The regexpatch package
Replacing etoolbox patching commands*

Enrico Gregorio†

Released 2012/04/15

Important preliminary notice

This is an experimental version and it might cease to work if the commands in the package l3regex are modified. When that LaTeX3 package will be declared stable, this package will replace xpatch and calling `\usepackage{regexpatch}` will load the main package.

1 Introduction

The well known etoolbox package provides a bunch of functions for patching existing commands; in particular `\patchcmd`, `\pretocmd` and `\apptocmd` that do a wonderful job, but suffer from a limitation: if some package has defined

```
\newcommand{\xyz}[1][x]{-#1!}
```

where `\xyz` has an optional argument, then `\patchcmd` and siblings cannot be used to modify the workings of `\xyz`. The same happens when a command has been defined with `\DeclareRobustCommand`.

The reason for this is TeXnical or, better, LaTeXnical. When LaTeX performs the above definition, the expansion of `\xyz` will be

```
\@protected@testopt \xyz \\xyz {x}
```

where `\@protected@testopt` is a macro that essentially checks whether we are in a "protected" context, so that expansion should not be performed all the way (in moving arguments or write operations), or not; in the former case it issues a protected version of `\xyz`, while in the latter case it expands the macro `\\xyz` that is a *single* command (yes, with a backslash in its name) which contains the real definition; a way to access this definition is to issue the command

```
\expandafter\show\csname\string\xyz\endcsname
```

which will print in the log file the message

```
> \\xyz=\long macro:
[#1]->-#1!.
```

*This file describes version 0.1dev, last revised 2012/04/15.

†E-mail: Enrico DOT Gregorio AT univr DOT it

As usual, after `->` we see the definition. In order to use `\patchcmd` to change the exclamation mark into a hyphen one must do

```
\expandafter\patchcmd\csname\string\xyz\endcsname{!}{-}{}{}
```

(see the documentation of etoolbox for details about the arguments).

A similar situation happens if `\xyz` has been defined by

```
\DeclareRobustCommand{\xyz}{something}
```

A `\show\xyz` would show the cryptic

```
> \xyz=macro:
->\protect \xyz  .
```

and only a close look reveals the clever trick used by the LaTeX team: the `\protect` is not applied to `\xyz`, but to the macro `\xyz␣` which has a space at the end of its name! And this macro is the one that contains the real definition. Indeed,

```
\expandafter\show\csname xyz\space\endcsname
```

produces the message

```
> \xyz =\long macro:
->something.
```

In this case, in order to apply `\patchcmd` we must say

```
\expandafter\patchcmd\csname xyz\space\endcsname{s}{S}{}{}
```

If the macro with `\DeclareRobustCommand` is defined to have an optional argument, say

```
\DeclareRobustCommand{\xyz}[1][x]{-#1!}
```

one has to combine the two tricks:

```
\expandafter\patchcmd\csname\string\xyz\space\endcsname{!}{-}{}{}
```

It's hard and error prone to remember all of these tricks, so this package comes to the rescue.

The package is now completely independent of etoolbox. It doesn't feature commands analogous to `\preto` and `\appto` that, in the author's opinion, are a bit dangerous, since somebody might apply them to commands defined with `\DeclareRobustCommand` or `\newrobustcmd`, with the obvious problems.

The regexpatch package uses many features of the LaTeX3 experimental packages, in particular of l3regex. This has a clear advantage: we can have a *-variant of `\xpatchcmd` that does a "replace all" that can avoid multiple uses of `\patchcmd` on the same macro. Moreover there's a very powerful `\regexpatchcmd` function that uses regular expression syntax for search and replace which can even patch commands defined under different category code setup.

For example, let's see how the LaTeX kernel defines `\strip@pt`:

```
\begingroup
  \catcode`P=12
  \catcode`T=12
  \lowercase{
    \def\x{\def\rem@pt##1.##2PT{##1\ifnum##2>\z@.##2\fi}}
  \expandafter\endgroup\x
\def\strip@pt{\expandafter\rem@pt\the}
```

The same result can be obtained by

```
\begingroup\def\defrem@pt{\endgroup
  \def\rem@pt##1.##2pt{##1\ifnum##2>\z@.##2\fi}}
\regexpatchcmd{\defrem@pt}{pt}{\cOp\cOt}{}{}
\defrem@pt
\def\strip@pt{\expandafter\rem@pt\the}
```

Perhaps not so striking, but the pattern seems to be more intuitive; however the package supplies also a function for patching the parameter text of a macro:

```
\def\rem@pt#1.#2pt{#1\ifnum#2>\z@.#2\fi}
\xpatchparametertext{\rem@pt}{pt}{\cO p \cO t}{}{}
```

Of course, reading the manual of l3regex is necessary for being able to exploit the full power of `\regexpatchcmd` or `\xpatchparametertext`; in this case, '`\cO p`' (the space in between is optional) specifies a character '`p`' with category code 'other'. Actually neither the `\cO` escape is necessary, as all letters in a replacement text in the context of regular expressions has category code 12 by default, but clarity is often to be preferred to efficiency.

2 Important notices

If the command to be patched contains '`@`-commands' in its replacement text, *always* ensure that the patching code is enclosed between `\makeatletter` and `\makeatother`. It's recommended to turn on `\tracingxpatches` when testing a patch, to get maximum information.

