Abstract
exPkvDef provides a small \langle key\rangle=(\langle value\rangle) interface to define keys for exPkv. Key-types are declared using prefixes, similar to static typed languages. The stylised name is exPkvDef but the files use expkv-def, this is due to CTAN-rules which don't allow | in package names since that is the pipe symbol in *nix shells.

Contents
1 Documentation ...................................... 2
  1.1 Macros .............................................. 2
  1.2 Prefixes ............................................. 2
    1.2.1 p-Prefixes .................................... 2
    1.2.2 t-Prefixes .................................... 3
  1.3 Bugs ............................................... 6
  1.4 Example ........................................... 6
  1.5 License ............................................ 7

2 Implementation ................................... 8
  2.1 The \LaTeX Package ................................ 8
  2.2 The Generic Code ................................ 8
    2.2.1 Key Types .................................. 10
    2.2.2 Key Type Helpers ............................ 18
    2.2.3 Tests ......................................... 18
    2.2.4 Messages .................................. 20

Index .................................................. 22
  *jspratte@yahoo.de
1 Documentation

Since the trend for the last couple of years goes to defining keys for a \(\langle key\rangle=\langle value\rangle\) interface using a \(\langle key\rangle=\langle value\rangle\) interface, I thought that maybe providing such an interface for \texttt{expkv} will make it more attractive for actual use, besides its unique selling points of being fully expandable, and fast and reliable. But at the same time I don’t want to widen \texttt{expkv}’s initial scope. So here it is \texttt{expkv def}, go define \(\langle key\rangle=\langle value\rangle\) interfaces with \(\langle key\rangle=\langle value\rangle\) interfaces.

Unlike many of the other established \(\langle key\rangle=\langle value\rangle\) interfaces to define keys, \texttt{expkv def} works using prefixes instead of suffixes (e.g., \texttt{/tl\_set:N} of \texttt{l3keys}) or directory like handlers (e.g., \texttt{/store in} of \texttt{pgfkeys}). This was decided as a personal preference, more over in \TeX{} parsing for the first space is way easier than parsing for the last one. \texttt{expkv def}’s prefixes are sorted into two categories: \(p\)-type, which are equivalent to \TeX{}’s prefixes like \texttt{\long}, and \(t\)-type defining the type of the key. For a description of the available \(p\)-prefixes take a look at subsection 1.2.1, the \(t\)-prefixes are described in subsection 1.2.2.

\texttt{expkv def} is usable as generic code and as a \LaTeX{} package. It’ll automatically load \texttt{expkv} in the same mode as well. To use it, just use one of
\begin{verbatim}
\usepackage{expkv-def} \% LaTeX
\input expkv-def \% plainTeX
\end{verbatim}

1.1 Macros

Apart from version and date containers there is only a single user-facing macro, and that should be used to define keys.
\begin{verbatim}
\ekvdefinekeys \ekvdefinekeys{(set)}{(\langle key\rangle=\langle value\rangle, ...)}
\end{verbatim}

In \(\langle set\rangle\), define \(\langle key\rangle\) to have definition \(\langle value\rangle\). The general syntax for \(\langle key\rangle\) should be
\begin{verbatim}
\langle prefix\rangle \langle name\rangle
\end{verbatim}

Where \(\langle prefix\rangle\) is a space separated list of optional \(p\)-type prefixes followed by one \(t\)-type prefix. The syntax of \(\langle value\rangle\) is dependent on the used \(t\)-prefix.

\begin{verbatim}
\ekvdDate \ekvdVersion
\end{verbatim}

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

1.2 Prefixes

As already said there are \(p\)-prefixes and \(t\)-prefixes. Not every \(p\)-prefix is allowed for all \(t\)-prefixes.

1.2.1 \(p\)-Prefixes

The \(p\)-type prefixes are pretty simple by nature, so their description is pretty simple. They affect the \(\langle key\rangle\) at use-time, so omitting \texttt{long} doesn’t mean that a \(\langle definition\rangle\) can’t contain a \texttt{\par} token, only that the \(\langle key\rangle\) will not accept a \texttt{\par} in \(\langle value\rangle\).
The following key will be defined \protected. Note that key-types which can't be defined expandable will always use \protected.

The following key will be defined \long.

1.2.2 t-Prefixes

Since the p-type prefixes apply to some of the t-prefixes 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 \langle key \rangle = \{\langle definition \rangle\} protected long

Define \langle key \rangle to expand to \langle definition \rangle. The \langle key \rangle will require a \langle value \rangle for which you can use \#1 inside \langle definition \rangle. The ecode variant will fully expand \langle definition \rangle inside an \edef.

noval
noval \langle key \rangle = \{\langle definition \rangle\} protected long

The noval type defines \langle key \rangle to expand to \langle definition \rangle. The \langle key \rangle will not take a \langle value \rangle. enoval fully expands \langle definition \rangle inside an \edef.

default
default \langle key \rangle = \{\langle definition \rangle\} protected long

This serves to place a default \langle value \rangle for a \langle key \rangle that takes an argument, the \langle key \rangle can be of any argument-grabbing kind, and when used without a \langle value \rangle it will be passed \langle definition \rangle instead. The qdefault variant will expand the \langle key \rangle's code once, so will be slightly quicker, but not change if you redefine \langle key \rangle. The edefault on the other hand fully expands the \langle key \rangle-code with \langle definition \rangle as its argument inside of an \edef.

initial
initial \langle key \rangle = \{\langle value \rangle\} protected long

With initial you can set an initial \langle value \rangle for an already defined argument taking \langle key \rangle. It'll just call the key-macro of \langle key \rangle and pass it \langle value \rangle. The einitial variant will expand \langle value \rangle using an \edef expansion prior to passing it to the key-macro and the oinitial variant will expand the first token in \langle value \rangle once.
bool \langle key \rangle = \langle cs \rangle

The \langle cs \rangle should be a single control sequence, such as \texttt{\iffoo}. This will define \langle key \rangle 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 without a \langle value \rangle it'll have the same effect as if you use \langle key \rangle = true. bool and gbool will behave like \TeX-if-s so either be \texttt{\iftrue} or \texttt{\iffalse}. The boolTF and gboolTF variants will both take two arguments and if true the first will be used else the second, so they are always either \texttt{\@firstoftwo} or \texttt{\@secondoftwo}. The variants with a leading \texttt{g} will set the control sequence globally, the others locally. If \langle cs \rangle is not yet defined it'll be initialised as the \texttt{false} version. Note that the initialisation is \texttt{not} done with \texttt{\newif}, so you will not be able to do \texttt{\footrue} outside of the \langle key \rangle = \langle value \rangle interface, but you could use \texttt{\newif} yourself. Even if the \langle key \rangle will not be \texttt{\protected} the commands which execute the \texttt{true} or \texttt{false} choice will be, so the usage should be safe in an expansion context (e.g., you can use \texttt{edefault} \langle key \rangle = \texttt{false} without an issue to change the default behaviour to execute the \texttt{false} choice).

