate argument (or if you want to forward this value to another macro). Hence the macro `\ekvcValueSplit` will extract one specific ⟨*key*⟩'s ⟨*value*⟩ from the list and forward it as an argument to ⟨*next*⟩, so the result of this will be ⟨*next*⟩{⟨*value*⟩}. This is roughly as fast as `\ekvcValue` and runs the same tests.

Example: The following defines a macro `\foo` which will take three keys. Since the next parsing step will need the value of one of the keys multiple times we split that key off the list (in this example the next step doesn't use the key multiple times for simplicity though), and the entire list is forwarded as the second argument:

```
\ekvcHash\foo{a=a,b=b,c=c}
  {\ekvcValueSplit{a}{#1}\foobar{#1}}
\newcommand*\foobar[2]{a is #1.\par
                       b is \ekvcValue{b}{#2}.\par
                       c is \ekvcValue{c}{#2}.\par}
\foo{}
```

a is a.
b is b.
c is c.

`\ekvcValueSplitFast` `\ekvcValueSplitFast{⟨key⟩}{⟨key list⟩}{⟨next⟩}`

This behaves just like `\ekvcValueSplit`, but it won't run the safety tests, hence it is faster but more error prone, just like the relation between `\ekvcValue` and `\ekvcValueFast`.

2.2 Secondary Keys

To lift some limitations of each primary key matching one argument or hash entry, you can define secondary keys. Those have to be defined for each macro individually but it doesn't matter whether that macro was set up as a split or hash variant.

Secondary keys can have a *prefix* (`long`), and must have a *type* (like `meta`). Some *types* might require some *prefix* while other *types* might forbid the usage of a specific *prefix*.

Please keep in mind that key names shouldn't start with `EKVC|`.

`\ekvcSecondaryKeys` `\ekvcSecondaryKeys⟨cs⟩{⟨key⟩=⟨value⟩, ...}`

This is the front facing macro to define secondary keys. For the macro ⟨*cs*⟩ define ⟨*key*⟩ to have definition ⟨*value*⟩. The general syntax for ⟨*key*⟩ should be

⟨*prefix*⟩ ⟨*name*⟩

Where ⟨*prefix*⟩ is a space separated list of optional *prefixes* followed by one *type*. The syntax of ⟨*value*⟩ is dependent on the used *type*.

2.2.1 *Prefixes*

Currently there is only one *prefix* available, which is

`long` The following key will be defined `\long`.

2.2.2 *Types*

Compared to expkv|def you might notice that the *types* here are much fewer. Unfortunately the expansion only concept doesn't allow for great variety in the common key *types*.

The syntax examples of the *types* will show which *prefix* will be automatically used by printing those black (`long`), which will be available in grey (`long`), and which will be disallowed in red (`long`). This will be put flush right next to the syntax line.

If a secondary key references another key it doesn't matter whether that other key is a primary or secondary key (unless explicitly stated otherwise).

meta `meta` ⟨*key*⟩ `= {`⟨*key*⟩`=`⟨*value*⟩`, ...}` long

With a `meta` key you can set other keys. Whenever ⟨*key*⟩ is used the keys in the ⟨*key*⟩=⟨*value*⟩ list will be set to the values given there. You can use the ⟨*value*⟩ given to ⟨*key*⟩ by using `#1` in the ⟨*key*⟩=⟨*value*⟩ list.

nmeta `nmeta` ⟨*key*⟩ `= {`⟨*key*⟩`=`⟨*value*⟩`, ...}` long

An `nmeta` key is like a `meta` key, but it doesn't take a value at use time, so the ⟨*key*⟩=⟨*value*⟩ list is static.

alias `alias` ⟨*key*⟩ `= {`⟨key_2⟩`}` long

This assigns the definition of ⟨key_2⟩ to ⟨*key*⟩. As a result ⟨*key*⟩ is an alias for ⟨key_2⟩ behaving just the same. Both the `Val-`⟨*key*⟩ and the `NoVal-`⟨*key*⟩ will be copied if they are defined when `alias` is used. Of course, ⟨key_2⟩ has to be defined as at least one of `NoVal-`⟨*key*⟩ or `Val-`⟨*key*⟩.

default `default` ⟨*key*⟩ `= {`⟨*default*⟩`}` long

If ⟨*key*⟩ is defined as a `Val-`⟨*key*⟩ you can define a `NoVal-`⟨*key*⟩ version with this. The `NoVal-`⟨*key*⟩ will behave as if ⟨*key*⟩ was given ⟨*default*⟩ as its ⟨*value*⟩. Note that this doesn't change the initial values of primary keys set at definition time (see `\ekvcChange` in subsection 2.3 for this). If ⟨*key*⟩ isn't yet defined this results in an error.

enum `enum` ⟨*key*⟩ `= {`⟨key_2⟩`}{`⟨*key*⟩`=`⟨*value*⟩`, ...}` long

This defines ⟨*key*⟩ to only accept the values given in the list of the second argument of its definition. It forwards the position of ⟨*value*⟩ in that list to ⟨key_2⟩ (zero-based). The ⟨key_2⟩ has to already be defined by the time an `enum` key is set up. Each ⟨*value*⟩ in the list (and at use time) is `\detokenized`, so no expansion takes place here.

If you use `enum` twice on the same ⟨*key*⟩ the new values will again start at zero (so it is possible to define multiple values with the same outcome), however since you can't skip values you'll have to use the same as in the first call for values with just a single variant. There is no interface to delete existing values.

Example: First a small example that might give you an idea of what the description above could mean:

```
\ekvcSplit\foo{k-internal=-1}{#1}
\ekvcSecondaryKeys\foo
  {enum k = {k-internal}{a,b,c}}
\foo{}\foo{k=a}\foo{k=b}\foo{k=c}
```

```
-1012
```

Example: We can define a choice setup that might do different things based on the choice encountered, and the numeric value is easy to parse using `\ifcase`:

```
\ekvcSplit\foo{k-internal=-1}
  {%
    \ifcase#1
      is\or
      This\or
      easy%
    \else
      .%
    \fi
  }
\ekvcSecondaryKeys\foo
  {enum k = {k-internal}{a,b,c}}
\foo{k=b} \foo{k=a} \foo{k=c}\foo{}
```

This is easy.

choice

`choice` ⟨*key*⟩ `= {`⟨*key*$_2$⟩`}{`⟨*key*⟩`=`⟨*value*⟩`, ...}` long

This is pretty similar to an `enum`, but unlike with `enum` the forwarded ⟨*value*⟩ will not be numeric, instead the ⟨*value*⟩ as given during the definition time will be forwarded. This means that while the user input has to match in a `\detokenized` form, the ⟨*value*⟩ might still expand further during your macro's expansion (if what you provided as a choice is expandable).

Example: We could use this to filter out the possible vertical placements of a LaTeX 2$_\varepsilon$ `tabular`:

```
\ekvcSplit\foo{v-internal=c,a=t,b=c,c=b}
  {%
    \begin{tabular}[#1]{@{} c @{:} c @{}}
      a & #2\\
      b & #3\\
      c & #4\\
    \end{tabular}%
  }
\ekvcSecondaryKeys\foo
  {choice v = {v-internal}{t,c,b}}
\foo{} \foo{v=t} \foo{v=c} \foo{v=b}
```

a:t
a:t a:t b:c
b:c a:t b:c c:b
c:b b:c c:b
c:b

aggregate
e-aggregate

`aggregate` ⟨*key*⟩ `= {`⟨*primary*⟩`}{`⟨*definition*⟩`}` long

While other key *types* replace the current value of the associated primary key, with `aggregate` you can create keys that append or prepend (or whatever you like) the new value to the current one. Your definition of an `aggregate` key must be exactly two TeX arguments, where ⟨*primary*⟩ should be the name of a primary key, and ⟨*definition*⟩ the way you want to store the current and the new value. Inside ⟨*definition*⟩ you can use `#1` for the current, and `#2` for the new value. The ⟨*definition*⟩ will not expand any further during the entire parsing for `aggregate`, whereas in `e-aggregate` everything that ends up in ⟨*definition*⟩ (so whatever you provide including the values in `#1` and `#2`) will be fully expanded (using the `\expanded` primitive), so use `\noexpand` and `\unexpanded` to protect what shouldn't be expanded. The resulting ⟨*key*⟩ will inherit being either `short` or `long` from the ⟨*primary*⟩ key.

Example: The following defines an internal key (`k-internal`), which is used to build a comma separated list from each call of the user facing key (`k`):

```
\ekvcSplit\foo
  {k-internal=0,color=red}
  {\textcolor{#2}{#1}}
\ekvcSecondaryKeys\foo
  {aggregate k = {k-internal}{#1,#2}}
\foo{}\par
\foo{k=1,k=2,k=3,k=4}
```

```
0
0,1,2,3,4
```

Example: But also more strange stuff could end there, like macros or using the same value multiple times:

```
\ekvcSecondaryKeys\foo
  {aggregate k = {k-internal}{\old{#1}\new{#2\old{#1}}}}
```

flag-bool `flag-bool` ⟨*key*⟩ `=` ⟨*cs*⟩ long

This is a secondary *type* that doesn't involve any of the primary or other secondary keys. This defines ⟨*key*⟩ to take a value, which should be either `true` or `false`, and set the flag called ⟨*cs*⟩ accordingly as a boolean. If ⟨*cs*⟩ isn't defined yet it will be initialised as a flag. Note that the flag will not be set to a specific state automatically so a flag set in one macro might affect every other macro in the current scope. Please also read subsection 2.5.

Example: Provide a key `bold` to turn the output of our macro bold if the associated flag is `true`.

```
\ekvcSplit\foo{a=a,b=b}
  {%
    \ekvcFlagIf\fooFlag
      {\textbf{a is #1 and b is #2}\par}
      {a is #1 and b is #2\par}%
  }
\ekvcSecondaryKeys\foo{flag-bool bold = \fooFlag}
\foo{}\foo{bold=true}\foo{}\foo{bold=false}\foo{}
```

```
a is a and b is b
a is a and b is b
a is a and b is b
a is a and b is b
a is a and b is b
```

flag-true `flag-true` ⟨*key*⟩ `=` ⟨*cs*⟩ long

flag-false

This is similar to `flag-bool`, but the ⟨*key*⟩ will be a `NoVal-`⟨*key*⟩ and if used will set the flag to either `true` or `false`. If ⟨*cs*⟩ isn't defined yet it will be initialised as a flag. Note that the flag will not be set to a specific state automatically. Please also read subsection 2.5.

flag-raise `flag-raise` ⟨*key*⟩ `=` ⟨*cs*⟩ long

This defines ⟨*key*⟩ to be a `NoVal-`⟨*key*⟩ that will raise the flag called ⟨*cs*⟩ on usage. If ⟨*cs*⟩ isn't defined yet it will be initialised as a flag. Note that the flag will not be set to a specific state automatically. Please also read subsection 2.5.