In the present version, commands that have been redefined via the etoolbox command `\robustify` are *not* patchable, in general. The next version will probably provide a new way for robustifying commands.

3 Commands

The main commands introduced by this package are

- `\xpretocmd`
- `\xapptocmd`
- `\xpatchcmd`
- `\regexpatchcmd`

which have the same syntax as the similar commands provided by etoolbox and apply to all kind of commands defined by

- the LaTeX kernel macros `\newcommand`, `\renewcommand`, `\providecommand`, but also `\newenvironment` and `\renewenvironment`;
- the LaTeX kernel macro for defining robust commands `\DeclareRobustCommand`;
- the etoolbox macros `\newrobustcmd`, `\renewrobustcmd`, `\providerobustcmd`.

Notice that patching the definition of the environment `foo` requires patching `\foo` or `\endfoo`.

These commands will act as the original ones if the macro to patch is not robust or with optional arguments.

There is also added functionality that etoolbox doesn't provide (at least easily for the first command):

- `\xpatchoptarg`
- `\xpatchparametertext`
- `\checkifpatchable`

Moreover the package defines

- `\xpretobibmacro`
- `\xapptobibmacro`
- `\xpatchbibmacro`
- `\regexpatchbibmacro`

that can be used to patch commands defined with biblatex's `\newbibmacro`. Say that we have

```
\newbibmacro{foo.bar}[2]{#1 and #2}
```

Then, to change `and` into `und`, we can now say

```
\xpatchbibmacro{foo.bar}{and}{und}{}{}
```

Patching these macros with etoolbox requires resorting to the *very* cryptic

```
\expandafter\patchcmd\csname abx@macro@\detokenize{foo.bar}\endcsname
  {and}{und}{}{}
```

that would become an astonishing

```
\expandafter\patchcmd\csname\expandafter\string\csname
  abx@macro@\detokenize{foo.bar}\endcsname\endcsname
  {and}{und}{}{}
```

if the original definition had been with an optional argument, say

```
\newbibmacro{foo.bar}[2][x]{#1 and #2}
```

For biblatex users there are also

- `\xpretobibdriver`
- `\xapptobibdriver`
- `\xpatchbibdriver`

- `\regexpatchbibdriver`

for patching commands defined with `\DeclareBibliographyDriver`. One could use, for patching the driver `foo`,

```
\makeatletter
\patchcmd{\blx@bbx@foo}{X}{Y}{success}{failure}
\preto{\blx@bbx@foo}{P}
\appto{\blx@bbx@foo}{A}
\makeatother
```

but having a lighter interface can be handy. Since our macros use `\pretocmd` and `\apptocmd` for consistency, remember to always use the `{success}` and `{failure}` arguments also with `\xpretobibdriver` and `\xapptobibdriver`.

Finally, the package defines the commands

- `\xshowcmd`
- `\xshowbibmacro`
- `\xshowbibdriver`
- `\tracingxpatches`

The first three are the analog of `\show` to see the "real" definition of a macro, be it defined with optional arguments or as a robust command; the `bib` ones are for the corresponding biblatex macros. The last one takes an optional argument for activating and deactivating the tracing system. So

```
\tracingxpatches
```

will activate it (it's equivalent to `\tracingxpatches[1]`), while

```
\tracingxpatches[0]
```

will stop issuing messages.

4 Syntax

`\xpretocmd{`⟨*command*⟩`}{`⟨*prepend*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`
`\xapptocmd{`⟨*command*⟩`}{`⟨*append*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`
`\xpatchcmd[*]{`⟨*command*⟩`}{`⟨*search*⟩`}{`⟨*replace*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`
`\regexpatchcmd[*]{`⟨*command*⟩`}{`⟨*search*⟩`}{`⟨*replace*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`

`\xpretobibmacro{`⟨*name*⟩`}{`⟨*prepend*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`
`\xapptobibmacro{`⟨*name*⟩`}{`⟨*append*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`
`\xpatchbibmacro[*]{`⟨*name*⟩`}{`⟨*search*⟩`}{`⟨*replace*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`
`\regexpatchbibmacro[*]{`⟨*name*⟩`}{`⟨*search*⟩`}{`⟨*replace*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`

`\xpretobibdriver{`⟨*name*⟩`}{`⟨*prepend*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`
`\xapptobibdriver{`⟨*name*⟩`}{`⟨*append*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`
`\xpatchbibdriver[*]{`⟨*name*⟩`}{`⟨*search*⟩`}{`⟨*replace*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`
`\regexpatchbibdriver[*]{`⟨*name*⟩`}{`⟨*search*⟩`}{`⟨*replace*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`

`\xshowcmd[*]{`⟨*command*⟩`}`

`\xshowbibname{`⟨*name*⟩`}`
`\xshowbibdriver{`⟨*name*⟩`}`

`\xpatchoptarg{`⟨*name*⟩`}{`⟨*replace*⟩`}`
`\xpatchparametertext{`⟨*name*⟩`}{`⟨*search*⟩`}{`⟨*replace*⟩`}{`⟨*success*⟩`}{`⟨*failure*⟩`}`
`\checkifpatchable{`⟨*name*⟩`}`
`\tracingxpatches[`⟨*number*⟩`]`

Here ⟨*command*⟩ is the command's name (with the backslash), while ⟨*name*⟩ is the string that appears as the argument to `\newbibmacro` or `\DeclareBibliographyDriver` respectively; ⟨*search*⟩, ⟨*replace*⟩, ⟨*prepend*⟩ and ⟨*append*⟩ are the list of tokens that are to be used for the specific tasks; ⟨*success*⟩ and ⟨*failure*⟩ are token lists to be executed if the patching succeeds or fails respectively. I find it useful to use `\ddt` as ⟨*failure*⟩, so that TEX will stop for the undefined control sequence when the patching fails.