store \langle key \rangle = \langle cs \rangle

The \langle cs \rangle should be a single control sequence, such as \texttt{\foo}. This will define \langle key \rangle to store \langle value \rangle inside of the control sequence. If \langle cs \rangle isn't yet defined it will be initialised as empty. The variants behave similarly to their \texttt{\edef}, \texttt{\edef}, \texttt{\gdef}, and \texttt{\xdef} counterparts, but store and gstore will allow you to store macro parameters inside of them by using \texttt{\unexpanded}.

data \langle key \rangle = \langle cs \rangle

The \langle cs \rangle should be a single control sequence, such as \texttt{\foo}. This will define \langle key \rangle to store \langle value \rangle inside of the control sequence. But unlike the \texttt{store} type, the macro \langle cs \rangle will be a switch at the same time, it'll take two arguments and if \langle key \rangle was used expands to the first argument followed by \langle value \rangle in braces, if \langle key \rangle was not used \langle cs \rangle will expand to the second argument (so behave like \texttt{\@secondoftwo}). The idea is that with this type you can define a key which should be typeset formatted. The \texttt{edata} and \texttt{xdata} variants will fully expand \langle value \rangle, the \texttt{gdata} and \texttt{xdata} variants will store \langle value \rangle inside \langle cs \rangle globally. The \texttt{p}-prefixes will only affect the key-macro, \langle cs \rangle will always be expandable and \texttt{\long}.

dataT \langle key \rangle = \langle cs \rangle

Just like data, but instead of \langle cs \rangle grabbing two arguments it'll only grab one, so by default it'll behave like \texttt{\@gobble}, and if a \langle value \rangle was given to \langle key \rangle the \langle cs \rangle will behave like \texttt{\@firstoftwo} appended by \langle \langle value \rangle \rangle.

int \langle key \rangle = \langle cs \rangle

The \langle cs \rangle should be a single control sequence, such as \texttt{\foo}. An int key will be a \TeX-counter register. If \langle cs \rangle isn't defined yet, \texttt{\newcount} will be used to initialise it. The \texttt{eint} and \texttt{xint} versions will use \texttt{\numexpr} to allow basic computations in their \langle value \rangle. The \texttt{gint} and \texttt{xint} variants set the register globally.

dimen \langle key \rangle = \langle cs \rangle

The \langle cs \rangle should be a single control sequence, such as \texttt{\foo}. This is just like \texttt{int} but uses a dimen register, \texttt{\newdimen} and \texttt{\dimexpr} instead.
skip \( \operatorname{skip} \langle \text{key} \rangle = \langle \text{cs} \rangle \)

The \( \langle \text{cs} \rangle \) should be a single control sequence, such as \( \backslash \text{foo} \). This is just like \texttt{int} but uses a skip register, \texttt{\newskip} and \texttt{\glueexpr} instead.

toks \( \operatorname{toks} \langle \text{key} \rangle = \langle \text{cs} \rangle \)

The \( \langle \text{cs} \rangle \) should be a single control sequence, such as \( \backslash \text{foo} \). Store \( \langle \text{value} \rangle \) inside of a toks-register. The g variants use \texttt{\global}, the app variants append \( \langle \text{value} \rangle \) to the contents of that register. If \( \langle \text{cs} \rangle \) is not yet defined it will be initialised with \texttt{\newtoks}.

box \( \operatorname{box} \langle \text{key} \rangle = \langle \text{cs} \rangle \)

The \( \langle \text{cs} \rangle \) should be a single control sequence, such as \( \backslash \text{foo} \). Typesets \( \langle \text{value} \rangle \) into a \texttt{\hbox} and stores the result in a box register. The boxes are colour safe. \texttt{\expandafter} doesn’t provide a vbox type.

meta \( \operatorname{meta} \langle \text{key} \rangle = \{\langle \text{key} \rangle=\langle \text{value} \rangle, \ldots\} \)

This key type can set other keys, you can access the \( \langle \text{value} \rangle \) which was passed to \( \langle \text{key} \rangle \) inside the \( \langle \text{key} \rangle=\langle \text{value} \rangle \) list with \#1. It works by calling a sub-\texttt{\ekvset} on the \( \langle \text{key} \rangle=\langle \text{value} \rangle \) list, so a set key will only affect that \( \langle \text{key} \rangle=\langle \text{value} \rangle \) list and not the current \texttt{\ekvset}.

smeta \( \operatorname{smeta} \langle \text{key} \rangle = \{\langle \text{set} \rangle\{\langle \text{key} \rangle=\langle \text{value} \rangle, \ldots\} \)

Yet another meta variant. An smeta key will take a \( \langle \text{value} \rangle \) which you can access using \#1, but it sets the \( \langle \text{key} \rangle=\langle \text{value} \rangle \) list inside of \( \langle \text{set} \rangle \), so is equal to \texttt{\ekvset{\langle \text{set} \rangle}\{\langle \text{key} \rangle=\langle \text{value} \rangle, \ldots\}}.

smeta \( \operatorname{smeta} \langle \text{key} \rangle = \{\langle \text{set} \rangle\{\langle \text{key} \rangle=\langle \text{value} \rangle, \ldots\} \)

And the last meta variant. smeta is a combination of smeta and nmeta. It doesn’t take an argument and sets the \( \langle \text{key} \rangle=\langle \text{value} \rangle \) list inside of \( \langle \text{set} \rangle \).

set \( \operatorname{set} \langle \text{key} \rangle = \{\langle \text{set} \rangle \)

This will define \( \langle \text{key} \rangle \) to change the set of the current \texttt{\ekvset} invocation to \( \langle \text{set} \rangle \). You can omit \( \langle \text{set} \rangle \) (including the equals sign), which is the same as using set \( \langle \text{key} \rangle = \{\langle \text{key} \rangle \). The created set key will not take a \( \langle \text{value} \rangle \). Note that just like in \texttt{\expandafter} it’ll not be checked whether \( \langle \text{set} \rangle \) is defined and you’ll get a low-level \TeX error if you use an undefined \( \langle \text{set} \rangle \).

choice \( \operatorname{choice} \langle \text{key} \rangle = \{\langle \text{value} \rangle=\langle \text{definition} \rangle, \ldots\} \)