2.3 Changing the Initial Values

\ekvcChange `\ekvcChange⟨cs⟩{⟨key⟩=⟨value⟩, ...}`

This processes the ⟨*key*⟩=⟨*value*⟩ list for the macro ⟨*cs*⟩ to set new defaults for it (meaning the initial values used if you don't provide anything at use time, not those specified with the `default` *type*). ⟨*cs*⟩ should be defined with expkv-cs (but it doesn't matter if it's a split or hash variant). Inside the ⟨*key*⟩=⟨*value*⟩ list both primary and secondary keys can be used. If ⟨*cs*⟩ was defined `\long` earlier it will still be `\long`, every other TEX prefix will be stripped (but expkv-cs doesn't support them anywhere else so that should be fine). The resulting new defaults will be stored inside the ⟨*cs*⟩ locally (just as the original initial values were). If there was an unknown key forwarding added to ⟨*cs*⟩ (see subsection 2.4) any unknown key will be stored inside the list of unknown keys as well. `\ekvcChange` is not expandable!

Example: With `\ekvcChange` we can now do the following:

```
\ekvcSplit\foo{a=a,b=b}{a is #1.\par b is #2.\par}
\begingroup
  \ekvcChange\foo{b=B}
  \foo{}
  \ekvcSecondaryKeys\foo{meta c={a={#1},b={#1}}}
  \ekvcChange\foo{c=c}
  \foo{}
\endgroup
\foo{}
```

```
a is a.
b is B.
a is c.
b is c.
a is a.
b is b.
```

Example: As a result with this the typical setup macro could be implemented:

```
\ekvcHashAndUse\fooKV{keyA=a,keyB=b}
\def\fooA#1{\fooKV{#1}\fooAinternal}
\def\fooB#1{\fooKV{#1}\fooBinternal}
\protected\def\foosetup{\ekvcChange\fooKV}
```

Of course the usage is limited to a single macro `\fooKV`, hence this might not be as powerful as similar macros used with other ⟨*key*⟩=⟨*value*⟩ interfaces. But at least a few similar macros could be grouped using the same key parsing macro internally like `\fooA` and `\fooB` do in this example.

2.4 Handling Unknown Keys

If your macro should handle unknown keys without directly throwing an error you can use the special ... marker in the ⟨*primary keys*⟩ list. Since those keys will be processed once by expkv they will be forwarded normalised: The ⟨*key*⟩ and the ⟨*value*⟩ will be forwarded with one set of surrounding spaces and braces, so a ⟨*key*⟩=⟨*value*⟩ pair will result in ␣{⟨*key*⟩}␣=␣{⟨*val*⟩}␣ and a `NoVal`-⟨*key*⟩ is forwarded as ␣{⟨*key*⟩}␣ (this way most other ⟨*key*⟩=⟨*value*⟩ implementations should parse the correct input).

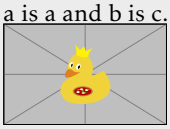
The exact behaviour differs slightly between the two variants (as all primary keys do). The behaviour inside the split variants will be similar to normal primary keys, the *n*-th argument (corresponding to the position of ... inside the primary keys list) will contain any unknown key encountered while parsing the argument. And inside the split

variant you can use a primary key named ... at the same time (since only the position in the list determines the argument, not the name).

Example: The following will forward any unknown key to `\includegraphics` to control the appearance while processing its own keys:

```
\newcommand*\foo{\ekvoptarg\fooKV{}}
\ekvcSplitAndForward\fooKV\fooOUT
  {
     a=a
    ,...
    ,b=b
    ,...={}
  }
\newcommand\fooOUT[5]
  {%
    a is #1 and b is #3.\par
    \includegraphics[{#2}]{#5}\par
    \texttt{...} is #4.\par
  }
\foo[width=.5\linewidth, b=c,
     ...={a stupid key name, but works}]
    {example-image-duck}
```

a is a and b is c.



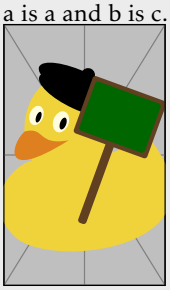
... is a stupid key name, but works.

Inside the hash variants the unknown keys list will be put inside the hash named ... (we have to use some name, and this one seems reasonable). As a consequence a primary key named ... would clash with the unknown key handler. If you still used such a key it would remove any unknown key stored there until that point and replace the list with its value.

Example: The following is more or less equivalent to the above example, but with the hash variant, and it will not contain the primary ... key. We have to make sure that `\includegraphics` sees the ⟨*key*⟩=⟨*value*⟩ list, so need to expand `\ekvcValue{...}{#1}` before `\includegraphics` parses it.

```
\newcommand*\foo{\ekvoptarg\fooKV{}}
\ekvcHashAndForward\fooKV\fooOUT
  {a=a, b=b, ...}
\newcommand\fooOUT[2]
  {%
    a is \ekvcValue{a}{#1} and
    b is \ekvcValue{b}{#1}.\par
    \ekvcValueSplit{...}{#1}{\includegraphics[]%
      {#2}\par
  }
\foo[width=\linewidth, b=c]
    {example-image-duck-portrait}
```

a is a and b is c.



2.5 Flags

The idea of flags is taken from expl3. They provide a way to store numerical information expandably, however only incrementing and accessing works expandably, decrementing is unexpandable. A flag has a height, which is a numerical value, and which can be raised

by 1. Flags come at a high computational cost (accessing them is slow and they require more memory than normal TEX data types like registers, both issues getting linearly worse with the height), so don't use them if not necessary.

The state of flags is always changed locally to the current group, but not to the current macro, so if you're using one of the *types* involving flags bear in mind that they can affect other macros using the same flags at the current scope!

expkvics provides some macros to access, alter, and use flags. Flags of **expkvics** don't share a name space with the flags of expl3.

`\ekvcFlagNew` `\ekvcFlagNew⟨flag⟩`

This initialises the macro ⟨*flag*⟩ as a new flag. It isn't checked whether the macro ⟨*flag*⟩ is currently undefined. A ⟨*flag*⟩ will expand to the flag's current height with a trailing space (so you can use it directly with `\ifnum` for example and it will terminate the number scanning on its own).

All other macros dealing with flags take as a parameter a macro defined as a ⟨*flag*⟩ with `\ekvcFlagNew`.

`\ekvcFlagHeight` `\ekvcFlagHeight⟨flag⟩`

This expands to the current height of ⟨*flag*⟩ in a single step of expansion (without a trailing space).

`\ekvcFlagRaise` `\ekvcFlagRaise⟨flag⟩`

This expandably raises the height of ⟨*flag*⟩ by 1.

`\ekvcFlagSetTrue` `\ekvcFlagSetTrue⟨flag⟩`
`\ekvcFlagSetFalse`

By interpreting an even value as false and an odd value as true we can use a flag as a boolean. This expandably sets ⟨*flag*⟩ to true or false, respectively, by raising it if necessary.

`\ekvcFlagIf` `\ekvcFlagIf⟨flag⟩{⟨true⟩}{⟨false⟩}`

This interprets a ⟨*flag*⟩ as a boolean and expands to either ⟨*true*⟩ or ⟨*false*⟩.

`\ekvcFlagIfRaised` `\ekvcFlagIfRaised⟨flag⟩{⟨true⟩}{⟨false⟩}`

This tests whether the ⟨*flag*⟩ is raised, meaning it has a height greater than zero, and if so expands to ⟨*true*⟩ else to ⟨*false*⟩.

`\ekvcFlagReset` `\ekvcFlagReset⟨flag⟩`
`\ekvcFlagResetGlobal`

This resets a flag (so restores its height to 0). This operation is *not* expandable and done locally for `\ekvcFlagReset` and globally for `\ekvcFlagResetGlobal`. If you really intend to use flags you can reset them every now and then to keep the performance hit low.

`\ekvcFlagGetHeight` `\ekvcFlagGetHeight⟨flag⟩{⟨next⟩}`

This retrieves the current height of the ⟨*flag*⟩ and provides it as a braced argument to ⟨*next*⟩, leaving ⟨*next*⟩{⟨*height*⟩} in the input stream.

\ekvcFlagGetHeights `\ekvcFlagGetHeights{⟨flag-list⟩}{⟨next⟩}`

This retrieves the current height of each ⟨*flag*⟩ in the ⟨*flag-list*⟩ and provides them as a single argument to ⟨*next*⟩. Inside that argument each height is enclosed in a set of braces individually. The ⟨*flag-list*⟩ is just a single argument containing the ⟨*flag*⟩s. So a usage like `\ekvcFlagGetHeights{\myflagA\myflagB}{\stuff}` will expand to `\stuff{{⟨height-A⟩}{⟨height-B⟩}}`.

2.6 Further Examples

Example: Using `\NewExpandableDocumentCommand` or expkv's `\ekvoptarg` or `\ekvoptargTF` and forwarding arguments one can easily define ⟨*key*⟩=⟨*value*⟩ macros with actual optional and mandatory arguments as well. A small nonsense example:

```
\makeatletter
\newcommand*\nonsense{\ekvoptarg\nonsense@a{}}
\ekvcHashAndForward\nonsense@a\nonsense@b
  {
    keyA = A,
    keyB = B,
    keyC = c,
    keyD = d,
  }
\newcommand*\nonsense@b[2]
  {%
    \begin{tabular}{lll|}
      key & A & \ekvcValue{keyA}{#1} \\
          & B & \ekvcValue{keyB}{#1} \\
          & C & \ekvcValue{keyC}{#1} \\
          & D & \ekvcValue{keyD}{#1} \\
      \multicolumn{2}{l}{mandatory} & #2 \\
    \end{tabular}%
  }
\makeatother
\nonsense{} % do nonsense
\nonsense[keyA=hihi]{haha}
\nonsense[keyA=hihi, keyB=A]{hehe}
\nonsense[keyC=huhu, keyA=hihi, keyB=A]{haha}
```

resulting in

key	A	A	key	A	hihi	key	A	hihi	key	A	hihi
	B	B		B	B		B	A		B	A
	C	c		C	c		C	c		C	huhu
	D	d		D	d		D	d		D	d
mandatory			mandatory		haha	mandatory		hehe	mandatory		haha

Example: In subsubsection 1.9.2 I presented an expandable macro to calculate the sine of some user input with a few keys, and there I hinted to expkv-cs, so here's the same function implemented with `\ekvcSplitAndForward`. There is a small difference here, we need to use an internal key to store whether degrees or radians will be used, but we don't

need to use an internal key to collect the values of our individual keys in the correct order.

```
\makeatletter
\newcommand\sine{\ekvoptarg\sine@kv{}}
\ekvcSplitAndForward\sine@kv\sine@do
  {
     f        = sin
    ,internal = d
    ,round    = 3
  }
\ekvcSecondaryKeys\sine@kv
  {
     nmeta degree = internal=d
    ,nmeta radian = internal={}
  }
\newcommand*\sine@do[4]{\fpeval{round(#1#2(#4),#3)}}
\makeatother
\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
```

2.7 Freedom for Keys!

If this had been the TEXbook this subsection would have had a double bend sign. Not because it is overly complicated, but because it shows things which could break expkv|cs's expandability and its alignment safety. This is for experienced users wanting to get the most flexibility and knowing what they are doing.