In the commands whose name contains the string `regex`, both ⟨*search*⟩ and ⟨*replace*⟩ are understood to represent regular expressions.

The *-variants of the `patch` type commands means that the replacement is performed on *all* matches. With `\xshowcmd*\foo` one gets all information on `\foo`, as if the tracing system were activated, including the default optional argument, if existent. So it's best to use it before trying `\xpatchoptarg` (and all the other commands, of course).

A curiosity about optional arguments: if one defines

```
\newcommand{\foo}[1][{bar}]{-#1-}
```

then the braces around `bar` are stripped off. So with

```
\newcommand{\foo}[1][{\itshape bar}]{-#1-}
```

all text following the call of `\foo` without an optional argument would be set in italics; one needs *two* sets of braces, in this case. However,

```
\xpatchoptarg{\foo}{{\itshape bar}}
```

would *not* strip the braces.

It's important to remember that patching commands that have `@` in their name must *always* be performed between `\makeatletter` and `\makeatother`.

5 Examples

From `http://tex.stackexchange.com/a/42894`: the series of successive patches for changing the three occurrences of `\mathcode` in `\@addligto` into `\Umathcodenum` can become

```
\xpatchcmd*{\@addligto}{\mathcode}{\Umathcodenum}{}{}
```

while the code

```
\expandafter\patchcmd\csname mathligsoff \endcsname
  {\mathcode}{\Umathcodenum}{}{}
```

needed without regexpatch can become

```
\xpatchcmd\mathligsoff{\mathcode}{\Umathcodenum}{}{}
```

Another one: changing the space reserved for the theorem number in the 'List of theorems' provided by ntheorem could be obtained with etoolbox's \patchcmd by

```
\patchcmd{\thm@@thmline}{2.3em}{5em}{}{}
\patchcmd{\thm@@thmline@name}{2.3em}{5em}{}{}
\patchcmd{\thm@@thmline@noname}{2.3em}{5em}{}{}
```

if the hyperref option is not used, but a long series of patches would be needed with the option, as 2.3em appears three times in each macro. With regexpatch one can do independently of the option:

```
\xpatchcmd*{\thm@@thmline}{2.3em}{5em}{}{}
\xpatchcmd*{\thm@@thmline@name}{2.3em}{5em}{}{}
\xpatchcmd*{\thm@@thmline@noname}{2.3em}{5em}{}{}
```

A user asked how to patch the rubric environment in the 'CurVe' class in order to avoid the repetition of the rubric's title on continuation pages. The environment is based on longtable and the task is to remove the \endhead material, which is delimited by \endfirsthead and \endhead. Instead of

```
\patchcmd{\rubric}
  {\endfirsthead\@rubrichead{#1\@continuedname}\\*[\rubricspace]\endhead}
  {\endfirsthead}{}{}
```

one can more simply exploit regular expressions:

```
\makeatletter % the replacement text has @-commands
\regexpatchcmd{\rubric}
  { \c{endfirsthead} .* \c{endhead} }
  { \c{endfirsthead} }{}{}
\makeatother
```

Assume you want to insert a patch in the argument of a command; with the traditional method this is possible provided the patch text doesn't contain #. Here's an example

```
\makeatletter
\usepackage{etoolbox} % for \ifdef
\ifdef{\H@old@part}
  {
   \regexpatchcmd{\H@old@part}{$}{\c{gdef}\c{cont@name@part}\cB\{\cP\#2\cE\}}{}{}%
  }
  {
   \regexpatchcmd{\@part}{$}{\c{gdef}\c{cont@name@part}\cB\{\cP\#2\cE\}}{}{}%
  }
\show\H@old@part
\makeatother
```

We want to add \gdef\cont@name@part{#2} at the end of the replacement text, distinguishing when hyperref is loaded or not. So we patch the command by doing just what's requested. The example is a bit contrived, as using \ifdefined instead of the argument form wrapper would allow the traditional \apptocmd. However, other applications may be foreseen.

A problem raised on `comp.text.tex` in 2008 was to extract the number from the name of the file being typeset; the name was of the form `lecture15.tex` and the question was how to define a macro `\lecturenumber` that acted on `\jobname` to do its work. The obvious

```
\def\lecturenumber{\expandafter\extractnumber\jobname;}
\def\extractnumber lecture#1;{#1}
```

doesn't work because the characters produced by `\jobname` all have category code 12 (spaces have 10, as usual). A nifty solution was provided by David Kastrup:

```
\begingroup
\escapechar=-1
\expandafter\endgroup
\expandafter\def\expandafter\extractnumber\string\lecture#1;{#1}
```

Now with `\xpatchparametertext` one can do

```
\def\lecturenumber{\expandafter\extractnumber\jobname;}
\def\extractnumber lecture#1;{#1}
\xpatchparametertext\extractnumber{lecture}{lecture}{}{}
```

recalling that the substitution performed by l3regex uses category code 12 characters by default. The command can be generalized to accept any (fixed) prefix:

```
\def\setupfilenumber#1{%
  \def\filenumber{\expandafter\extractnumber\jobname;}%
  \def\extractnumber#1##1;{##1}%
  \xpatchparametertext\extractnumber{#1}{#1}{}{}}
\setupfilenumber{lecture}
```

where the prefix is passed to `\setupfilenumber` and the macro to use is `\filenumber`.