Defines \( \langle \text{key} \rangle \) to be a choice key, meaning it will only accept a limited set of values. You should define each possible \( \langle \text{value} \rangle \) inside of the \( \langle \text{value} \rangle=\langle \text{definition} \rangle \) list. If a defined \( \langle \text{value} \rangle \) is passed to \( \langle \text{key} \rangle \) the \( \langle \text{definition} \rangle \) will be left in the input stream. You can make individual values protected inside the \( \langle \text{value} \rangle=\langle \text{definition} \rangle \) list. By default a choice key is expandable, an undefined \( \langle \text{value} \rangle \) will throw an error in an expandable way.
1.3 Bugs
I don’t think there are any (but every developer says that), if you find some please let me
know, either via the email address on the first page or on GitHub: https://github.com/
Skillmon/tex_expkv-def

1.4 Example
The following is an example code defining each base key-type once. Please admire the
very creative key-name examples.
\begin{verbatim}
\ekvdefinekeys{example}
{
,long code keyA = #1
,local keyB = $\keyB$
,bool keyC = \keyC
,store keyD = \keyD
,data keyE = \keyE
,store dataT keyF = \keyF
,int keyG = \keyG
,store dimen keyH = \keyH
,skip keyI = \keyI
,toks keyJ = \keyJ
,default keyK = \empty t e s t
,choice keyL =
|  ,protected 1 = \texttt{ttt}a
| ,2 = b
| ,3 = c
| ,4 = d
| ,5 = e
|,edefault keyL = 2
,meta keyM = {keyA=#1,keyB=false}
,data
}
\end{verbatim}

Since the data type might be a bit strange, here is another usage example for it.
\begin{verbatim}
\ekvdefinekeys{ex}
{
,data name = \Pname
,data age = \Page
,dataT hobby = \Phobby
}
\newcommand\Person[1]{
\begin{verbatim}
}
\end{verbatim}


\end{verbatim}
\texttt{\textbackslash ekvset \{ex\]\{#1\}\%}
\begin{description}
\item[\texttt{\textbackslash Pname}] \texttt{\textbackslash errmessage\{A person requires a name\}}
\item[\texttt{\textbackslash Age}] \texttt{\textbackslash Page\{\textit{\texttt{\textbackslash textit}}\}\texttt{\textbackslash errmessage\{A person requires an age\}}}
\item[\texttt{\textbackslash hobby}] \texttt{\textbackslash item[\texttt{\textbackslash Hobbies}]}
\end{description}
\endgroup

\Person[\texttt{name=}Jonathan P. Spratte, \texttt{age=}young, \texttt{hobby=}\TeX\ coding]
\Person[\texttt{name=}Some User, \texttt{age=}unknown, \texttt{hobby=}Reading Documentation]
\Person[\texttt{name=}Anybody, \texttt{age=}any]

In this example a person should have a name and an age, but doesn’t have to have hobbies. The name will be displayed as the description item and the age in Italics. If a person has no hobbies the description item will be silently left out. The result of the above code looks like this:

Jonathan P. Spratte
Age young
Hobbies \TeX\ coding

Some User
Age unknown
Hobbies Reading Documentation

Anybody
Age any

1.5 License

Copyright © 2020 Jonathan P. Spratte

This work may be distributed and/or modified under the conditions of the L\TeX\ Project Public License (LPPL), either version 1.3c of this license or (at your option) any later version. The latest version of this license is in the file:
http://www.latex-project.org/lppl.txt
This work is “maintained” (as per LPPL maintenance status) by Jonathan P. Spratte.
2 Implementation

2.1 The \LaTeX Package

Just like for \texttt{expkv} we provide a small \LaTeX package that sets up things such that we behave nicely on \LaTeX packages and files system. It’ll \texttt{\input} the generic code which implements the functionality.

\begin{verbatim}
\RequirePackage{expkv}
def\ekvd@tmp
{%
   \ProvidesFile{expkv-def.tex}%
   [\ekvdDate space v\ekvdVersion space a key-defining frontend for expkv]%
}%
\input{expkv-def.tex}
\ProvidesPackage{expkv-def}%
[\ekvdDate space v\ekvdVersion space a key-defining frontend for expkv]
\end{verbatim}

2.2 The Generic Code

The rest of this implementation will be the generic code.

Load \texttt{expkv} if the package didn’t already do so – since \texttt{expkv} has safeguards against being loaded twice this does no harm and the overhead isn’t that big. Also we reuse some of the internals of \texttt{expkv} to save us from retyping them.

\begin{verbatim}
\input expkv
\end{verbatim}

We make sure that \texttt{expkv-def.tex} is only input once:

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

\texttt{\ekvdVersion} and \texttt{\ekvdDate} We’re on our first input, so let’s store the version and date in a macro.

\begin{verbatim}
\def\ekvdVersion{0.2}
\def\ekvdDate{2020-03-30}
\end{verbatim}

\textit{(End definition for \ekvdVersion and \ekvdDate. These functions are documented on page 2.)}

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

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

\texttt{\ekvd@tmp} will be reused later to handle expansion during the key defining. But we don’t need it to ever store information long-term after \texttt{expkv-def} was initialized.

\begin{verbatim}
\ekvd@long \ekvd@prot
\ekvd@clear@prefixes \ekvd@empty
\end{verbatim}

\texttt{expkv-def} will use \texttt{\ekvd@long} and \texttt{\ekvd@prot} to store whether a key should be defined as \texttt{\long} or \texttt{\protected}, and we have to clear them for every new key. By default they’ll just be empty.

\begin{verbatim}
\def\ekvd@empty{}
\protected\def\ekvd@clear@prefixes
{%
\end{verbatim}
\let\ekvd@long\ekvd@empty
\let\ekvd@prot\ekvd@empty
\ekvd@clear@prefixes

(End definition for \ekvd@long and others.)

\ekvdefinekeys
This is the one front-facing macro which provides the interface to define keys. It’s using \ekvparse to handle the \texttt{(key)\texttt{=\texttt{value}}} list, the interpretation will be done by \ekv@noarg and \ekvd@. The \texttt{(set)} for which the keys should be defined is stored in \ekvd@set.

\protected\def\ekvdefinekeys#1{%
  \def\ekvd@set{#1}%
  \ekvparse\ekvd@noarg\ekvd@
}

(End definition for \ekvdefinekeys. This function is documented on page 2.)

\ekv@noarg
\ekvd@\ekv@noarg just places a special marker and gives control to \ekvd@. \ekvd@ has to test whether there is a space inside the key and if so calls the prefix grabbing routine, else we throw an error and ignore the key.

\protected\def\ekv@noarg@mark\ekv@noarg#1{
  \ekv@strip{#1}\ekv@mark
}

(End definition for \ekv@noarg and \ekvd@)

\ekvd@prefix
\ekvd@prefix\expandafter\prefix\long and \protected\ prefixes which use \texttt{@p@} in their \texttt{name}, the other being key-types (\texttt{code, int, etc.}) which use \texttt{@t@} instead. \ekvd@prefix splits at the first space and checks whether its a \texttt{@p@} or \texttt{@t@} type prefix. If it is neither throw an error and gobble the definition (the value).