In case you're wondering, it is possible to define other keys than the primaries and the secondary key *types* listed in subsection 2.2 for a macro defined with expkv|cs by using the low-level interface of expkv or even the interface provided by expkv|def. The set name used for expkv|cs's keys is the macro name, including the leading backslash, or more precisely the result of `\string`⟨*cs*⟩ is used. This can be exploited to define additional keys with arbitrary code. Consider the following *bad* example:

```
\ekvcSplit\foo{a=A,b=B}{a is #1.\par b is #2.\par}
\protected\ekvdef{\string\foo}{c}{\def\fooC{#1}}
```

This would define a key named `c` that will store its ⟨*value*⟩ inside a macro. The issue with this is that this can't be done expandably. As a result, the macro `\foo` isn't always expandable any more (not that bad if this was never required; killjoy if it was) and as soon as the key `c` is used it is also no longer alignment safe[3] (might be bad depending on the usage).

So why do I show you this? Because we could as well do something useful like creating a key that pre-parses the input and after that passes the parsed value on. This parsing would have to be completely expandable though (and we could perhaps also implement this using the `e-aggregate` *type*). For the pass-on part we can use the following function:

[3]This means that the ⟨*key*⟩=⟨*value*⟩ list can't contain alignment markers that are not inside an additional set of braces if used inside a TEX alignment

\ekvcPass `\ekvcPass⟨cs⟩{⟨key⟩}{⟨value⟩}`

This passes ⟨*value*⟩ on to ⟨*key*⟩ for the expkvics-macro ⟨*cs*⟩. It should be used inside the key parsing of a macro defined with expkvics, else this most likely results in a low level TEX error. You can't forward anything to the special unknown key handler ... as that is no defined key.

Example: With this we could for example split the value of a key at a hyphen and pass the parts to different keys:

```
\ekvcSplit\foo{a=A,b=B}{a is #1.\par b is #2.\par}
\ekvdef{\string\foo}{c}{\fooSplit#1\par}
\def\fooSplit#1-#2\par
  {\ekvcPass\foo{a}{#1}\ekvcPass\foo{b}{#2}}
\foo{}
\foo{c=1-2}
```

a is A.
b is B.
a is 1.
b is 2.

Additionally, there is a more general version of the `aggregate` secondary key type, namely the `process` key type:

process `process ⟨key⟩ = {⟨primary⟩}{⟨definition⟩}` long

This will grab the current value of a ⟨*primary*⟩ key as `#1` (without changing the current value) and the new value as `#2` and leave all the processing to ⟨*definition*⟩. You should use `\ekvcPass` to forward the values afterwards. Unlike `aggregate` you can specify whether the ⟨*key*⟩ should be long or not, this isn't inherited from the ⟨*primary*⟩ key. Keep in mind that you could easily break things here if your code does not work by expansion.

Example: We could define a key that only accepts values greater than the current value with this:

```
\ekvcSplit\foo{internal=5}{a is #1.\par}
\ekvcSecondaryKeys\foo
  {
    process a={internal}
      {\ifnum#1<#2 \ekvcPass\foo{internal}{#2}\fi}
  }
\foo{a=1}
\foo{a=5}
\foo{a=9}
```

a is 5.
a is 5.
a is 9.

Example: The same is possible with an `e-aggregate` key as well though:

```
\ekvcSplit\foo{internal=5}{a is #1.\par}
\ekvcSecondaryKeys\foo
  {
    e-aggregate a={internal}
      {\ifnum#1<#2 \unexpanded{#2}\else\unexpanded{#1}\fi}
  }
```

2.8 Useless Macros

These macros are most likely of little to no interest to users.

`\ekvcDate`
`\ekvcVersion`

These two macros store the version and date of the package/generic code.

3 expkv|def

```
\input{expkv-def} % plain
\usepackage{expkv-def} % LaTeX
\usemodule[expkv-def] % ConTeXt
```

Since the trend for the last couple of years goes to defining keys for a ⟨*key*⟩=⟨*value*⟩ interface using a ⟨*key*⟩=⟨*value*⟩ interface, I thought that maybe providing such an interface for **expkv** will make it more attractive for actual use. But at the same time I didn't want to broaden **expkv**'s initial scope. So here is **expkv|def**, go define ⟨*key*⟩=⟨*value*⟩ interfaces with ⟨*key*⟩=⟨*value*⟩ interfaces.

Unlike many of the other established ⟨*key*⟩=⟨*value*⟩ interfaces to define keys, **expkv|def** works using prefixes instead of suffixes (*e.g.*, `.tl_set:N` of l3keys) or directory like handlers (*e.g.*, `/.store in` of pgfkeys). This was decided as a personal preference, more over in TEX parsing for the first spaces is way easier than parsing for the last one, so this should also turn out to be faster. **expkv|def**'s prefixes are sorted into two categories: *prefixes*, which are equivalent to TEX's prefixes like `\long` and of which a ⟨*key*⟩ can have multiple, and *types* defining the basic behaviour of the ⟨*key*⟩ and of which a ⟨*key*⟩ must have one. For a description of the available *prefixes* take a look at subsubsection 3.2.1, the *types* are described in subsubsection 3.2.2.

3.1 Macros

The number of user-facing macros is quite manageable:

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

In ⟨*set*⟩, define ⟨*key*⟩ to have definition ⟨*value*⟩. The general syntax for ⟨*key*⟩ should be

⟨*prefix*⟩ ⟨*name*⟩

where ⟨*prefix*⟩ is a space separated list of optional *prefixes* followed by one *type*. The syntax of ⟨*value*⟩ is dependent on the used *type*.

`\ekvdDate` `\ekvdVersion` These two macros store the version and date of the package.

3.2 Prefixes

As already said, prefixes are separated into two groups, *prefixes* and *types*. Not every *prefix* is allowed for all *types*.

3.2.1 *Prefixes*

`new` The following ⟨*key*⟩ must be new (so previously undefined). An error is thrown if it is already defined and the new definition is ignored. `new` only asserts that there are no conflicts between `NoVal-`⟨*key*⟩s and other `NoVal-`⟨*key*⟩s or `Val-`⟨*key*⟩s and other `Val-`⟨*key*⟩s.

Example: You can test the following (lines throwing an error are marked by a comment, error messages are printed in red for this example):

```
\ekvdefinekeys{new-example}
  {
     new code  key = \domystuffwitharg{#1}
    ,new noval KEY = \domystuffwithoutarg
    ,new bool  key = \mybool % Error!
    ,new bool  KEY = \mybool % Error!
    ,new meta  key = {KEY}   % Error!
    ,new nmeta KEY = {key}   % Error!
  }
```

```
! expkv-def Error: The key for 'new bool key' is already defined
! expkv-def Error: The key for 'new bool KEY' is already defined
! expkv-def Error: The key for 'new meta key' is already defined
! expkv-def Error: The key for 'new nmeta KEY' is already defined
```

`also` The following key *type* will be *added* to an existing ⟨*key*⟩'s definition. You can't add a *type* taking an argument at use time to an existing ⟨*key*⟩ which doesn't take an argument and vice versa. Also you'll get an error if you try to add an action which isn't allowed to be either `\long` or `\protected` to a ⟨*key*⟩ which already is `\long` or `\protected` (the opposite order would be suboptimal as well, but can't be really captured with the current code).

A ⟨*key*⟩ already defined as `\long` or `\protected` will stay that way, but you can add `\long` or `\protected` to a ⟨*key*⟩ which isn't by using `also`.

Example: Suppose you want to create a boolean ⟨*key*⟩, but additionally to setting a boolean value you want to execute some more code as well. For this you can use the following:

```
\ekvdefinekeys{also-example}
  {
     bool key      = \ifmybool
    ,also code key = \domystuff{#1}
  }
```

If you use `also` on a `choice`, `bool`, `invbool`, or `boolpair` ⟨*key*⟩ it is tried to determine if the key already is of one of those types. If this test is true the declared choices will be added to the possible choices but the key's definition will not be changed other than that. If that wouldn't have been done, the callbacks of the different choices could get called multiple times.

`protected` `protect` The following ⟨*key*⟩ will be defined `\protected`. Note that *types* which can't be defined expandable will always use `\protected`. This only affects the key at use time not the ⟨*key*⟩ definition.

`long` The following ⟨*key*⟩ will be defined `\long` (so can take an explicit `\par` token in its ⟨*value*⟩). Please note that this only changes the ⟨*key*⟩ at use time. `long` being present or not doesn't limit you to use `\par` inside of the ⟨*key*⟩'s definition (if the *type* allows this).

3.2.2 *Types*

Since the *prefixes* apply to some of the *types* automatically but sometimes one might be disallowed we need some way to highlight this behaviour. In the following an enforced *prefix* will be printed black (`protected`), allowed *prefixes* will be grey (`protected`), and disallowed *prefixes* will be red (`protected`). This will be put flush-right in the syntax showing line.

`code` `code` ⟨*key*⟩ `= {`⟨*definition*⟩`}` `new also protected long`

`ecode`

Define ⟨*key*⟩ to be a `Val-`⟨*key*⟩ expanding to ⟨*definition*⟩. You can use `#1` inside ⟨*definition*⟩ to access the ⟨*key*⟩'s ⟨*value*⟩. The `ecode` variant will fully expand ⟨*definition*⟩ inside an `\edef`.

Example: The following defines the key `foo`, that'll count the number of tokens passed to it (we'll borrow a function from expl3 for this). It'll accept explicit `\par` tokens. Also it'll flip the TEX-if `\iffoo` to true. The result of the counting will be stored in a count register. (Don't get confused, all the next examples are part of this `\ekvdefinekeys` call, so there is no closing brace here.)

```
\ExplSyntaxOn
  \cs_new_eq:NN \exampleCount \tl_count_tokens:n
\ExplSyntaxOff
\newcount\examplefoocount
\newif\iffoo
\ekvdefinekeys{example}
  {
     protected long code foo =
      \footrue
      \examplefoocount=\exampleCount{#1}\relax
```

`noval` `noval` ⟨*key*⟩ `= {`⟨*definition*⟩`}` `new also protected long`

`enoval`

The `noval` *type* defines ⟨*key*⟩ as a `NoVal-`⟨*key*⟩ expanding to ⟨*definition*⟩. `enoval` fully expands ⟨*definition*⟩ inside an `\edef`.