A proper LaTeX3 definition might be

```
\NewDocumentCommand{\setupfilenumber}{ m }
  {
   \group_begin:
   \cs_set:Npx \filenumber_aux:
     {
      \group_end:
      \cs_set:Npn \exp_not:N \filenumber_extract:w
        \tl_to_str:n { #1 } ####1 ; { ####1 }
     }
   \filenumber_aux:
  }
\NewDocumentCommand{\filenumber} {}
  {
   \exp_after:wN \filenumber_extract:w \c_job_name_tl ;
  }
```

6 The implementation of regexpatch

The usual starting stuff.

```
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\@ifpackagelater { expl3 } { 2012/01/19 }
  { }
  {
    \PackageError { regexpatch } { Support~package~l3kernel~too~old. }
      {
        Please~install~an~up~to~date~version~of~l3kernel~
        using~your~TeX~package~manager~or~from~CTAN.\\ \\
        Loading~regexpatch~will~abort!
      }
    \tex_endinput:D
  }
\RequirePackage{xparse,l3regex,etoolbox}
```

6.1 Variables

We define a bunch of variables: some booleans and token lists. The first tells us when the macro to patch has been defined by `\DeclareRobustCommand`, the second if it has an optional argument, the third if it's patchable, that is it can be reconstructed from its decomposition under the current category code regime. The last boolean is used for the tracing system: if true, messages about patching are issued.

```
\bool_new:N \l_xpatch_protect_bool
\bool_new:N \l_xpatch_optional_bool
\bool_new:N \l_xpatch_patchable_bool
\bool_new:N \l_xpatch_tracing_bool
```

The token list variables contain various items regarding the macro to patch: the name, the first level replacement text (we distinguish it from the 'real' replacement text), the prefixes, the argument spec and the 'real' replacement text.

```
\tl_new:N \l_xpatch_name_tl
\tl_new:N \l_xpatch_repl_tl
\tl_new:N \l_xpatch_prefix_tl
\tl_new:N \l_xpatch_arg_tl
\tl_new:N \l_xpatch_replacement_tl
\tl_new:N \l_xpatch_type_tl % for debugging messages
```

A variant for checking a regex match so that we can give the second argument as a token list.

```
\cs_generate_variant:Nn \regex_match:nnT {nVT}
```

6.2 Functions

The function `\xpatch_main_check:N` is responsible for telling us what kind of macro we're patching. Only one of the first four tests can be true; if none is, the macro is not 'special' and can be patched without doing anything particular to get its 'real name'. The check consists in matching with a suitable regex at the start of the replacement text (which is in detokenized form). If the macro passes one of the first two tests, it can still have an optional argument, so a supplementary test is needed.

Some technical remarks. Suppose we have the following definitions:

```
\DeclareRobustCommand{\xaa}[1]{xaa (DeclareRobustCommand-noopt)}
\DeclareRobustCommand{\xab}[1][x]{xab (DeclareRobustCommand-opt)}
\newcommand{\xac}[1][]{xac (newcommand-opt)}
\newrobustcmd\xad[1][]{xad (newrobustcmd-opt)}
\DeclareRobustCommand{\1}[1]{1 (DeclareRobustCommand-noopt)}
\DeclareRobustCommand{\2}[1][]{2 (DeclareRobustCommand-opt)}
\newcommand{\3}[1][]{3 (newcommand-opt)}
\newrobustcmd\4[1][]{4 (newrobustcmd-opt)}
```

Then the first level expansions are, respectively,

```
+\protect␣\xaa␣␣+
+\protect␣\xab␣␣+
+\@protected@testopt␣\xac␣\\xac␣{}+
+\@testopt␣\\xad␣{}+
+\x@protect␣\1\protect␣\1␣␣+
+\x@protect␣\2\protect␣\2␣␣+
+\@protected@testopt␣\3\\3␣{}+
+\@testopt␣\\4␣{}+
```