\protected\def\ekvd@prefix#1\ekvd@prefix@{\
  \ekv@strip{#1}\ekv@mark
  \protected\def\ekvd@prefix@#1\ekvd@stop{\
    \ekv@ifdefined{ekvd@t@#1}{\csname ekvd@t@#1\endcsname{#2}}{\
      \ekv@ifdefined{ekvd@p@#1}{\csname ekvd@p@#1\endcsname{#2}}{\
        \ekvd@err@undefined@prefix{#1}\@gobble}
    }
  }
}

(End definition for \ekvd@prefix and \ekvd@prefix@)
The \texttt{p@} type prefixes are all just modifying a following \texttt{t@} type, so they will need to search for another prefix. This is true for all of them, so we use a macro to handle this. It’ll throw an error if there is no other prefix.

\begin{alltt}
\begin{verbatim}
\protected\def\ekvd@prefix@after@p#1\%
  {\ekvd@ifspace{#1}
    {\ekvd@prefix#1\ekv@stop}
    {\expandafter\ekvd@err@missing@prefix\expandafter{\ekv@gobble@mark#1}}\%\@gobble
  }
\end{verbatim}
\end{alltt}

(End definition for \texttt{ekvd@prefix@after@p}.)

Define the \texttt{p@} type prefixes, they all just store some information in a temporary macro and call \texttt{ekvd@prefix@after@p}.

\begin{alltt}
\begin{verbatim}
\protected\def\ekvd@p@long{\let\ekvd@long\long\ekvd@prefix@after@p}
\protected\def\ekvd@p@protected{\let\ekvd@prot\protected\ekvd@prefix@after@p}
\let\ekvd@p@protect\ekvd@p@protected
\end{verbatim}
\end{alltt}

(End definition for \texttt{ekvd@p@long}, \texttt{ekvd@p@protected}, and \texttt{ekvd@p@protect}.)

\subsection{Key Types}

The \texttt{set} type is quite straightforward, just define a \texttt{NoVal} key to call \texttt{ekvchangeset}.

\begin{alltt}
\begin{verbatim}
\protected\def\ekvd@set#1#2\%
  {\ekvd@assert@not@long{set #1}}\%
  {\ekvd@assert@not@protected{set #1}}\%
  {\ekvd@ifnoarg{#2}
    {\ekvdefNoVal\ekvd@set{#1}{\ekvchangeset{#1}}}\%
    {\ekv@ifempty{#2}
      {\ekvd@err@missing@definition{set #1}}\%
      {\ekvdefNoVal\ekvd@set{#1}{\ekvchangeset{#2}}}\%
    }\%
  }
\end{verbatim}
\end{alltt}

(End definition for \texttt{ekvd@set}.)

Another pretty simple type, \texttt{noval} just needs to assert that there is a definition and that \texttt{long} wasn’t specified. There are types where the difference in the variants is so small, that we define a common handler for them, those common handlers are named with \texttt{@type@}. \texttt{noval} and \texttt{enoval} are so similar that we can use such a \texttt{@type@} macro, even if we could’ve done \texttt{noval} in a slightly faster way without it.

\begin{alltt}
\begin{verbatim}
\protected\long\def\ekvd@type@noval#1#2#3#4\%
  {\ekvd@assert@arg{#1\texttt{noval} #3}{#4}}\%
  {\ekvd@assert@not@long{\texttt{noval} #3}}\%
  {\ekvd@assert@not@protected{\texttt{noval} #3}}\%
  {\ekvletNoVal\ekvd@set{#2}{\ekvchangeset{#4}}}\%
\end{verbatim}
\end{alltt}
protected\def\ekvd@t@noval\{\ekvd@type@noval\}④
protected\def\ekvd@t@enoval\{\ekvd@type@noval e\}④

(End definition for \ekvd@type@noval, \ekvd@t@noval, and \ekvd@t@enoval.)

\ekvd@type@code code is simple as well, ecode has to use \edef on a temporary macro, since expkv doesn’t provide an \ekvedef.

\protected\long\def\ekvd@t@code#1#2#3#4\%{\ekvd@assert@arg{#1code #3}{#4}{\ekvd@prot\ekvd@long#2\ekvd@tmp##1{#4}\%\ekvlet\ekvd@set{#3}\ekvd@tmp}{\%})
\protected\def\ekvd@t@code\{\ekvd@type@code\}④
\protected\def\ekvd@t@ecode\{\ekvd@type@code e\edef\}④

(End definition for \ekvd@type@code, \ekvd@t@code, and \ekvd@t@ecode.)

\ekvd@type@default \ekvd@t@default \ekvd@t@qdefault \ekvd@type@default asserts there was an argument, also the key for which one wants to set a default has to be already defined (this is not so important for default, but qdefault requires is). If everything is good, \edef a temporary macro that expands \ekvd@set and the \csname for the key, and in the case of qdefault does the first expansion step of the key-macro.

\protected\long\def\ekvd@t@default#1#2#3#4\%{\ekvd@assert@arg{#1default #3}{#4}{\ekvd@prot\ekvfdefined\ekvd@set{#3}{\%\ekvd@assert@not@long{#1default #3}{\ekvd@prot\edef\ekvd@tmp{\ekv@name\ekvd@set{#3}\ekvd@tmp}{\%\ekvlet\ekvd@set{#3}\ekvd@tmp}{\%})\ekvletNoVal\ekvd@set{#3}\ekvd@tmp}{\%\ekvlet\ekvd@err@undefined@key{#3}}}④
\protected\def\ekvd@t@default\{\ekvd@type@default\}④
\protected\def\ekvd@t@qdefault\{\ekvd@type@default q{\expandafter}\expandafter\}④

(End definition for \ekvd@type@default, \ekvd@t@default, and \ekvd@t@qdefault.)

\ekvd@t@edefault \edefault is too different from default and qdefault to reuse the @type@ macro, as it doesn’t need \unexpanded inside of \edef.

\protected\long\def\ekvd@t@edefault#1#2#3\%{\ekvd@assert@arg{edefault #1}{#2}{\%})

11
\ekvifdefined\ekvd@set{#1} %
  \%  \ekvd@assert@not@long(edefault \#1) %
  \ekvd@prot@edef@ekvd@tmp
  {\csname\ekv@name\ekvd@set{#1}\endcsname{#2}} %
  \ekvletNoVal\ekvd@set{#1}\ekvd@tmp %
  }%  \%  \{\ekvd@err@undefined@key{#1}\}%  \%
}\%

(End definition for \ekvd@t@edefault.)