Example: The following defines the `NoVal-`⟨*key*⟩ `foo` to toggle the TEX-if `\iffoo` to false and set `\examplecount` to `0`. It'll be `\protected` and mustn't override any existing key.

```
,new protected noval foo = \foofalse\examplefoocount=0\relax
```

`default` `default` ⟨*key*⟩ `= {`⟨*definition*⟩`}` `new also protected long`

`odefault`
`fdefault`
`edefault`

This serves to place a default ⟨*value*⟩ for a `Val-`⟨*key*⟩. Afterwards if you use ⟨*key*⟩ as a `NoVal-`⟨*key*⟩ it will be the same as if ⟨*key*⟩ got passed ⟨*definition*⟩ as its ⟨*value*⟩. The `odefault` variant will expand the key-macro once, so will be slightly quicker, but not change if you redefine the `Val-`⟨*key*⟩ afterwards. The `fdefault` version will expand the key-code until a non-expandable token or a space is found, a space would be gobbled.[4] The `edefault` on the other hand fully expands the key-code with ⟨*definition*⟩ as its argument in `\expanded`. The *prefix* `new` means that there should be no `NoVal-`⟨*key*⟩ of that name yet.

[4] For those familiar with TEX-coding: This uses a `\romannumeral`-expansion

Example: We later decide that the above behaviour isn't what we need any more and instead redefine the `NoVal-⟨key⟩ foo` to pass some default value to the `Val-⟨key⟩ foo`.

```
,default foo = {Some creative default text}
```

`initial` `oinitial` `finitial` `einitial`

`initial ⟨key⟩ = {⟨value⟩}` `new also protected long`
`initial ⟨key⟩`

With `initial` you can set an initial ⟨*value*⟩ for an already defined ⟨*key*⟩. It'll just call the ⟨*key*⟩ and pass it ⟨*value*⟩. The `einitial` variant will expand ⟨*value*⟩ using `\expanded` prior to passing it to the ⟨*key*⟩ and the `oinitial` variant will expand the first token in ⟨*value*⟩ once. `finitial` will expand ⟨*value*⟩ until a non-expandable token or a space is found, a space would be gobbled.[5]

If you don't provide a ⟨*value*⟩ (and no equals sign) the `NoVal-⟨key⟩` of the same name is called once (or, if you specified a `default` for a `Val-⟨key⟩` that would be used).

Example: We want to get a defined initial behaviour for our `foo`. So we count o tokens.

```
,initial foo = {}
```

`bool` `gbool` `boolTF` `gboolTF`

`bool ⟨key⟩ = ⟨cs⟩` `new also protected long`

The ⟨*cs*⟩ should be a single control sequence, such as `\iffoo`. This will define ⟨*key*⟩ to be a boolean key, which only takes the values `true` or `false` and will throw an error for other values. If the ⟨*key*⟩ is used as a `NoVal-⟨key⟩` it'll have the same effect as if you use `true`. `bool` and `gbool` will behave like TEX-ifs, so either be `\iftrue` or `\iffalse`. The ⟨*cs*⟩ in the `boolTF` and `gboolTF` variants will take two arguments and if true the first will be used else the second, so they are always either `\@firstoftwo` or `\@secondoftwo`. The variants with a leading g will set the ⟨*cs*⟩ globally, the other locally. If ⟨*cs*⟩ is not yet defined it'll be initialised as the `false` version. Note that the initialisation is *not* done with `\newif`, so you will not be able to do `\footrue` outside of the ⟨*key*⟩=⟨*value*⟩ interface, but you could use `\newif` yourself. Even if the ⟨*key*⟩ will not be `\protected` the commands which execute the `true` or `false` choice will be, so the usage should be safe in an expansion context (*e.g.*, you can use `edefault ⟨key⟩ = false` without an issue to change the default behaviour to execute the `false` choice). Internally a `bool` is the same as a `choice` *type* which is set up to handle `true` and `false` as choices. `new` will assert that neither the `Val-⟨key⟩` nor the `NoVal-⟨key⟩` are already defined.

Example: Also we want to have a direct way to set our `\iffoo`, now that the `NoVal-⟨key⟩` doesn't toggle it any longer.

```
,bool dofoo = \iffoo
```

`invbool` `ginvbool` `invboolTF` `ginvboolTF`

`invbool ⟨key⟩ = ⟨cs⟩` `new also protected long`

These are inverse boolean keys, they behave like `bool` and friends but set the opposite meaning to the macro ⟨*cs*⟩ in each case. So if `key=true` is used `invbool` will set ⟨*cs*⟩ to `\iffalse` and vice versa.

Example: And since traditional interfaces lacked ⟨*key*⟩=⟨*value*⟩ support for packages, often a negated boolean key was used as well.

[5] Again using `\romannumeral`

```
,invbool nofoo = \iffoo
```

---

boolpair
gboolpair
boolpairTF
gboolpairTF

`boolpair` ⟨*key*⟩ `=` ⟨*cs*$_1$⟩⟨*cs*$_2$⟩ new also protected long

The `boolpair` *type* behaves like both `bool` and `invbool`, the ⟨*cs*$_1$⟩ will be set to the meaning according to the rules of `bool`, and ⟨*cs*$_2$⟩ will be set to the opposite.

---

store
estore
gstore
xstore

`store` ⟨*key*⟩ `=` ⟨*cs*⟩ new also protected long

The ⟨*cs*⟩ should be a single control sequence, such as `\foo`. This will define a `Val-`⟨*key*⟩ to store ⟨*value*⟩ inside of the control sequence. If ⟨*cs*⟩ isn't yet defined it will be initialised as empty. The variants behave similarly to their `\def`, `\edef`, `\gdef`, and `\xdef` counterparts, but will allow you to store macro parameters inside them without needing to double them. So `estore foo = \foo, initial foo = #1` will not result in a low level TEX error.

Example: Not only do we want to count the tokens handed to `foo`, but we want to also store them inside of a macro (and we don't need to specify `long` here, since `foo` is already `\long` from our `code` definition above).

```
,also store foo = \examplefoostore
```

---

data
edata
gdata
xdata

`data` ⟨*key*⟩ `=` ⟨*cs*⟩ new also protected long

The ⟨*cs*⟩ should be a single control sequence, such as `\foo`. This will define a `Val-`⟨*key*⟩ to store ⟨*value*⟩ inside of the control sequence. But unlike the `store` *type* the macro ⟨*cs*⟩ will be a switch at the same time, it'll take two arguments and if ⟨*key*⟩ was used expands to the first argument followed by ⟨*value*⟩ in braces, if ⟨*key*⟩ was not used ⟨*cs*⟩ will expand to the second argument (so behave like `\@secondoftwo`). The idea is that with this type you can define a key which should be typeset formatted. The `edata` and `xdata` variants will fully expand ⟨*value*⟩, the `gdata` and `xdata` variants will store ⟨*value*⟩ inside ⟨*cs*⟩ globally. Juts like with `store` you can use macro parameters without having to double them. The *prefixes* only affect the key-macro, ⟨*cs*⟩ will always be expandable and `\long`.

Example: Next we start to define other keys, now that our `foo` is pretty much exhausted. The following defines a key `bar` to be a `data` key.

```
,data bar = \examplebar
```

---

dataT
edataT
gdataT
xdataT

`dataT` ⟨*key*⟩ `=` ⟨*cs*⟩ new also protected long

Just like `data`, but instead of ⟨*cs*⟩ grabbing two arguments it'll only grab one, so by default it'll behave like `\@gobble`, and if ⟨*value*⟩ was given to ⟨*key*⟩ the ⟨*cs*⟩ will behave like `\@firstofone` appended by {⟨*value*⟩}.

Example: Another key we want to use is `baz`.

```
,dataT baz = \examplebaz
```

int
eint
gint
xint

`int` ⟨*key*⟩ `=` ⟨*cs*⟩ — new also protected long

The ⟨*cs*⟩ should be a single control sequence, such as `\foo`. An `int` key will be a `Val-`⟨*key*⟩ setting a TEX count register. If ⟨*cs*⟩ isn't defined yet, `\newcount` will be used to initialise it. The `eint` and `xint` variants will use `\numexpr` to allow basic computations in their ⟨*value*⟩. The `gint` and `xint` variants set the register globally.

dimen
edimen
gdimen
xdimen

`dimen` ⟨*key*⟩ `=` ⟨*cs*⟩ — new also protected long

The ⟨*cs*⟩ should be a single control sequence, such as `\foo`. This is just like `int` but uses a dimen register, `\newdimen`, and `\dimexpr` instead.

skip
eskip
gskip
xskip

`skip` ⟨*key*⟩ `=` ⟨*cs*⟩ — new also protected long

The ⟨*cs*⟩ should be a single control sequence, such as `\foo`. This is just like `int` but uses a skip register, `\newskip`, and `\glueexpr` instead.

Example: Exemplary for the different register keys, the following defines `distance` so that we can store some distance.

```
,eskip distance = \exampledistance
```

toks
gtoks
apptoks
gapptoks
pretoks
gpretoks

`toks` ⟨*key*⟩ `=` ⟨*cs*⟩ — new also protected long

The ⟨*cs*⟩ should be a single control sequence, such as `\foo`. Store ⟨*value*⟩ inside of a toks-register. The `g` variants use `\global`, the `app` variants append ⟨*value*⟩ to the contents of that register, the `pre` variants will prepend ⟨*value*⟩. If ⟨*cs*⟩ is not yet defined it will be initialised with `\newtoks`.

box
gbox

`box` ⟨*key*⟩ `=` ⟨*cs*⟩ — new also protected long

The ⟨*cs*⟩ should be a single control sequence, such as `\foo`. Typesets ⟨*value*⟩ into a `\hbox` and stores the result in a box register. The boxes are colour safe. expkv|def currently doesn't provide a `vbox` type.

meta

`meta` ⟨*key*⟩ `= {`⟨*key*⟩`=`⟨*value*⟩`, ...}` — new also protected long

This key *type* can set other keys, you can access the ⟨*value*⟩ given to the created `Val-`⟨*key*⟩ inside the ⟨*key*⟩=⟨*value*⟩ list using `#1`. This works by injecting the ⟨*key*⟩=⟨*value*⟩ list into the currently parsed list, so behaves just as if the ⟨*key*⟩=⟨*value*⟩ list was directly used instead of ⟨*key*⟩.