where the + is used to delimit the expansions and show the spaces. Remember that `\show` always adds a space after a control word, but not after a control symbol such as `\1`. However, in lines 5 and 6, `\1␣` is not a control symbol any more. So we have to take care of `\protect`, `\x@protect`, `\@protected@testopt` and `\@testopt`. But it's not simply sufficient to check for the presence of such a token at the start of the replacement text, or we'll be confused by macros such as `\linebreak`, whose replacement text starts with `\@testopt`. So we'll check also for the presence of the subsequent tokens, that depend on the macro's name. If the macro is recognized to have an optional argument, its default value is stored in `\tl_xpatch_repl_tl` (that we wouldn't use any more) to be shown by `\xshowcmd*` or when the tracing system is active: we throw away everything except what's contained between the final pair of braces.

```
\cs_new:Npn \xpatch_main_check:N #1
  {
   \bool_set_false:N \l_xpatch_protect_bool
   \bool_set_false:N \l_xpatch_optional_bool
   \tl_set:Nx \l_xpatch_name_tl { \cs_to_str:N #1  }
   \tl_set:Nx \l_xpatch_repl_tl { \token_get_replacement_spec:N #1 }
   \tl_clear:N \l_xpatch_type_tl
   \regex_match:nVT % \DeclareRobustCommand<control word>
     {^\\protect\ \\\u{l_xpatch_name_tl}\ \ }
     \l_xpatch_repl_tl
     {
      \bool_set_true:N \l_xpatch_protect_bool
      \tl_put_right:Nx \l_xpatch_name_tl { \c_space_tl }
      \tl_set:Nn \l_xpatch_type_tl { DRCw }
     }
   \regex_match:nVT % \DeclareRobustCommand<control symbol>
     {^\\x@protect\ \\\u{l_xpatch_name_tl}\\}
     \l_xpatch_repl_tl
     {
      \bool_set_true:N \l_xpatch_protect_bool
      \tl_put_right:Nx \l_xpatch_name_tl { \c_space_tl }
      \tl_set:Nn \l_xpatch_type_tl { DRCs }
```

```
    }
  \regex_match:nVT % \newcommand<control word> with opt arg
    {^\\@protected@testopt\ \\\u{l_xpatch_name_tl}\ \\\\}
    \l_xpatch_repl_tl
    {
     \bool_set_true:N \l_xpatch_optional_bool
     \tl_put_left:Nx \l_xpatch_name_tl { \c_backslash_str }
     \tl_set:Nn \l_xpatch_type_tl { ncw+o }
    }
  \regex_match:nVT % \newcommand<control symbol> with opt arg
    {^\\@protected@testopt\ \\\u{l_xpatch_name_tl}\\\\}
    \l_xpatch_repl_tl
    {
     \bool_set_true:N \l_xpatch_optional_bool
     \tl_put_left:Nx \l_xpatch_name_tl { \c_backslash_str }
     \tl_set:Nn \l_xpatch_type_tl { ncs+o }
    }
  \regex_match:nVT % \newrobustcmd<any cs> with opt arg
    {^\\@testopt\ \\\\\u{l_xpatch_name_tl}}
    \l_xpatch_repl_tl
    {
     \bool_set_true:N \l_xpatch_optional_bool
     \tl_put_left:Nx \l_xpatch_name_tl { \c_backslash_str }
     \tl_set:Nn \l_xpatch_type_tl { nrc+o }
    }
  \bool_if:NT \l_xpatch_protect_bool
    {
     \tl_set:Nx \l_xpatch_repl_tl
       { \exp_after:wN \token_get_replacement_spec:N
           \cs:w \l_xpatch_name_tl \cs_end: }
     \regex_match:nVT % \DeclareRobustCommand<any cs> with opt arg
       {^\\@protected@testopt\ \\\u{l_xpatch_name_tl}\ \\\\}
       \l_xpatch_repl_tl
       {
        \bool_set_true:N \l_xpatch_optional_bool
        \tl_put_left:Nx \l_xpatch_name_tl { \c_backslash_str }
        \tl_put_right:Nn \l_xpatch_type_tl { +o }
       }
    }
  \bool_if:NT \l_xpatch_optional_bool
    {
     \regex_replace_once:nnN { .*? \{ (.*) \} \Z } { \1 }
       \l_xpatch_repl_tl
    }
}
```

We use the information gathered with `\xpatch_main_check:N` to perform the patch; the macro to patch is `#3`, the function to execute is `#1`; `#2` is a function that gobbles the next items in case the macro's name is misspelled.

```
\cs_new:Npn \xpatch_main:NNN #1 #2 #3
 {
  \cs_if_exist:NTF #3
    {
     \xpatch_main_check:N #3
```

```
      \bool_if:NT \l_xpatch_tracing_bool
        { \xpatch_message_cstype:n #3 }
      \exp_after:wN #1 \cs:w \l_xpatch_name_tl \cs_end:
     }
     {
      \msg_term:x
        {
         xpatch~message \\
        `\token_to_str:N #3'~is~undefined;~
        I'll~ignore~the~request.
        }
      #2
     }
  }
```

Now we define the patching functions. We get all the parts in which a macro can be split: prefixes, parameter text and replacement text; the name is already available. The token lists `\l_xpatch_X_tl` will contain the prefix or parameter text or replacement text of `#1` first in 'detokenized' and then in 'tokenized' form.

```
\cs_new:Npn \xpatch_get_all:N #1
  {
   \exp_args:NNf \tl_set:Nn \l_xpatch_prefix_tl
      {\token_get_prefix_spec:N #1 }
   \exp_args:NNnx \tl_set_rescan:Nnn
     \l_xpatch_prefix_tl { } \l_xpatch_prefix_tl
   \exp_args:NNf \tl_set:Nn \l_xpatch_arg_tl
      {\token_get_arg_spec:N #1 }
   \exp_args:NNnx \tl_set_rescan:Nnn
     \l_xpatch_arg_tl { } \l_xpatch_arg_tl
   \exp_args:NNf \tl_set:Nn \l_xpatch_replacement_tl
      {\token_get_replacement_spec:N #1 }
   \exp_args:NNnx \tl_set_rescan:Nnn
     \l_xpatch_replacement_tl { } \l_xpatch_replacement_tl
  }
```