\ekvd@t@initial
\ekvd@t@oinitial
\ekvd@t@einitial

\long\def\ekvd@t@oinitial#1#2%
  {%
    \ekvd@assert@arg{oinitial \#1}{\#2}
    \%  \ekvifdefined\ekvd@set{#1} %
    \%  \ekvd@assert@not@long(oinitial \#1) %
    \ekvd@assert@not@protected{oinitial \#1} %
    \csname\ekv@name\ekvd@set{#1}\endcsname{oinitial\#2} %
    \%  \{\ekvd@err@undefined@key{#1}\}%  \%
  }

\long\def\ekvd@t@initial#1#2%
  {%
    \ekvd@assert@arg{initial \#1}{\#2}
    \%  \ekvifdefined\ekvd@set{#1} %
    \%  \ekvd@assert@not@long(inital \#1) %
    \ekvd@assert@not@protected{initial \#1} %
    \csname\ekv@name\ekvd@set{#1}\endcsname{#2} %
    \%  \{\ekvd@err@undefined@key{#1}\}%  \%
  }

\long\def\ekvd@t@einitial#1#2%
  {%
    \ekvd@assert@arg{einitial \#1}{\#2}
    \%  \ekvifdefined\ekvd@set{#1} %
    \%  \ekvd@assert@not@long(einitial \#1) %
    \ekvd@assert@not@protected{einitial \#1} %
    \edef\ekvd@tmp{\#2} %
    \csname\ekv@name\ekvd@set{#1}\endcsname\expandafter{\expandafter\ekvd@tmp}\expandafter %
    \%  \{\ekvd@err@undefined@key{#1}\}%  \%
  }
The boolean types are a quicker version of a choice that accept true and false, and set up the NoVal action to be identical to $\langle k \rangle = \text{true}$. The true and false actions are always just letting the macro in #7 to some other macro (e.g., \iftrue).

\begin{verbatim}
\protected\def\ekvd@type@bool#1#2#3#4#5#6#7% 
 \% \ekvd@assert@filledarg{#1bool#2 #6}{#7}% 
 \% \ekvd@newlet#7#5% \ekvd@type@choice{#1bool#2}{#6}% \protected\ekvdef\ekvd@set{#6}{\let#7#4}% \protected\expandafter\def\csname\ekvd@choice@name\ekvd@set{#6}{true}\endcsname{#3\let#7#4}% \protected\expandafter\def\csname\ekvd@choice@name\ekvd@set{#6}{false}\endcsname{#3\let#7#5}% 
\} 
\protected\def\ekvd@t@bool{\ekvd@type@bool{}{}{}\iftrue\iffalse}
\protected\def\ekvd@t@gbool{\ekvd@type@bool g{}\global\iftrue\iffalse}
\protected\def\ekvd@t@boolTF{\ekvd@type@bool{}{TF}{\@firstoftwo\@secondoftwo}
\protected\def\ekvd@t@gboolTF{\ekvd@type@bool g{TF}\global\@firstoftwo\@secondoftwo}
\end{verbatim}

Set up our boxes. Though we’re a generic package we want to be colour safe, so we put an additional grouping level inside the box contents, for the case that someone uses color.

\begin{verbatim}
\protected\def\ekvd@type@data#1#2#3#4#5#6#7% 
 \% \ekvd@assert@filledarg{#1data#2 #6}{#7}% 
 \% \ekvd@newlet#7#3% \protected\ekvd@long\ekvdef\ekvd@set{#6}{\long#4#7####1#5{####1{##1}}}% 
\} 
\protected\def\ekvd@t@data{\ekvd@type@data{}{}\@secondoftwo\def{####2}}
\protected\def\ekvd@t@edata{\ekvd@type@data e{}\@secondoftwo\edef{####2}}
\protected\def\ekvd@t@gdata{\ekvd@type@data g{}\@secondoftwo\gdef{####2}}
\protected\def\ekvd@t@xdata{\ekvd@type@data x{}\@secondoftwo\xdef{####2}}
\protected\def\ekvd@t@dataT{\ekvd@type@data{}T\@gobble\def{}}
\protected\def\ekvd@t@edataT{\ekvd@type@data eT\@gobble\edef{}}
\protected\def\ekvd@t@gdataT{\ekvd@type@data gT\@gobble\gdef{}}
\protected\def\ekvd@t@xdataT{\ekvd@type@data xT\@gobble\xdef{}}
\end{verbatim}

\begin{verbatim}
\end{verbatim}
\evd@assert@filledarg{#!box #3}{#4} \%
% \evd@newreg#4{box} \%
\protected\evd@long\evd@set{#3} \%
{#2}@setbox#4\hbox{\begingroup##1\endgroup} \%
% 
\protected\def\evd@t@box{\evd@type@box{}{}}
\protected\def\evd@t@gbox{\evd@type@box g\global}
(End definition for \evd@type@box, \evd@t@box, and \evd@t@gbox.)

\evd@type@toks
\evd@t@toks
\evd@t@gtoks
Similar to box, but set the toks.
\protected\def\evd@type@toks#1#2#3#4\%
{\evd@assert@filledarg{#1@toks #3}{#4} \%
  \evd@newreg#4{toks} \%
  \protected\evd@long\evd@set{#3}{#2#4{##1}} \%
} \%
\protected\def\evd@t@toks{\evd@type@toks{}{}}
\protected\def\evd@t@gtoks{\evd@type@toks{g}\global}
(End definition for \evd@type@toks, \evd@t@toks, and \evd@t@gtoks.)

\evd@type@apptoks
\evd@t@apptoks
\evd@t@gapptoks
Just like toks, but expand the current contents of the toks register to append the new contents.
\protected\def\evd@type@apptoks#1#2#3#4\%
{\evd@assert@filledarg{#1@apptoks #3}{#4} \%
  \evd@newreg#4{toks} \%
  \protected\evd@long\evd@set{#3}{#2#4\expandafter{\the#4##1}} \%
} \%
\protected\def\evd@t@apptoks{\evd@type@apptoks{}{}}
\protected\def\evd@t@gapptoks{\evd@type@apptoks{g}\global}
(End definition for \evd@type@apptoks, \evd@t@apptoks, and \evd@t@gapptoks.)