Example: And we want to set a full set of keys with just this single one called `all`.

```
,meta all =
  {distance=5pt,baz=cheese cake,bar=cocktail bar,foo={#1}}
```

nmeta

`nmeta` ⟨*key*⟩ `= {`⟨*key*⟩`=`⟨*value*⟩`, ...}` — new also protected long

This *type* sets other keys, but unlike `meta` this defines a `NoVal-`⟨*key*⟩, so the ⟨*key*⟩=⟨*value*⟩ list is static.

Example: and if `all` is set without a value we want to do something about it as well.

```
,nmeta all =
  {distance=10pt,baz=nothing,bar=Waikiki bar,foo}
```

smeta `smeta ⟨key⟩ = {⟨set⟩}{⟨key⟩=⟨value⟩, ...}` new also protected long

Yet another meta variant. smeta will define a Val-⟨key⟩, you can access the given ⟨value⟩ in the provided ⟨key⟩=⟨value⟩ list using #1. Unlike meta this will process that ⟨key⟩=⟨value⟩ list inside of ⟨set⟩ using a nested \ekvset call, so this is equal to \ekvset{⟨set⟩}{⟨key⟩=⟨value⟩, ...}. As a result you can't use \ekvsneak using keys or similar macros in the way you normally could.

snmeta `snmeta ⟨key⟩ = {⟨set⟩}{⟨key⟩=⟨value⟩, ...}` new also protected long

And the last meta variant. snmeta combines smeta and nmeta, so parses the ⟨key⟩=⟨value⟩ list inside of ⟨set⟩ and defines a NoVal-⟨key⟩ with a static list.

set `set ⟨key⟩ = {⟨set⟩}` new also protected long
`set ⟨key⟩`

This will define a NoVal-⟨key⟩ that will change the current set to ⟨set⟩. If you give no value to this definition (omit = {⟨set⟩}) the set name will be the same as ⟨key⟩ so set ⟨key⟩ is equivalent to set ⟨key⟩ = {⟨key⟩}. Note that just like in expkv it'll not be checked whether ⟨set⟩ is defined and you'll get a low-level TEX error if you use an undefined ⟨set⟩.

choice `choice ⟨key⟩ = {⟨value⟩=⟨definition⟩, ...}` new also protected long

choice defines a Val-⟨key⟩ that will only accept a limited set of values. You should define each possible ⟨value⟩ inside of the ⟨value⟩=⟨definition⟩ list. If a defined ⟨value⟩ is passed to ⟨key⟩ the ⟨definition⟩ will be left in the input stream. You can make individual values protected inside the ⟨value⟩=⟨definition⟩ list by using that *prefix*. To also allow choices that shouldn't be \protected but which start with the word protected you can also use unprotected as a special *prefix*. By default a choice key and all its choices are expandable, an undefined ⟨value⟩ will throw an error in an expandable way. You can add additional choices after the ⟨key⟩ was created by using choice again for the same ⟨key⟩, redefining choices is possible the same way, but there is no interface to remove certain choices. To change the behaviour of unknown choices see also the unknown-choice *type*.

Example: We give the users a few choices.

```
,choice choose =
  {
     protected lemonade = \def\exampledrink{something sour}
    ,protected water = \def\exampledrink{something boring}
  }
```

choice-store `choice-store ⟨key⟩ = ⟨cs⟩{⟨value⟩=⟨definition⟩, ...}` new also protected long

The ⟨*cs*⟩ should be a single control sequence, such as \foo. This is a special *type* of the choice *type* that'll store the given choice inside the macro ⟨*cs*⟩. Since storing inside a macro can't be done expandably every choice-code is \protected, and you might define the choice-store key itself as \protected as well if you want. Inside the ⟨*value*⟩=⟨*definition*⟩ list the =⟨*definition*⟩ part is optional, if you omit it the ⟨*value*⟩ will be stored as given during define-time inside of ⟨*cs*⟩ (during use-time the ⟨*value*⟩ needs to be matched \detokenized), and if you specify =⟨*definition*⟩ that ⟨*definition*⟩ will be stored inside of ⟨*cs*⟩ instead. If ⟨*cs*⟩ doesn't yet exist it's initialised as empty.

Example: The following keys key1 and key2 are equivalent at use time (this doesn't continue the \ekvdefinekeys-call for the set example above):

```
\newcommand*\mya{}% initialise \mya
\ekvdefinekeys{choice-store-example}
  {
     choice key1 =
      {
         protected a = \def\mya{a}
        ,protected b = \def\mya{b}
        ,protected c = \def\mya{c}
        ,protected d = \def\mya{FOO}
      }
    ,choice-store key2 = \myb{a,b,c,d=FOO}
  }
```

Example: (this continues the \ekvdefinekeys-call for the set example from above) After the above drinks we define a few more choices which are directly stored.

```
    ,choice-store choose = \exampledrink{beer,wine}
```

One might notice that the entire setup of the choose key could've been done using only choice-store.

choice-enum `choice-enum ⟨key⟩ = ⟨cs⟩{⟨value⟩, ...}` new also protected long

The ⟨*cs*⟩ should be a single control sequence, such as \foo. This is similar to choice-store, the differences are: ⟨*cs*⟩ should be a count register or is initialised as such using \newcount; instead of the ⟨*value*⟩ itself being stored its position in the list of choices is stored (zero-based). It is not possible to specify a ⟨*definition*⟩ to store something else than the numerical position inside the list.

Example: The following keys key1 and key2 are equivalent at use time (another example not using the example set of above's \ekvdefinekeys):

```
\newcount\myc
\ekvdefinekeys{choice-enum-example}
  {
     choice key1 =
      {
         protected a={\myc=0 }
        ,protected b={\myc=1 }
        ,protected c={\myc=2 }
```

```
    }
  ,choice-enum key2 = \myd{a,b,c}
}
```

unknown-choice `unknown-choice ⟨key⟩ = {⟨definition⟩}` new also protected long

By default an unknown ⟨value⟩ passed to a `choice` or `bool` *type* (and all their variants) will throw an error. However, with this prefix you can define an alternative action which should be executed if ⟨key⟩ received an unknown choice. In ⟨definition⟩ you can refer to the given invalid choice with `#1`.

Example: If a drink was chosen with `choose` that's not defined we don't want to throw an error, but store something else instead.

```
  ,protected unknown-choice choose =
    \def\exampledrink{something unavailable}
}% closing brace for \ekvdefinekeys
```

unknown␣code `unknown code = {⟨definition⟩}` new also protected long

By default expkv throws errors when it encounters unknown keys in a set. With the `unknown` *type* you can define handlers that deal with undefined keys, instead of a ⟨key⟩ name you have to specify a subtype for this, here the subtype is `code`.

With `unknown code` the ⟨definition⟩ is used for unknown keys which were provided a value (so corresponds to `\ekvdefunknown`), you can access the unknown ⟨key⟩ name with `#1` (`\detokenized`), the given ⟨value⟩ with `#2`, and the unprocessed ⟨key⟩ name with `#3` (in case you want to further expand it).[6]

unknown␣noval `unknown noval = {⟨definition⟩}` new also protected long

This is like `unknown code` but uses ⟨definition⟩ for unknown keys to which no value was passed (so corresponds to `\ekvdefunknownNoVal`). You can access the `\detokenized` ⟨key⟩ name with `#1` and the unprocessed one with `#2`.

unknown␣redirect-code `unknown redirect-code = {⟨set-list⟩}` new also protected long

This uses a predefined action for `unknown code`. Instead of throwing an error, it is tried to find the ⟨key⟩ in each ⟨set⟩ in the comma separated ⟨set-list⟩. The first found match will be used and the remaining options from the list discarded. If the ⟨key⟩ isn't found in any ⟨set⟩ an expandable error will be thrown eventually. Internally expkv's `\ekvredirectunknown` will be used.

unknown␣redirect-noval `unknown redirect-noval = {⟨set-list⟩}` new also protected long

This behaves just like `unknown redirect-code` but will set up means to forward keys for `unknown noval`. Internally expkv's `\ekvredirectunknownNoVal` will be used.

unknown␣redirect `unknown redirect = {⟨set-list⟩}` new also protected long

This is a short cut to apply both, `unknown redirect-code` and `unknown redirect-noval`, as a result you might get doubled error messages, one from each.

[6] There is some trickery involved to get this more intuitive argument order without any performance hit if you compare this to `\ekvdefunknown` directly

Time to use all those keys defined in the different examples!

```
\newcommand\defexample[1][]
  {%
    \ekvset{example}{#1}%
    After walking \the\exampledistance\space we finally reached
    \examplebar{\emph}{no particular place}.
    There I ordered
    \iffoo
      a drink called \examplefoostore\space (that has
      \the\examplefoocount\space tokens in it)%
    \else
      nothing of particular interest%
    \fi
    \examplebaz{ and ate \emph}.
    Then a friend of mine also chose \exampledrink.
    \par
  }
\defexample[nofoo]
\defexample[all,choose=lemonade]
\defexample
  [all=wheat beer,bar=Biergarten,baz=pretzel,choose=champagne]
```

Which results in three paragraphs of text:

> After walking 0.opt we finally reached no particular place. There I ordered nothing of particular interest. Then a friend of mine also chose .
> After walking 10.opt we finally reached *Waikiki bar*. There I ordered a drink called Some creative default text (that has 26 tokens in it) and ate *nothing*. Then a friend of mine also chose something sour.
> After walking 5.opt we finally reached *Biergarten*. There I ordered a drink called wheat beer (that has 10 tokens in it) and ate *pretzel*. Then a friend of mine also chose something unavailable.

3.3 Another Example

This picks up the standard use case from subsubsection 1.9.1, but defines the keys using `\ekvdefinekeys`.

```
\makeatletter
\ekvdefinekeys{myrule}
  {
     store   ht    = \myrule@ht
    ,initial ht    = 1ex
    ,store   wd    = \myrule@wd
    ,initial wd    = 0.1em
    ,store   raise = \myrule@raise
    ,initial raise = \z@
    ,meta    lower = {raise={-#1}}
  }
\ekvsetdef\myruleset{myrule}
\newcommand*\myrule[1][]
  {%
    \begingroup
      \myruleset{#1}%
      \rule[\myrule@raise]{\myrule@wd}{\myrule@ht}%
    \endgroup
  }
\makeatother
a\myrule\par
a\myrule[ht=2ex,lower=.5ex]\par
\myruleset{wd=5pt}
a\myrule
```

4 expkv|opt

```
\usepackage{expkv-opt} % LaTeX
```

expkv|opt allows to parse LaTeX 2ε class and package options as ⟨*key*⟩=⟨*value*⟩ lists using sets of expkv.