After possible modifications to the replacement text, we can call `\xpatch_rebuild:N` to redo the definition of `#1`; we can also use it for checking if `#1` is patchable. Of course we need to use `\tex_def:D` at this point.

```
\cs_new:Npn \xpatch_rebuild:N #1
  {
   \tl_clear:N \l_tmpa_tl
   \tl_put_right:NV \l_tmpa_tl \l_xpatch_prefix_tl
   \tl_put_right:Nn \l_tmpa_tl { \tex_def:D #1 }
   \tl_put_right:NV \l_tmpa_tl \l_xpatch_arg_tl
   \tl_put_right:No \l_tmpa_tl { \exp_after:wN {\l_xpatch_replacement_tl} }
   \tl_use:N \l_tmpa_tl
  }
```

To check if `#1` is patchable, we rebuild it as `\xpatch_tmpa:w` and look whether `#1` and `\xpatch_tmpa:w` are the same. This is always the first thing to do, so we put `\xpatch_get_all:N` here; `#1` is the macro to patch.

```
\cs_new:Npn \xpatch_check_patchable:N #1
  {
   \cs_if_exist:NTF #1
     {
```

```
      \xpatch_get_all:N #1
      \xpatch_rebuild:N \xpatch_tmpa:w
      \cs_if_eq:NNTF #1 \xpatch_tmpa:w
        {
         \bool_set_true:N \l_xpatch_patchable_bool
         \xpatch_message:n
           {
            Macro `\token_to_str:N #1'~is~patchable
           }
        }
        {
         \bool_set_false:N \l_xpatch_patchable_bool
         \xpatch_message:n
           {
            Macro `\token_to_str:N #1'~is~NOT~patchable\\
            (Check~if~it~contains~`@'~commands)
           }
        }
    }
    {
     \bool_set_false:N \l_xpatch_patchable_bool
     \xpatch_message:n
       {
        Macro `\token_to_str:N #1'~doesn't~exist.
       }
    }
 }
```

Defining the internal versions of `\xpretocmd` and `\xapptocmd` is easy: we check if the command is patchable and, if so, we prepend or append the second argument to the replacement text and rebuild the macro.

```
\cs_new:Npn \xpatch_pretocmd:Nnnn #1 #2 #3 #4
 {
  \xpatch_check_patchable:N #1
  \bool_if:NTF \l_xpatch_patchable_bool
     {
      \tl_put_left:Nn \l_xpatch_replacement_tl { #2 }
      \xpatch_rebuild:N #1
      #3
     }
     {
      #4
     }
 }
\cs_new:Npn \xpatch_apptocmd:Nnnn #1 #2 #3 #4
 {
  \xpatch_check_patchable:N #1
  \bool_if:NTF \l_xpatch_patchable_bool
     {
      \tl_put_right:Nn \l_xpatch_replacement_tl { #2 }
      \xpatch_rebuild:N #1
      #3
     }
     {
```

```
      #4
     }
  }
```

Substituting tokens in the replacement text is a bit harder, but not conceptually different. First the internal version of `\regexpatchcmd(*)`: check if `#1` is patchable, do the replacement if possible; beware that characters in the replacement string are of category 12 by default.

```
\cs_new:Npn \xpatch_regexpatchcmd_all:Nnnnn #1 #2 #3 #4 #5
  {
   \xpatch_check_patchable:N #1
   \bool_if:NTF \l_xpatch_patchable_bool
      {
       \regex_replace_all:nnN { #2 } { #3 } \l_xpatch_replacement_tl
       \xpatch_rebuild:N #1
       #4
      }
      {
       #5
      }
  }
\cs_new:Npn \xpatch_regexpatchcmd_once:Nnnnn #1 #2 #3 #4 #5
  {
   \xpatch_check_patchable:N #1
   \bool_if:NTF \l_xpatch_patchable_bool
      {
       \regex_replace_once:nnN { #2 } { #3 } \l_xpatch_replacement_tl
       \xpatch_rebuild:N #1
       #4
      }
      {
       #5
      }
  }
```

Thanks to the features of l3regex, we can also implement directly the analog of `\patchcmd`, but also with a 'replace all' version.

```
\cs_new:Npn \xpatch_patchcmd_once:Nnnnn #1 #2 #3 #4 #5
  {
   \xpatch_check_patchable:N #1
   \bool_if:NTF \l_xpatch_patchable_bool
      {
       \tl_set:Nn \l_tmpa_tl { #2 }
       \tl_set:Nn \l_tmpb_tl { #3 }
       \regex_replace_once:nnN
         { \u{l_tmpa_tl} }
         { \u{l_tmpb_tl} }
         \l_xpatch_replacement_tl
       \xpatch_rebuild:N #1
       #4
      }
      {
       #5
      }
  }
```

```
\cs_new:Npn \xpatch_patchcmd_all:Nnnnn #1 #2 #3 #4 #5
  {
   \xpatch_check_patchable:N #1
   \bool_if:NTF \l_xpatch_patchable_bool
      {
       \tl_set:Nn \l_tmpa_tl { #2 }
       \tl_set:Nn \l_tmpb_tl { #3 }
       \regex_replace_all:nnN
         { \u{l_tmpa_tl} }
         { \u{l_tmpb_tl} }
         \l_xpatch_replacement_tl
       \xpatch_rebuild:N #1
       #4
      }
      {
       #5
      }
  }
```