\evd@type@reg
\evd@t@int
\evd@t@eint
\evd@t@gint
\evd@t@xint
\evd@t@dimen
\evd@t@edimen
\evd@t@gdimen
\evd@t@xdimen
\evd@t@skip
\evd@t@eskip
\evd@t@gskip
\evd@t@xskip
The \evd@type@reg can handle all the types for which the assignment will just be
\langle register\rangle=\langle value\rangle.
\protected\def\evd@type@reg#1#2#3#4#5#6#7\%
{\evd@assert@filledarg{#1@reg#1}{#7} \%
  \evd@newreg#7{#2} \%
  \protected\evd@long\evd@set{#3}{#2\#4} \%
  \protected\def\evd@t@int{\evd@type@reg{int}{\count}}\{}
  \protected\def\evd@t@eint{\evd@type@reg{eint}{\count}}\{}
  \protected\def\evd@t@gint{\evd@type@reg{gint}{\count}}\%
  \protected\def\evd@t@xint{\evd@type@reg{xint}{\count}}\%
  \protected\def\evd@t@dimen{\evd@type@reg{dimen}{\count}}\%
  \protected\def\evd@t@edimen{\evd@type@reg{edimen}{\count}}\%
  \protected\def\evd@t@gdimen{\evd@type@reg{gdimen}{\count}}\%
  \protected\def\evd@t@xdimen{\evd@type@reg{xdimen}{\count}}\%
  \protected\def\evd@t@skip{\evd@type@reg{skip}{\count}}\%
  \protected\def\evd@t@eskip{\evd@type@reg{eskip}{\count}}\%
  \protected\def\evd@t@gskip{\evd@type@reg{gskip}{\count}}\%
  \protected\def\evd@t@xskip{\evd@type@reg{xskip}{\count}}\%
(End definition for \evd@type@reg, \evd@t@reg, and \evd@t@gtoks.)
\protected\def\ekvd@t@dimen{\ekvd@type@reg{dimen}{dimen}{}}
\protected\def\ekvd@t@edimen{\ekvd@type@reg{edimen}{dimen}{\dimexpr\relax}}
\protected\def\ekvd@t@gdimen{\ekvd@type@reg{gdimen}{dimen}{\global{}}}
\protected\def\ekvd@t@xdimen{\ekvd@type@reg{xdimen}{dimen}{\global{\dimexpr\relax}}}
\protected\def\ekvd@t@skip{\ekvd@type@reg{skip}{skip}{}}
\protected\def\ekvd@t@eskip{\ekvd@type@reg{eskip}{skip}{}{\glueexpr\relax}}
\protected\def\ekvd@t@gskip{\ekvd@type@reg{gskip}{skip}{\global{}}}
\protected\def\ekvd@t@xskip{\ekvd@type@reg{xskip}{skip}{\global{\glueexpr\relax}}}

(End definition for \ekvd@type@reg and others.)

\ekvd@type@store \ekvd@t@store \ekvd@t@gstore
The none-expanding store types use an \edef or \xdef and \unexpanded to be able to
also store # easily.
\protected\def\ekvd@type@store#1#2#3#4{%
  \ekvd@assert@filledarg{#1store #3}{#4}%
  {%
    %\unless\ifdefined#4\let#4\ekvd@empty\fi
    \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4{\unexpanded{\##1}}}%
  %}
%}
\protected\def\ekvd@t@store{\ekvd@type@store{e}\edef}
\protected\def\ekvd@t@gstore{\ekvd@type@store{g}\xdef}
(End definition for \ekvd@type@store, \ekvd@t@store, and \ekvd@t@gstore.)

\ekvd@type@estore \ekvd@t@estore \ekvd@t@xstore
And the straight forward estore types.
\protected\def\ekvd@type@estore#1#2#3#4{%
  \ekvd@assert@filledarg{#1store #3}{#4}%
  {%
    \ekvd@newlet#4\ekvd@empty
    \protected\ekvd@long\ekvdef\ekvd@set{#3}{#2#4{\#1}}%
  %}
%}
\protected\def\ekvd@t@estore{\ekvd@type@estore{e}\edef}
\protected\def\ekvd@t@xstore{\ekvd@type@estore{x}\xdef}
(End definition for \ekvd@type@estore, \ekvd@t@estore, and \ekvd@t@xstore.)

\ekvd@type@meta \ekvd@t@meta \ekvd@t@nmeta
meta sets up things such that another instance of \ekvset will be run on the argument,
with the same \langle set \rangle.
\protected\long\def\ekvd@type@meta#1#2#3#4#5{%
  \ekvd@assert@filledarg{#1meta #4}{#5}%
  {%
    \edef\ekvd@tmp{\ekvd@set}{#5}
    \expandafter\ekvd@type@meta0\expandafter\ekvd@tmp{#3}{#5}{#2}{\ekvd@set{#4}}{\ekvd@tmp}
  %}
%}
\protected\long\def\ekvd@t@meta0{\ekvd@type@meta0{\#1}{\#2}{\#3}}
{%
  \ekvd@prot\ekvd@long\def\ekvd@tmp2{\ekvset{\#1}{\#3}}%
}
\protected\def\ekvd@t@meta{\ekvd@type@meta{}\ekvlet{##1}}
\protected\long\def\ekvd@t@nmeta#1#2% {
\ekvd@assert@not@long{nmeta #1}%
\ekvd@type@meta n\ekvletNoVal{}{#1}{#2}% }

(End definition for \ekvd@type@meta and others.)

\ekvd@type@smeta
\ekvd@type@smeta@
\ekvd@t@smeta
\ekvd@t@snmeta

smeta is pretty similar to meta, but needs two arguments inside of \texttt{⟨value⟩}, such that the first is the \texttt{⟨set⟩} for which the sub-\texttt{\ekvset} and the second is the \texttt{⟨key⟩}=\texttt{⟨value⟩} list.
\protected\long\def\ekvd@type@smeta#1#2#3#4#5% {
\ekvd@assert@twoargs{s#1meta #4}{#5}%
\expandafter\ekvd@type@smeta@\expandafter{\@secondoftwo#5}{#5}{#3}
#2\ekvd@set{#4}\ekvd@tmp
}
\protected\long\def\ekvd@type@smeta@#1#2#3% {
\expandafter\ekvd@type@meta@\expandafter{\@firstoftwo#2}{#3}{#1}
}

(End definition for \ekvd@type@smeta and others.)