With the 2021-05-01 release of LaTeX 2ε there were some very interesting changes to the package and class options code. It is now possible to use braces inside the options, and we can access options without them being preprocessed. As a result, some but not all restrictions were lifted from the possible option usage. What will still fail is things that aren't save from an `\edef` expansion (luckily, the exp:notation can be used to get around that as well). One feature of expkv|opt that doesn't work any more is the possibility to parse the unused option list, because that one doesn't contain the full information any more. expkv|opt will fall back to v0.1 if the kernel is older than 2021-05-01.

Another very interesting change in LaTeX 2ε was the addition of ltkeys and its `\ProcessKeyOptions` with the possibility to parse future options with it instead of getting the dreaded `Option clash` error. The idea is brilliant and changes made in the 2022-10-22 version allow us to provide the same feature without having to hack any kernel internals, so starting with kernel version 2022-11-01 expkv|opt supports this as well.

expkv|opt shouldn't place any restrictions on the keys, historic shortcomings of the kernel cannot be helped though, so the supported things vary with the kernel version (see above). The one thing that expkv|opt doesn't support, which expkv alone would, is active commas or equals signs. But there is no good reason why any of the two should be active in the preamble.

You can use LaTeX 2ε's rollback support, so to load v0.1 explicitly use:

```
\usepackage{expkv-opt}[=v0.1]
```

which will load the last version of expkv|opt that doesn't use the raw option lists (this shouldn't be done by a package author, but only by a user on a single-document basis if there are some incompatibilities, which is unlikely).

4.1 Macros

4.1.1 Option Processors

expkv|opt's behaviour if it encounters a defined or an undefined ⟨*key*⟩ depends on which list is being parsed and whether the current file is a class or not. Of course in every case a defined ⟨*key*⟩'s callback will be invoked but an additional action might be executed. For this reason the rule set of every macro will be given below the short description which list it will parse.

During each of the processing macros the current list element (not processed in any way) is stored within the macro `\CurrentOption`.

`\ekvoProcessOptions` `\ekvoProcessOptions{`⟨*set*⟩`}`

This runs `\ekvoProcessGlobalOptions`, then `\ekvoProcessLocalOptions`, and finally `\ekvoProcessFutureOptions`. If you're using `\ekvoUseUnknownHandlers` it'll affect all three option processors. Else the respective default unknown-rules are used.

\ekvoProcessLocalOptions `\ekvoProcessLocalOptions{⟨set⟩}`

This parses the options which are directly passed to the current class or package for an **expkv** ⟨*set*⟩.

Class: defined remove the option from the list of unused global options if the local option list matches the option list of the main class and the unused global options list is not empty; else *nothing*

undefined add the key to the list of unused global options (if the local option list matches the option list of the main class)

Package: defined *nothing*

undefined throw an error

\ekvoProcessGlobalOptions `\ekvoProcessGlobalOptions{⟨set⟩}`

In LaTeX 2ε the options given to `\documentclass` are global options. This macro processes the global options for an **expkv** ⟨*set*⟩.

Class: defined remove the option from the list of unused global options

undefined *nothing*

Package: defined remove the option from the list of unused global options

undefined *nothing*

\ekvoProcessFutureOptions `\ekvoProcessFutureOptions{⟨set⟩}`

This parses the option list of every future call of the package with `\usepackage` or similar with an **expkv** ⟨*set*⟩, circumventing the `Option clash` error that'd be thrown by LaTeX 2ε. It is only available for kernel versions starting with `2022-11-01`. It is mutually exclusive with LaTeX 2ε's `\ProcessKeyOptions` (which ever comes last defines how future options are parsed).

Class: defined *nothing*

undefined throw an error

Package: defined *nothing*

undefined throw an error

\ekvoProcessOptionsList `\ekvoProcessOptionsList⟨list⟩{⟨set⟩}`

Process the ⟨*key*⟩=⟨*value*⟩ list stored in the macro ⟨*list*⟩.

Class: defined *nothing*

undefined *nothing*

Package: defined *nothing*

undefined *nothing*

4.1.2 Other Macros

\ekvoUseUnknownHandlers `\ekvoUseUnknownHandlers`⟨cs_1⟩⟨cs_2⟩ *or*
`\ekvoUseUnknownHandlers*`

With this macro you can change the action expkv-opt executes if it encounters an undefined ⟨*key*⟩ for the next (and only the next) list processing macro. The macro ⟨cs_1⟩ will be called if an undefined `NoVal-`⟨*key*⟩ is encountered and get one argument being the ⟨*key*⟩ (without being `\detokenized`). Analogous the macro ⟨cs_2⟩ will be called if an undefined `Val-`⟨*key*⟩ was parsed and get two arguments, the first being the ⟨*key*⟩ (without being `\detokenized`) and the second the ⟨*value*⟩.

If you use the starred variant, it'll not take further arguments. In this case the undefined handlers defined via `\ekvdefunknown` and `\ekvdefunknownNoVal` in the parsing set get used, and if those aren't available they'll simply do nothing.

\ekvoVersion
\ekvoDate

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

4.2 Examples

Example: Let's say we want to create a package that changes the way footnotes are displayed in LaTeX. For this it will essentially just redefine `\thefootnote` and we'll call this package ex-footnote. First we report back which package we are:

```
\ProvidesPackage{ex-footnote}[2020-02-02 v1 change footnotes]
```

Next we'll need to provide the options we want the package to have.

```
\RequirePackage{color}
\RequirePackage{expkv-opt}% also loads expkv
\ekvdef{ex-footnote}{color}{\def\exfn@color{#1}}
\ekvdef{ex-footnote}{format}{\def\exfn@format{#1}}
```

We can provide initial values just by defining the two macros storing the value.

```
\newcommand*\exfn@color{}
\newcommand*\exfn@format{arabic}
```

Next we need to process the options given to the package. The package should only obey options directly passed to it, so we're using `\ekvoProcessLocalOptions` and `\ekvoProcessFutureOptions`:

```
\ekvoProcessLocalOptions {ex-footnote}
\ekvoProcessFutureOptions{ex-footnote}
```

Now everything that's still missing is actually changing the way footnotes appear:

```
\renewcommand*\thefootnote
  {%
    \ifx\exfn@color\@empty
      \csname\exfn@format\endcsname{footnote}%
    \else
      \textcolor{\exfn@color}{\csname\exfn@format\endcsname{footnote}}%
    \fi
  }
```

So the complete code of the package would look like this:

```
\ProvidesPackage{ex-footnote}[2020-02-02 v1 change footnotes]

\RequirePackage{color}
\RequirePackage{expkv-opt}% also loads expkv

\ekvdef{ex-footnote}{color}{\def\exfn@color{#1}}
\ekvdef{ex-footnote}{format}{\def\exfn@format{#1}}
\newcommand*\exfn@color{}
\newcommand*\exfn@format{arabic}

\ekvoProcessLocalOptions {ex-footnote}
\ekvoProcessFutureOptions{ex-footnote}

\renewcommand*\thefootnote
  {%
    \ifx\exfn@color\@empty
      \csname\exfn@format\endcsname{footnote}%
    \else
      \textcolor{\exfn@color}{\csname\exfn@format\endcsname{footnote}}%
    \fi
  }
```

And it could be used with one (or thanks to `\ekvoProcessFutureOptions` all) of the following lines:

```
\usepackage{ex-footnote}
\usepackage[format=fnsymbol]{ex-footnote}
\usepackage[color=green]{ex-footnote}
\usepackage[color=red,format=roman]{ex-footnote}
```

Example: This document was compiled with the global options `[exfoo=value, exbar, exfoo=\empty ]` in use. If we define the following keys

```
\ekvdef{optexample}{exfoo}
  {Global option \texttt{exfoo} got \texttt{\detokenize{#1}}.\par}
\ekvdefNoVal{optexample}{exbar}
  {Global option \texttt{exbar} set.\par}
```

we can use those options to control the result of the following:

```
\ekvoProcessGlobalOptions{optexample}
```

Global option exfoo got value.
Global option exbar set.
Global option exfoo got \empty .

Please note that under normal conditions `\ekvoProcessGlobalOptions` is only useable in the preamble; this example is only for academic purposes, you'll not be able to reproduce this with the exact code shown above.

5 expkv|pop

```
\input{expkv-pop} % plain
\usepackage{expkv-pop} % LaTeX
\usemodule[expkv-pop] % ConTeXt
```

The expkv|pop is mainly written to lay the basis for expkv|cs's and expkv|def's key-defining front ends. Historically the two packages shared pretty similar code. To unify this and reduce the overall code amount some auxiliary package was originally planned, but then I realised that with little to no overhead (apart from documentation) this can also be provided to users. Well, and then I thought, why not make the whole thing expandable as well. And here we are.

So what's the idea? This package provides a **p**refix **o**riented **p**arser[7] with two kinds of prefixes. The first is called a *prefix*, of which an item can have arbitrary many, the second a *type*, of which every item has only one. To distinguish the concept of an optional *prefix* from the generic term "prefix" I'll use this formatting whenever the special kind of prefix is meant.

Another peculiarity of expkv|pop compared to the other packages in expkv|bundle is that it doesn't separate `NoVal-`⟨*key*⟩s from `Val-`⟨*key*⟩s as strictly. Instead a `NoVal`-marker is used as the value. If this is not what you want you can use `\ekvpValueAlwaysRequired` (see there).

5.1 Parsing Rules

A parser is processing only the ⟨*key*⟩ of a ⟨*key*⟩=⟨*value*⟩ pair. The ⟨*key*⟩ is split at spaces (braces might be lost by this process!). Each split off piece is checked whether it's a defined prefix. If it's a *type* parsing of the ⟨*key*⟩ stops and the remainder is considered a ⟨*name*⟩. If it's a *prefix* it'll be recorded and parsing goes on. If it's neither parsing is also stopped (and the last parsed space delimited part is put back – braces might've been lost at this step). If a no-type rule has been defined (`\ekvpDefNoType`) that one is executed else an error is thrown.

The *prefix* or *type* has to match after being `\detokenized`, whereas the ⟨*name*⟩ will be unchanged (except for stripping off the prefixes). If only a ⟨*key*⟩ is given (so no =⟨*value*⟩ used) the same is done, and instead of ⟨*value*⟩ a no-value marker is used (if that accidentally ends up in the input stream this looks like this: -NoValue-; this is the same as the marker used by expl3 for those familiar with it).

A *prefix* has two parts to it, a ⟨*pre*⟩ and a ⟨*post*⟩ code, whereas a *type* only results in a *type*-action (or the no-type action if that's defined and no *type* found). The parsing result can also be seen in Figure 1.