Now the tracing system.

```
\cs_new:Npn \xpatch_message:n #1
  {
   \bool_if:NT \l_xpatch_tracing_bool
     {
      \msg_term:x { xpatch~message \\ #1 }
     }
  }
\cs_new:Npn \xpatch_message_cstype:n #1
  {
   \prg_case_str:onn { \l_xpatch_type_tl }
     {
      { DRCw }   {
                  \xpatch_message:n
                    {
                     `\token_to_str:N #1'~is~a~control~word~defined~
                     with~\token_to_str:N \DeclareRobustCommand
                    }
                 }
      { DRCw+o } {
                  \xpatch_message:n
                    {
                     `\token_to_str:N #1'~is~a~control~word~defined~
                     with~`\token_to_str:N \DeclareRobustCommand'~
                     and~a~default~optional~argument~`\l_xpatch_repl_tl'
                    }
                 }
      { DRCs }   {
                  \xpatch_message:n
                    {
                     `\token_to_str:N #1'~is~a~control~symbol~defined~
                     with~`\token_to_str:N \DeclareRobustCommand'
                    }
                 }
      { DRCs+o } {
                  \xpatch_message:n
```

```
                    {
                     `\token_to_str:N #1'~is~a~control~symbol~defined~
                     with~`\token_to_str:N \DeclareRobustCommand'~
                     and~a~default~optional~argument~`\l_xpatch_repl_tl'
                    }
                }
      { ncw+o }  {
                  \xpatch_message:n
                    {
                     `\token_to_str:N #1'~is~a~control~word~defined~
                     with~`\token_to_str:N \newcommand'~
                     and~a~default~optional~argument~`\l_xpatch_repl_tl'
                    }
                }
      { ncs+o }  {
                  \xpatch_message:n
                    {
                     `\token_to_str:N #1'~is~a~control~symbol~defined~
                     with~`\token_to_str:N \newcommand'~
                     and~a~default~optional~argument~`\l_xpatch_repl_tl'
                    }
                }
      { nrc+o }  {
                  \xpatch_message:n
                    {
                     `\token_to_str:N #1'~is~a~control~sequence~defined~
                     with~`\token_to_str:N \newrobustcmd'~
                     and~a~default~optional~argument~`\l_xpatch_repl_tl'
                    }
                }
      }
      {
       \xpatch_message:n
         {
          `\token_to_str:N #1'~is~not~especially~defined
         }
      }
  }
```

6.3 The user level functions

Here are the functions for patching usual macros; the *-variants for `\xpatchcmd` do a 'replace all'.

```
\NewDocumentCommand{\xshowcmd} { s m }
  {
   \IfBooleanT{#1}
     {
      \group_begin:
      \bool_set_true:N \l_xpatch_tracing_bool
     }
   \xpatch_main:NNN \cs_show:N \prg_do_nothing: #2
   \IfBooleanT{#1}
     {
```

```
      \group_end:
    }
  }
\NewDocumentCommand{\xpretocmd}{ }
  { \xpatch_main:NNN \xpatch_pretocmd:Nnnn \use_none:nn }
\NewDocumentCommand{\xapptocmd}{ }
  { \xpatch_main:NNN \xpatch_apptocmd:Nnnn \use_none:nn }
\NewDocumentCommand{\regexpatchcmd}{ s }
  {
   \IfBooleanTF{#1}
     { \xpatch_main:NNN \xpatch_regexpatchcmd_all:Nnnnn \use_none:nnn }
     { \xpatch_main:NNN \xpatch_regexpatchcmd_once:Nnnnn \use_none:nnn }
  }
\NewDocumentCommand{\xpatchcmd}{ s }
  {
   \IfBooleanTF{#1}
     { \xpatch_main:NNN \xpatch_patchcmd_all:Nnnnn \use_none:nn }
     { \xpatch_main:NNN \xpatch_patchcmd_once:Nnnnn \use_none:nn }
  }
```