\ekvd@type@choice
\ekvd@populate@choice
\ekvd@populate@choice@noarg
\ekvd@choice@prefix
\ekvd@choice@p@protected
\ekvd@choice@p@protect
\ekvd@choice@p@long
\ekvd@t@choice

The choice type is by far the most complex type, as we have to run a sub-parser on the choice-definition list, which should support the \texttt{@p@} type prefixes as well (but long will always throw an error, as they are not allowed to be long). \texttt{\ekvd@type@choice} will just define the choice-key, the handling of the choices definition will be done by \texttt{\ekvd@populate@choice}.
\protected\def\ekvd@type@choice#1#2% {
\ekvd@assert@not@long{#1 #2}%
\prot\edef\ekvd@tmp##1% {
\unexpanded{\ekvd@h@choice}{\ekvd@choice@name\ekvd@set{#2}{##1}}%
}
\ekvlet\ekvd@set{#2}\ekvd@tmp
}
\protected\def\ekvd@populate@choice#1#2% {
\ekvd@assert@not@long{#1 #2}%
\ekvd@prot\edef\ekvd@tmp##1% {
\unexpanded{\ekvd@h@choice}{\ekvd@choice@name\ekvd@set{#2}{##1}}%
}
\ekvlet\ekvd@set{#2}\ekvd@tmp
}
\ekvd@populate@choice just uses \texttt{\ekvparse} and then gives control to \texttt{\ekvd@populate@choice@noarg}, which throws an error, and \texttt{\ekvd@populate@choice@}. 
\protected\def\ekvd@populate@choice% {
\%
\ekvparse\ekvd@populate@choice@noarg\ekvd@populate@choice@
}
\protected\long\def\ekvd@populate@choice@noarg#1%
\expandafter\ekvd@err@missing@definition\expandafter{\ekvd@set@choice : #1}\%

\ekvd@populate@choice@ runs the prefix-test, if there is none we can directly define the
choice, for that \ekvd@set@choice will expand to the current choice-key’s name, which
will have been defined by \ekvd@t@choice. If there is a prefix run the prefix grabbing
routine, which was altered for @type@choice.

\protected\long\def\ekvd@populate@choice@#1#2%\%
\ekvd@clear@prefixes\%
\expandafter\expandafter\expandafter\expandafter{\ekvd@set@choice : #1}#2\%
\ekvd@clear@prefixes{#1}\%
{\ekvd@choice@prefix\ekv@mark#1\ekv@stop}\%
\expandafter\def\csname\ekvd@choice@name\ekvd@set\ekvd@set@choice{#1}\endcsname
{#2}\%

\protected\def\ekvd@choice@prefix@#1\%
{\ekv@strip{#1}\ekvd@choice@prefix@\ekv@mark}
\protected\def\ekvd@choice@prefix@#1#2\ekv@stop\%
{\ekv@ifdefined{ekvd@choice@p@#1}\%
\csname ekvd@choice@name\p@#1\endcsname\ekv@strip{#1}\%
{\ekvd@choice@prefix#2\ekv@stop}{\endcsname\%
{\ekvd@prot\expandafter\def\csname \ekv@strip{#2}\ekvd@choice@name\ekvd@set\ekvd@set@choice}\%
\endcsname\%
{\ekvd@err@undefined@prefix{#1}\@gobble}\%
}
\protected\def\ekvd@choice@p@{\let\ekvd@choice@p@\ekvd@choice@p@protected}{\let\ekvd@choice@p@\ekvd@choice@p@}\%
\protected\def\ekvd@choice@p@long\%
{\ekvd@choice@p@long\ekvd@ifspace#1%\%
{\expandafter\ekvd@choice@p@long@\expandafter{\ekv@gobble@mark#1}\ekvd@ifspace{#1}\%
{\expandafter\ekvd@err@no@long\expandafter\%
{\ekvd@set@choice : long #1}\%
}
Finally we’re able to set up the \texttt{\textbackslash@t\textbackslash@choice} macro, which has to store the current choice-key’s name, define the key, and parse the available choices.

```
\protected\long\def\ekvd@t@choice#1#2\% {
  \ekvd@assert\arg{choice \ #1}{#2}\%
  {\ekvd@type@choice{choice}{#1}\%
    \def\ekvd@set@choice{#1}\%
    \ekvd@populate@choice(#2)\%
  }%
}
```

(End definition for \texttt{\textbackslash@t\textbackslash@choice} and others.)

### 2.2.2 Key Type Helpers

There are some keys that might need helpers during their execution (not during their definition, which are gathered as \texttt{\textbackslash@type\textbackslash@} macros). These helpers are named \texttt{\textbackslash@h\textbackslash@}.

```
\def\ekvd@h@choice#1\% {
  \expandafter\ekvd@h@choice@
  \csname\ifcsname#1\endcsname#1\else relax\fi\endcsname{#1}\%
}
```

```
\def\ekvd@h@choice@#1#2\% {
  \ifx#1\relax
    \ekvd@err@choice@invalid{#2}\%
    \expandafter\@gobble
  \fi
  #1\%
}
```

(End definition for \texttt{\textbackslash@h@choice} and \texttt{\textbackslash@h@choice@}.)

### 2.2.3 Tests

#### \texttt{\textbackslash@noarg@mark}

This macro serves as a flag for the case that no \texttt{\textbackslash{value}} was specified for a key. As such it is not a test, but exists only for some tests.

```
\def\ekvd@noarg@mark\ekvd@noarg@mark\{
    \expandafter\ekvd@noarg@mark\}
```

(End definition for \texttt{\textbackslash@noarg@mark}.)

#### \texttt{\textbackslash@fi@firstoftwo}

While we can reuse many of the internals of \texttt{\textbackslash{exp\textbackslash@\textbackslash@fi\textbackslash@\textbackslash@firstoftwo}} the specific case for this branch wasn’t needed by \texttt{\textbackslash{exp\textbackslash@\textbackslash@fi\textbackslash@\textbackslash@firstoftwo}} and hence isn’t defined. We’ll need it, so we define it.

```
\long\def\ekvd@fi@firstoftwo\fi\ekvd@secondoftwo\#1\#2\fi\fi#1\}
```

(End definition for \texttt{\textbackslash@fi@firstoftwo}.)
These macros test whether a control sequence is defined, if it isn’t they define it, either via \let or via the correct \new⟨reg⟩.

\protected\def\ekvd@newlet#1#2\% {\unless\ifdefined#1\let#1#2\fi} \protected\def\ekvd@newreg#1#2\% {\unless\ifdefined#1\csname new#2\endcsname#1\fi}

(End definition for \ekvd@newlet and \ekvd@newreg)

A test for exactly two tokens can be reduced for an empty-test after gobbling two tokens, in the case that there are fewer tokens than two in the argument, only macros will be gobbled that are needed for the true branch, which doesn’t hurt, and if there are more this will not be empty.

\long\def\ekvd@assert@twoargs#1#2\% {\ekvd@ifnottwoargs{#2}\% {\ekvd@err@missing@definition{#1}}} \long\def\ekvd@ifnottwoargs#1\% {\ekvd@ifempty@gtwo#1\ekv@ifempty@false\ekv@ifempty@A\ekv@ifempty@B\@firstoftwo}

\long\def\ekvd@ifempty@gtwo#1#2{\ekv@ifempty@A

(End definition for \ekvd@assert@twoargs, \ekvd@ifnottwoargs, and \ekvd@ifempty@gtwo)

The test for an argument is just an \ifx comparison with our noarg@mark.