Please note that expkv|pop's parsers are fully expandable as long as your *prefixes* and *types* are. Additionally expkv|pop doesn't provide means to define parsers, *prefixes*, or *types* `\protected`. As a result, make sure you'll always call `\ekvpParse` inside of a `\protected` macro if you need anything that's unexpandable or else your code might not do what you intended since some states may not be updated when expandable code tries

[7] Naming packages is hard, especially when the name should fit a particular naming scheme. Big thanks to samcarter for helping me: https://topanswers.xyz/tex?q=1985. The author apologises that there is no `expkv-pnk`, `expkv-rok`, `expkv-jaz` or any other music themed name in expkv|bundle.

$$\mathtt{pre}_1\ \mathtt{pre}_2\ \ldots\ \mathtt{pre}_n\ \textit{type}\text{-}\mathtt{action}\ \mathtt{post}_n\ \ldots\ \mathtt{post}_2\ \mathtt{post}_1$$

Figure 1: Structure of a single ⟨*key*⟩=⟨*value*⟩ pair's parsing result with *n prefix*es

to access them. The macro \ekvpProtect can help to overcome this issue, but it's more of a last resort than a clean solution.

5.2 Defining Parsers

\ekvpNewParser `\ekvpNewParser{⟨parser⟩}`

Defines a new parser called ⟨parser⟩. Every parser has to be defined this way. Throws an error if the parser is already defined.

\ekvpDefType `\ekvpDefType{⟨parser⟩}{⟨type⟩}{⟨code⟩}`

Defines a *type* that is called ⟨type⟩ for the parser ⟨parser⟩. If the *type* is parsed the ⟨code⟩ will be used as a *type*-action. Inside of ⟨code⟩ you can use #1 to refer to the ⟨name⟩ (the remainder of the ⟨key⟩ after stripping off all the prefixes), #2 to use the unaltered ⟨key⟩, and #3 to access the ⟨value⟩ which was given to your ⟨key⟩.

\ekvpDefPrefix `\ekvpDefPrefix{⟨parser⟩}{⟨prefix⟩}{⟨pre⟩}{⟨post⟩}`

Defines a *prefix* that is called ⟨prefix⟩ for the parser ⟨parser⟩. If the *prefix* is encountered the code in ⟨pre⟩ will be put before the *type*-action and the code in ⟨post⟩ will be put behind it. If multiple *prefix*es are used the ⟨pre⟩ of the first will be put first and the ⟨post⟩ of the first will be put last. Inside of ⟨pre⟩ and ⟨post⟩ #1 is replaced with the found *type*, #2 the ⟨name⟩, and #3 the unaltered ⟨key⟩. If no valid type was found and the no-type rule defined with \ekvpDefNoType is executed the argument #1 will be empty.

\ekvpDefAutoPrefix `\ekvpDefAutoPrefix{⟨parser⟩}{⟨pre⟩}{⟨post⟩}`

You can also define a *prefix*-like rule that is executed on each element automatically. So the ⟨pre⟩ and ⟨post⟩ code of this will be inserted for every valid element of the ⟨key⟩=⟨value⟩ list. Just like for \ekvpDefPrefix you can access the *type* with #1, the ⟨name⟩ with #2, and the unaltered ⟨key⟩ with #3.

\ekvpDefPrefixStore `\ekvpDefPrefixStore{⟨parser⟩}{⟨prefix⟩}⟨cs⟩{⟨pre⟩}{⟨post⟩}`

This is a shortcut to define a *prefix* named ⟨prefix⟩ for ⟨parser⟩ that'll store ⟨pre⟩ inside of ⟨cs⟩ (which should be a single control sequence) before the *type*-action and afterwards store ⟨post⟩ in it. Both definitions (in ⟨pre⟩ and in ⟨post⟩) are put inside \ekvpProtect.

\ekvpDefPrefixLet `\ekvpDefPrefixLet{⟨parser⟩}{⟨prefix⟩}⟨cs⟩⟨pre⟩⟨post⟩`

This is similar to \ekvpDefPrefixStore, but instead of storing in the ⟨cs⟩ it'll be let to the single tokens specified by ⟨pre⟩ and ⟨post⟩. If either ⟨pre⟩ or ⟨post⟩ contains more than a single token the remainder is put after the \let statement. Both assignments (in ⟨pre⟩ and in ⟨post⟩) are put inside \ekvpProtect.

\ekvpLet `\ekvpLet{⟨parser$_1$⟩}{⟨type⟩}{⟨name$_1$⟩}[⟨parser$_2$⟩]{⟨name$_2$⟩}`

Copies the definition of a *prefix* or *type*. The ⟨type⟩ should be one of prefix, or type. The *prefix* or *type* ⟨name$_1$⟩ for ⟨parser$_1$⟩ will be let equal to the *prefix* or *type* ⟨name$_2$⟩ of ⟨parser$_2$⟩. If you omit the optional ⟨parser$_2$⟩ it will default to ⟨parser$_1$⟩.

5.3 Changing Default Behaviours

\ekvpValueAlwaysRequired `\ekvpValueAlwaysRequired{⟨parser⟩}`

By default a special no-value marker will be provided for ⟨*value*⟩ if no value was given to a key. If this is used instead an error will be thrown that a value is required.

\ekvpDefNoValue `\ekvpDefNoValue{⟨parser⟩}{⟨code⟩}`

This is a third alternative to the default behaviour and `\ekvpValueAlwaysRequired`. With this macro you can stop normal parsing if no value was specified and instead run ⟨*code*⟩. Inside of ⟨*code*⟩ the unprocessed `NoVal-`⟨*key*⟩ is available as `#1`. No further processing of this ⟨*key*⟩=⟨*value*⟩ list element takes place.

\ekvpUseNoValueMarker `\ekvpUseNoValueMarker{⟨parser⟩}{⟨marker⟩}`

This macro changes the no-value marker from the package default to ⟨*marker*⟩. Note that macros like `\ekvpAssertValue` don't work with markers different from the default one.

\ekvpDefNoValuePrefix `\ekvpDefNoValuePrefix{⟨parser⟩}{⟨pre⟩}{⟨post⟩}`

It is also possible to handle `NoVal-`⟨*key*⟩s as if this was some special *prefix*. So if a `NoVal-`⟨*key*⟩ is encountered you'll have ⟨*pre*⟩ and ⟨*post*⟩ put before and behind the *type*-action (as the outermost *prefix*). The no-value marker will be forwarded as ⟨*value*⟩. If you want to change a parser's marker and use this you have to use `\ekvpUseNoValueMarker` before calling `\ekvpDefNoValuePrefix`, and you must not use `\ekvpDefNoValue` or `\ekvpValueAlwaysRequired` before using `\ekvpDefNoValuePrefix` (both result in undefined behaviour).

\ekvpDefNoType `\ekvpDefNoType{⟨parser⟩}{⟨code⟩}`

This defines an action if no valid *type* was found, otherwise this behaves like `\ekvpDefType` with the same arguments `#1` (⟨*name*⟩), `#2` (unaltered ⟨*key*⟩), and `#3` (⟨*value*⟩) in ⟨*code*⟩. If this isn't used for the ⟨*parser*⟩ instead an error will be thrown whenever no *type* is found.

5.4 Markers

expkv|pop will place three markers for each list element that was parsed and defines an auxiliary to gobble up to that marker. After each marker an additional group is placed containing the current list element (excluding the ⟨*value*⟩). The gobblers gobble that group as well. Those markers are:

\ekvpEOP
\ekvpGobbleP

Is placed after all the *prefix*es' ⟨*pre*⟩ code, directly before the *type*-action.

\ekvpEOT
\ekvpGobbleT

Is placed after the *type*-action, directly before the last *prefix*'s ⟨*post*⟩ code.

\ekvpEOA
\ekvpGobbleA

Is placed at the end of the complete result of the current element, so after all the *prefix*es' ⟨*post*⟩ code.

5.5 Helpers in Actions

\ekvpIfNoVal

\ekvpIfNoVal{⟨*arg*⟩}{⟨*true*⟩}{⟨*false*⟩}

This will expand to ⟨*true*⟩ if the ⟨*arg*⟩ is the special no-value marker, otherwise ⟨*false*⟩ is left in the input stream.

\ekvpAssertIf
\ekvpAssertIfNot

\ekvpAssertIf[⟨*marker*⟩]{⟨*if*⟩}{⟨*message*⟩}

This macro will run the TEX-if test specified by ⟨*if*⟩ (should expand to any TEX-style if, e.g., \iftrue or \ifx⟨*A*⟩⟨*B*⟩). If the test is true everything's fine, else an error message is thrown using ⟨*message*⟩ followed by the current element and everything up to ⟨*marker*⟩ is gobbled (⟨*marker*⟩ can be any of EOT, which is the default, EOP, or EOA). The Not variant will invert the logic, so if the TEX-style if is true this will throw the error.

\ekvpAssertTF
\ekvpAssertTFNot

\ekvpAssertTF[⟨*marker*⟩]{⟨*if*⟩}{⟨*message*⟩}

This is pretty similar to \ekvpAssertIf, but ⟨*if*⟩ should be a test that uses its first argument if it's true and its second otherwise (so an error is thrown if the second argument is used, nothing happens otherwise). The Not variant is again inversed.

\ekvpAssertValue
\ekvpAssertNoValue

\ekvpAssertValue[⟨*marker*⟩]{⟨*arg*⟩}

Asserts that ⟨*arg*⟩ is not the no-value marker (\ekvpAssertValue) or is the no-value marker (\ekvpAssertNoValue), otherwise throws an error and gobbles everything up to ⟨*marker*⟩ (like \ekvpAssertIf).

\ekvpAssertOneValue
\ekvpAssertTwoValues

\ekvpAssertOneValue[⟨*marker*⟩]{⟨*arg*⟩}

Asserts that ⟨*arg*⟩ contains exactly one or two values (which could both be either single tokens or braced groups – spaces between the two values in the \ekvpAssertTwoValues case are ignored), if the number of values doesn't match an error is thrown and everything up to ⟨*marker*⟩ gobbled.

\ekvpProtect

\ekvpProtect{⟨*code*⟩}

This macro protects ⟨*code*⟩ from further expanding in every context a \protected macro wouldn't expand. It needs at least two steps of expansion. When a \protected macro would expand this will remove the braces around ⟨*code*⟩ and TEX will process ⟨*code*⟩ the same way it normally would. After the first step of expansion it'll leave two macros, and after each further full expansion these two macros stay there. Since expkv|pop offers no method to define *prefixes* or *types* \protected you can instead use this macro. But if your parser needs any assignments you should nest the \ekvpParse call in a \protected macro anyway.