The functions for patching biblatex related macros that are given by name. We need a variant of `\xpatch_main:NN`.

```
\cs_generate_variant:Nn \xpatch_main:NNN {NNc}
\NewDocumentCommand{\xshowbibmacro} { s m }
  {
   \IfBooleanT{#1}
     {
      \group_begin:
      \bool_set_true:N \l_xpatch_tracing_bool
     }
   \xpatch_main:NNc \cs_show:N \prg_do_nothing: { abx@macro@ \tl_to_str:n {#2} }
   \IfBooleanT{#1}
     {
      \group_end:
     }
  }
\NewDocumentCommand{\xpretobibmacro} { m m m }
  {
   \xpatch_main:NNc \xpatch_pretocmd:Nnnn \use_none:nn
     { abx@macro@ \tl_to_str:n {#1} }
  }
\NewDocumentCommand{\xapptobibmacro} { m m m }
  {
   \xpatch_main:NNc \xpatch_apptocmd:Nnnn \use_none:nn
     { abx@macro@ \tl_to_str:n {#1} }
  }
\NewDocumentCommand{\xpatchbibmacro} { s m m m }
  {
   \IfBooleanTF{#1}
     {
      \xpatch_main:NcN \xpatch_patchcmd_all:Nnnnn \use_none:nnn
        { abx@macro@ \tl_to_str:n {#2} }
     }
     {
```

```
      \xpatch_main:NcN \xpatch_patchcmd_once:Nnnnn \use_none:nnn
       { abx@macro@ \tl_to_str:n {#2} }
    }
  }
\NewDocumentCommand{\regexpatchbibmacro} { s m m m }
  {
   \IfBooleanTF{#1}
     {
      \xpatch_main:NcN \xpatch_regexpatchcmd_all:Nnnnn \use_none:nnn
        { abx@macro@ \tl_to_str:n {#2} }
     }
     {
      \xpatch_main:NcN \xpatch_regexpatchcmd_once:Nnnnn \use_none:nnn
        { abx@macro@ \tl_to_str:n {#2} }
     }
  }
\NewDocumentCommand{\xshowbibdriver} { s m }
  {
   \IfBooleanT{#1}
     {
      \group_begin:
      \bool_set_true:N \l_xpatch_tracing_bool
     }
   \xpatch_main:NNc \cs_show:N \prg_do_nothing: { blx@bbx@#2 }
   \IfBooleanT{#1}
     {
      \group_end:
     }
  }
\NewDocumentCommand{\xpretobibdriver} { m }
  { \exp_args:Nc \xpatch_pretocmd:Nnnn {blx@bbx@#1} }
\NewDocumentCommand{\xapptobibdriver} { m }
  { \exp_args:Nc \xpatch_apptocmd:Nnnn {blx@bbx@#1} }
\NewDocumentCommand{\xpatchbibdriver} { s m }
  {
   \IfBooleanTF{#1}
     { \exp_args:Nc \xpatch_patchcmd_all:Nnnnn {blx@bbx@#2} }
     { \exp_args:Nc \xpatch_patchcmd_once:Nnnnn {blx@bbx@#2} }
  }
\NewDocumentCommand{\regexpatchbibdriver} { s m }
  {
   \IfBooleanTF{#1}
     { \exp_args:Nc \xpatch_regexpatchcmd_all:Nnnnn {blx@bbx@#2} }
     { \exp_args:Nc \xpatch_regexpatchcmd_once:Nnnnn {blx@bbx@#2} }
  }
```

A macro to check if the macro is patchable. It just prints a message on the terminal and in the log file.

```
\NewDocumentCommand{\checkpatchable}{ m }
  {
   \group_begin:
   \bool_set_true:N \l_xpatch_tracing_bool
   \xpatch_check_patchable:N #1
   \group_end:
```

```
}
```

The last user level command: a macro for changing the optional argument in a macro that has one.

```
\cs_generate_variant:Nn \xpatch_get_all:N {c}
\cs_generate_variant:Nn \xpatch_rebuild:N {c}
\NewDocumentCommand{\xpatchoptarg}{ m m }
  {
   \xpatch_main_check:N #1
   \bool_if:NTF \l_xpatch_optional_bool
     {
```

We have a macro with optional argument; so we strip off the first backslash from the name and proceed.

```
       \tl_set:Nx \l_xpatch_name_tl { \tl_tail:V \l_xpatch_name_tl }
```

Gather the prefix (it is `\protected` when `#1` has been defined with `\newrobustcmd`).

```
       \exp_args:NNf \tl_set:Nn \l_xpatch_prefix_tl
         {\token_get_prefix_spec:N #1 }
       \tl_clear:N \l_xpatch_prefix_tl
       \exp_args:NNnx \tl_set_rescan:Nnn
          \l_xpatch_prefix_tl { } \l_xpatch_prefix_tl
```

Get the replacement text in tokenized form: the control sequences have spaces in their names, so we can't rely on `\token_get_replacement_spec:N` because the spaces would be lost.

```
       \exp_args:NNV \tl_set_eq:Nc \l_xpatch_replacement_tl
         { \l_xpatch_name_tl }
```

Now we have to change the last item in the token list: we just store the new optional argument in a token list variable and do a regex substitution, based on the fact that the replacement text consists of control sequences, an open brace, the optional argument and a closed brace, so we anchor at the end of the token list.

```
       \tl_set:Nn \l_tmpa_tl { { #2 } }
       \regex_replace_once:nnN { \cB. .* \cE. \Z} { \u{l_tmpa_tl} }
         \l_xpatch_replacement_tl
```

Now we rebuild the control sequence.

```
       \xpatch_rebuild:c { \l_xpatch_name_tl }
       }
```

If the macro hasn't an optional argument we issue a message.

```
     {
      \group_begin:
      \bool_set_true:N \l_xpatch_tracing_bool
      \xpatch_message:n
        {
         Macro~`\token_to_str:N #1'~ has~no~optional~argument~
         or~it~has~been~defined~with~`xparse'~and~operating~
         on~such~commands~is~(still)~not~supported
        }
      \group_end:
     }
}
```

Just one more thing: enabling or disabling the tracing system.

```
\NewDocumentCommand{\tracingxpatches}{ O{1} }
  {
   \int_compare:nTF { #1 > 0 }
     { \bool_set_true:N \l_xpatch_tracing_bool }
     { \bool_set_false:N \l_xpatch_tracing_bool }
  }
```

One more thing: patching the parameter text!

```
\NewDocumentCommand{\xpatchparametertext}{m m m m m}
  {
   \xpatch_check_patchable:N #1
   \bool_if:NTF \l_xpatch_patchable_bool
      {
       \regex_replace_once:nnN { #2 } { #3 } \l_xpatch_arg_tl
       \xpatch_rebuild:N #1
       #4
      }
      {
       #5
      }
  }
```

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.

G

I

L

M

N

P

R

T

U

X

Z