\long\def\ekvd@assert@arg#1#2\% {\ekvd@ifnoarg{#2}\% {\ekvd@err@missing@definition{#1}}} \long\def\ekvd@ifnoarg#1\% {\ifx\ekvd@noarg@mark#1\ekvd@fi@firstoftwo}

\long\def\ekvd@ifnoarg@or@empty#1\% {\ifx\ekvd@noarg@empty\ekvd@fi@firstoftwo}

(End definition for \ekvd@assert@arg and \ekvd@ifnoarg.)

The test for an argument is just an \ifx comparison with our noarg@mark.

\long\def\ekvd@assert@filledarg#1#2\% {\ekvd@ifnoarg@or@empty\ekvd@fi@firstoftwo}

(End definition for \ekvd@assert@filledarg and \ekvd@ifnoarg@or@empty.)
Some key-types don’t want to be \long or \protected, so we provide macros to test this and throw an error, this could be silently ignored but now users will learn to not use unnecessary stuff which slows the compilation down.

\long\def\ekvd@assert@not@long#1{%
  \ifx\ekvd@long\long\ekvd@err@no@long{#1}\fi
}\long\def\ekvd@assert@not@protected#1{%
  \ifx\ekvd@prot\protected\ekvd@err@no@protected{#1}\fi
}(End definition for \ekvd@assert@not@long and \ekvd@assert@not@protected.)

Yet another test which can be reduced to an if-empty, this time by gobbling everything up to the first space.

\long\def\ekvd@ifspace#1{%
  \ekvd@ifspace@#1 \ekv@ifempty@B \ekv@ifempty@false\ekv@ifempty@A \ekv@ifempty@B \@firstoftwo
}\long\def\ekvd@ifspace@#1 % keep this space
{\ekv@ifempty@A}
(End definition for \ekvd@ifspace and \ekvd@ifspace0.)

### 2.2.4 Messages

Most messages of \expkvdef are not expandable, since they only appear during key-definition, which is not expandable anyway.

The non-expandable error messages are boring, so here they are:

\protected\def\ekvd@err@missing@definition#1{%
  \errmessage{expkv-def Error: Missing definition for key ‘\unexpanded{#1}’}
}\protected\def\ekvd@err@missing@prefix#1{%
  \errmessage{expkv-def Error: Missing prefix for key ‘\unexpanded{#1}’}
}\protected\def\ekvd@err@undefined@prefix#1{%
  \errmessage{expkv-def Error: Undefined prefix ‘\unexpanded{#1}’}
}\protected\def\ekvd@err@undefined@key#1{%
  \errmessage{expkv-def Error: Undefined key ‘\unexpanded{#1}’}
}\protected\def\ekvd@err@no@protected#1{%
  \errmessage{expkv-def Error: prefix ‘\unexpanded{#1}’ not accepted for ‘\unexpanded{#1}’}
The expandable error messages use \ekvd@err, which is just like \ekv@err from \expkv or the way expl3 throws expandable error messages. It uses an undefined control sequence to start the error message. \ekvd@err@choice@invalid will have to use this mechanism to throw its message. Also we have to retrieve the name parts of the choice in an easy way, so we use parentheses of catcode 8 here, which should suffice in most cases to allow for a correct separation.

```
\def\ekvd@err@choice@invalid#1\ekv@stop{% 
  \ekvd@err{invalid choice '#3' ('#2', set '#1')}%
} 
```

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

```
\catcode'@=\ekvd@tmp
```

(End definition for \ekvd@err@choice@invalid and others.)
# 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.

| A               | \texttt{agdef} .................................. | 200, 204 |
| A               | \texttt{agdimen} .................................. | 4 |
| A               | \texttt{agint} .................................. | 4 |
| A               | \texttt{agskip} .................................. | 5 |
| A               | \texttt{agtoks} .................................. | 5 |
| B               | \texttt{bool} .................................. | 4 |
| B               | \texttt{boolTF} .................................. | 4 |
| B               | \texttt{box} .................................. | 5 |
| C               | \texttt{choice} .................................. | 5 |
| C               | \texttt{code} .................................. | 3 |
| D               | \texttt{data} .................................. | 4 |
| D               | \texttt{dataT} .................................. | 4 |
| D               | \texttt{default} .................................. | 3 |
| D               | \texttt{dimen} .................................. | 4 |
| E               | \texttt{ecode} .................................. | 3 |
| E               | \texttt{edata} .................................. | 4 |
| E               | \texttt{edataT} .................................. | 4 |
| E               | \texttt{edefault} .................................. | 3 |
| E               | \texttt{edimen} .................................. | 4 |
| E               | \texttt{einitial} .................................. | 3 |
| E               | \texttt{eint} .................................. | 4 |
| E               | \texttt{ekvchangeset} ......................... | 68, 72 |
| E               | \texttt{ekvDate} ......................... | 2, 5, 9, 15 |
| E               | \texttt{ekvdef} ....................... | 195, 211, 222, 232, 242, 262, 272 |
| E               | \texttt{ekvdefinekeys} .................. | 2, 27 |
| E               | \texttt{ekvDefNoVal} ................ | 68, 72, 176 |
| E               | \texttt{ekvVersion} ................ | 2, 5, 9, 15 |
| E               | \texttt{ekvIfDefined} ............. | 100, 119, 133, 146, 159 |
| E               | \texttt{ekvlet} ................... | 91, 290, 308, 321 |
| E               | \texttt{ekvletNoVal} ................ | 81, 108, 124, 294, 312 |
| E               | \texttt{ekvpars} .................. | 30, 325 |
| E               | \texttt{ekvset} ................... | 288 |
| E               | \texttt{enoval} .................... | 3 |
| E               | \texttt{eskip} .................... | 5 |
| E               | \texttt{estore} .................... | 4 |
| G               | \texttt{gapptoks} .................. | 5 |
| G               | \texttt{gbool} .................... | 4 |
| G               | \texttt{gboolTF} .................. | 4 |
| G               | \texttt{gbox} .................... | 5 |
| G               | \texttt{gdata} .................... | 4 |
| G               | \texttt{gdataT} .................. | 4 |

\texttt{\LaTeX} and \texttt{\LaTeX}\texttt{\textsc{e}} commands:

\texttt{\LaTeX}:
- \texttt{\@firstofone} ............... | 486, 502 |
- \texttt{\@firstoftwo} ................ | 187, 189, 306, 418, 441, 455 |
- \texttt{\@gobble} .................. | 48, 57, 202, 203, 204, 205, 363, 396 |
- \texttt{\@secondoftwo} ........ | 187, 189, 198, 199, 200, 201, 300, 401, 431 |
- \texttt{\@ekv@err\@} ............. | 502 |
- \texttt{\@ekv@gobble@mark} .......... | 56, 369 |
- \texttt{\@ekv@ifdefined} .......... | 43, 46, 351 |