5.6 Using Parsers

\ekvpParse `\ekvpParse{⟨parser⟩}{⟨key⟩=⟨value⟩, ...}`

Parses the ⟨*key*⟩=⟨*value*⟩ list as defined for ⟨*parser*⟩. This expands in exactly two steps, and returns inside of \unexpanded, so doesn't expand any further in an \edef or \expanded. After the two steps it'll directly leave the code as though every *prefix*'s ⟨*pre*⟩ and ⟨*post*⟩ code and the *type*-action were input directly along with the different markers.

5.7 The Boring Macros

Just for the sake of completeness.

\ekvpDate
\ekvpVersion

Store the date and version, respectively.

5.8 Examples

Example: Let's define a parser that is similar to expkv-def's \ekvdefinekeys. First we define a new parser named exdef:

```
\ekvpNewParser{exdef}
```

We'll need to provide our prefixes, long and protected. The following initialises them and defines their action.

```
\newcommand*\exLong{}
\newcommand*\exProtected{}
\ekvpDefPrefixLet{exdef}{long}\exLong\long\empty
\ekvpDefPrefixLet{exdef}{protected}\exProtected\protected\empty
```

Now we define a few types, I'll go with noval, store, and code for starters. We exploit the fact that \ekvdef and \ekvdefNoVal will expand the first argument, so we can simply store the set name in a macro.

```
\ekvpDefType{exdef}{code}
  {%
    \ekvpAssertValue{#3}%
    \exProtected\exLong\ekvdef\exSetName{#1}{#3}%
  }
\ekvpDefType{exdef}{noval}
  {%
    \ekvpAssertValue{#3}%
    \ekvpAssertIfNot{\ifx\exLong\long}{'long' not accepted}%
    \exProtected\ekvdefNoVal\exSetName{#1}{#3}%
  }
\ekvpDefType{exdef}{store}
  {%
    \ekvpAssertOneValue{#3}%
    \ifdefined#3\else
      \let#3\empty
```

```
    \fi
    \protected\exLong\ekvdef\exSetName{#1}{\edef#3{\unexpanded{##1}}}%
  }
```

Now we need a user facing macro that puts the pieces together (this uses `\NewDocumentCommand` instead of `\newcommand` because the former defines macros `\protected`).

```
\NewDocumentCommand\exdefinekeys{m +m}
  {\def\exSetName{#1}\ekvpParse{exdef}{#2}}
```

Now we got that sorted so we can use our little parser:

```
\newif\ifbar
\exdefinekeys{exampleset}
  {
     long store foo = \myfoo
    ,protected noval bar = \bartrue
    ,code baz = baz was called with \detokenize{#1}
  }
\ekvset{exampleset}{foo=Foobar,bar,baz=\empty}
\ifbar bar was true so: \myfoo\fi
```

baz was called with \empty bar was true so: Foo-bar

Example: With this example we want to take a closer look at the expansion of `\ekvpParse`. So please bear with me if this doesn't make much sense otherwise. One of the issues is that *prefixes* are a somewhat unordered concept, and only *types* must come last. We could juggle with flags (an expandable data-storage, see subsection 2.5) to overcome this somehow just to define some pseudo-syntax here, but I guess that's not worth it. Anyhow, here goes nothing.

First we need a parser

```
\ekvpNewParser{exexp}
```

and a *prefix*. We could define one that ensures the name starts of with a letter. We also want each element to start a new line, which we do using an auto prefix.

```
\newcommand\ifletter{}
\long\def\ifletter#1#2\stop{\ifcat a\noexpand#1}
\ekvpDefPrefix{exexp}{letter}
  {\ekvpAssertIf{\ifletter#2\stop}{not a letter}}
  { (really a letter)}
\ekvpDefAutoPrefix{exexp}{\noindent}{\par}
```

Finally we define a *type* and a `NoType` rule:

```
\ekvpDefType{exexp}{*}{$#1\cdot#3 = \the\numexpr#1*#3\relax$}
\ekvpDefNoType{exexp}{the #3th letter is #1}
```

Now we want to see whether the thing is expandable:

```
\raggedright
\edef\foo{\ekvpParse{exexp}{letter e = 5, * 4 = \empty3}}
1st full expansion
\texttt{\meaning\foo}\par
\medskip
\edef\foo{\foo}
2nd full expansion
```

```
\texttt{\meaning\foo}\par
\medskip
\foo
```

1st full expansion `macro:->\noindent \ekvpAssertIf {\ifletter e\stop }{not a letter}\ekvpEOP {letter e}the 5th letter is e\ekvpEOT {letter e} (really a letter)\par \ekvpEOA {letter e}\noindent \ekvpEOP {* 4}$4\cdot \empty 3 = \the \numexpr 4*\empty 3\relax $\ekvpEOT {* 4}\par \ekvpEOA {* 4}`

2nd full expansion `macro:->\noindent the 5th letter is e (really a letter)\par \noindent $4\cdot 3 = 12$\par`

the 5th letter is e (really a letter)
$4 \cdot 3 = 12$

6 Comparisons

This section 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, comparisons are done against the base **expkv** not counting other packages in **expkv|bundle** that extend it, so "bigger feature set" might not necessarily be true if everything is included), 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 two tests (for empty key names and only a single equal sign) which are omitted.

keyval is the fastest ⟨*key*⟩=⟨*value*⟩ package there is and has a minimal feature set with a slightly different way how it handles keys without values compared to **expkv**. 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~~feature, 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 outer most braces aren'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}’
```

keyval doesn't work with non-standard category codes of `=` and `,`. Also if a ⟨*key*⟩=⟨*value*⟩ pair contains multiple equals signs outside of braces everything post the first is silently ignored so the following two inputs yield identical outputs:

```
\setkeys{foo}{bar=baz}
\setkeys{foo}{bar=baz=and more}
```

xkeyval is pretty slow (yet not the slowest), but it provides more functionality, e.g., it has an interface to disable a list of keys, can search multiple sets simultaneously, and has an intriguing mechanism it calls "Pointers" to save the value of particular keys for later reuse. 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.

xkeyval does work with non-standard category codes of `=` and `,`, but the used mechanism fails if the input contains a mix of different category codes for the same character. Just like with keyval equals signs after the first and everything after those is ignored.

ltxkeys is no longer compatible with the LaTeX kernel starting with the release 2020-10-01. It is by far the slowest ⟨*key*⟩=⟨*value*⟩ package I've tested – 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. Some of the more intriguing features (e.g., the `\argpattern` mechanism) didn't work for me. 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).

ltxkeys works with non-standard category codes, it also silently ignores any additional equals signs and the following tokens.

l3keys is at the slower end of the midfield yet not unusably slow, 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 has pretty much been bound to expl3 code before. Nowadays the LaTeX kernel has an interface with the macros `\DeclareKeys`, `\SetKeys`, and `\ProcessKeyOptions` that provides access to l3keys from the LaTeX 2_ε layer as well as parsing package options with it. Because of the `\ProcessKeyOptions` macro and its features the only two viable options to provide ⟨*key*⟩=⟨*value*⟩ options for new projects in my opinion are the kernel's methods and **expkv|opt** as those are the only two until now up to my knowledge that support parsing the raw options, and future options.

l3keys handles active commas and equals signs fine. Multiple equals signs lead to an error if additional equals signs aren't nested inside of braces, so perfectly predictable behaviour here.

pgfkeys is among the top 4 of speed if one uses `\pgfqkeys` over `\pgfkeys`, else the initialisation parsing the family path takes roughly 43 ops and moves it two spots down the list (so in Table 1 both p_0 and T_0 would be about 43 ops bigger if `\pgfkeys{`⟨*path*⟩`/.cd,`⟨*keys*⟩`}` was used instead). It has an *enormous* feature set. It stores keys in a way that reminds one of folders in a Unix system which allows interesting features and has other syntactic sugars. It is another package that implements something like the **exp:notation** with less different options though. To get the best performance `\pgfqkeys` was used in the benchmark. 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.

pgfkeys silently drops anything after the second unbraced equals sign in a ⟨*key*⟩=⟨*value*⟩ pair.

kvsetkeys with kvdefinekeys is in the slower midfield, but it works even if commas and equals have category codes different from 12 (just as some other packages in this list). It has quadratic run-time unlike most other ⟨*key*⟩=⟨*value*⟩ implementations which behave linear. The features of the keys are equal to those of keyval, the parser adds handling of unknown keys.

kvsetkeys does include any additional equals sign in the value. But any active equals sign is turned into one of category code 12 if it's not nested in braces. Also spaces around superfluous equals signs are stripped. So the following all result in the same:

```
\kvsetkeys{foo}{bar=baz=morebaz}
\kvsetkeys{foo}{bar=baz =morebaz}
\kvsetkeys{foo}{bar=baz= morebaz}
\kvsetkeys{foo}{bar=baz = morebaz}
```

options is in the midfield of speed. It is faster per individual key than pgfkeys but has no shortcut like \pgfqkeys. It has a much bigger feature set than **expkv**. Similar to pgfkeys it uses a folder like structure, makes searching multiple paths easy, incorporates package options and more. It also features a form of expansion control, predefined expansion kinds are limited though one can define additional ones. Unfortunately it also suffers from the premature unbracing bug keyval has.

options can't handle non-standard category codes and will silently ignore superfluous equals signs and following tokens.

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 added functionality so that it can be used more like most of the other ⟨*key*⟩=⟨*value*⟩ packages. Speed wise it is pretty close to **expkv**. 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*⟩.

simplekv can't correctly handle non-standard category codes. It silently ignores any unbraced equals sign beyond the first and any following tokens.

YAX is the second slowest package I've tested. 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, quite the contrary, it's just atypical). It has the premature unbracing bug, too. YAX features some prefixes one can use to make an assignment use \edef, \gdef or \xdef so has something that comes close to expansion control. 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
```

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 original expkv's 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	13.6	2.2	7.2	yes	yes	2022-05-29
expkv	16.7	3.1	5.8	no	no	2023-01-10
simplekv	19.9	2.9	15.1	no	yes	2022-10-01
pgfkeys	24.5	2.2	10.3	yes	yes	2021-05-15
options	23.3	16.2	20.4	yes	yes	2015-03-01
kvsetkeys	*	*	40.4	no	no	2022-10-05
l3keys	70.6	35.6	32.2	no	no	2022-12-17
xkeyval	255.9	221.3	173.4	yes	yes	2022-06-16
YAX	438.2	131.8	114.8	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 $p_2 = 7.6$, $p_1 = 47.7$, and $p_0 = 58.0$. 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 $p_0 = 6547.5$.

This seems important to state as YAX supports two different input syntaxes, the tested one was the one closer to traditional ⟨*key*⟩=⟨*value*⟩ input.

YAX won't handle non-standard category codes correctly. Superfluous equals signs end up in the value in an unaltered form (just like with expkv).

List of Examples

expkv

expkv|cs

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.

T