Macros

Macros with optional arguments

Victor Eijkhout

Users of LaTeX are familiar with macros that have optional arguments, such as `\newcommand`.

```
\newcommand\testa{ ... }
\newcommand\testb[2]{ ... }
```

Here, the second argument is optional; its inclusion alters the workings of `\newcommand`. Wouldn't it be nice to be able to write such macros yourself?

Let's set ourselves the project, for now, of writing a macro with one optional and one required argument. If the optional, first, argument is omitted, the value of the second, required one should be used. That is, our macro, which we shall call `\aa`, should have the following behaviour: the input

```
\aa [1]2
\aa 2
```

should give the output

```
Opt: [1] Req: [2]
Opt: [2] Req: [2]
```

The crux to these optional arguments is a test for the occurrence of the opening square bracket; that is, somehow we need to peek at what follows the macro. For this, TeX has the `\futurelet` command. The example

```
\futurelet\x\y z
```

has the effect of

```
\let\x z\y
```

That is, the first argument `\x` is '`\let`' to whatever follows the second argument `\y`, and then the second argument is executed. Why does this help us? Well, we can now `\futurelet` the token after `\aa` to, say, `\next`, and then call a macro that investigates whether `\next` is a square bracket, and acts accordingly. The 'acting accordingly' part means calling one or the other of two macros, the one handling the case where there was a square bracket, the other the case where there wasn't.

```
\def\aa{\futurelet\next\aaX}
\def\aaX{%
  \ifx[\next \expandafter\aaXX
  \else \expandafter\aaXXX \fi}
```

We now define two separate macros, one with and one without optional argument.

```
\def\aaXX[#1]#2{
  Opt: [#1] Req: [#2]\par}
\def\aaXXX#1{
  Opt: [#1] Req: [#1]\par}
```

Since we decided that calling the macro without the optional argument would have the effect of doubling the required argument — something that happens frequently in the internals of LaTeX — we can write

```
\def\aaXXX#1{\aaXX[#1]{#1}}
```

and save ourselves some code duplication.

If, instead of duplicating the first required argument, we wanted to use some default value for the omitted optional argument, we would write

```
\def\aaXXX#1{\aaXX[<default>]{#1}}
```

Just one remark. Note that until the end, where we call `\aaXX`, we never look at the other arguments, and we do not care how many of them there are. In calls such as `\aa [1]234...` we repeatedly replace the `\aa` control sequences by other control sequences. Only the final macros `\aaXXX` — in the case of an optional argument — and `\aaXX` — in the case of none — touch the actual arguments.

Now let's consider the case where the optional argument is not the first but the second. Say, we want a macro `\bb` that, called as

```
\bb 1[2]3
\bb 13
```

gives

```
One: [1] Opt: [2] Req: [3]
One: [1] Opt: [3] Req: [3]
```

The trick here is to scoop up the first argument and store it away:

```
\def\bb#1{\def\savedargone{{#1}}%
  \futurelet\next\bbX}
```

After that, we proceed for a while as before

```
\def\bbX{%
  \ifx[\next \expandafter\bbXX
  \else \expandafter\bbXXX \fi}
\def\bbXXX#1{\bbXX[#1]{#1}}
```

and then we insert the saved argument in between the final macro and the the remaining arguments:

```
\def\bbXX{\expandafter\bbY\savedargone}
\def\bbY#1[#2]#3{
  One: [#1] Opt: [#2] Req: [#3]\par}
```

The `\expandafter` in `\bbXX` turns the sequence

```
\bbXX <arg 2><arg 3>
```

which first becomes

```
\expandafter\bbY\savedargone
                <arg 2><arg 3>
```

into

```
\bb <arg1><arg 2><arg 3>
```

Well, there you have it. All the ingredients for writing macros with optional arguments.

So why isn't this article over? Well, after writing macros like this becomes a second nature to you, you might start wondering if there isn't a way to automate this rather repetitive process. And of course there is. But it is a bit of work. In fact, this may well be the most mind-bending macro I have ever written.

A small device to save us some typing:

```
\let\expa\expandafter
\let\noex\noexpand
```

We now set ourselves the goal of writing a macro `\defoptargcomm` — 'define an optional argument command' — that allows us to write

```
\defoptargcomm\def\aa[#1]#2{%
  Opt: [#1] Req: [#2]\par}
```

so that again, as above,

```
\aa [1]2
\aa 2
```

gives the right result.

I will give the macro in increments. First of all, the name of the macro to be defined is changed from a control sequence into a string of characters, so that we can base other macro names on it:

```
\def\defoptargcomm#1#2{%
  \edef\bnon{\stringcsnoescape#2}%
```

The auxiliary macro `\stringcsnoescape` is given below, as will be all further auxiliaries.

Next we define the macro that peeks at a possible square bracket. To get the effect of

```
\def\aa{\futurelet\next\aaX}
```

we write (where `\nxarg` and `\nxname` are auxiliaries; see below)

```
  \edef\anon{\nxarg#1{\bnon}{%
    \futurelet\noex\next
    \nxname{\bnon X}}}\anon
```

This is a good trick: since we will have to form some new control sequences, we build the define statement inside the `\edef` of an otherwise unimportant macro. Calling this macro will then execute the definition. (The auxiliary `\nxname` serves to build a control sequence and further prevent it from being expanded. See the end of this article.)

We use this trick again, this time to define the macro that will decide, based on the presence or not of a square bracket, which further macro to call. For

```
\def\aaX{%
  \ifx[\next \expa\aaXX
  \else \expa\aaXXX}\fi}
```

we write

```
  \edef\anon{\nxarg#1{\bnon X}{%
    \noex\ifx[\noex\next
      \noex\expa
        \nxname{\bnon XX}%
    \noex\else
      \noex\expa
        \nxname{\bnon XXX}%
    \noex\fi}}\anon
```

Here is the macro that duplicates the first argument if there is no optional argument. For the equivalent of

```
\def\aaXXX#1{\aaXX[#1]{#1}}
```

we write

```
  \edef\anon{%
   \nxarg#1{\bnon XXX}####1{%
    \nxname{\bnon XX}[####1]{####1}}%
            }\anon
```

And now we would have to do the actual definition of the macro with optional argument, as we did in the first part of the article. However, we can skip this, as the definition was already in the input stream, so we conclude the definition of `\defoptargcomm` with the `\def` control sequence (argument `#1`) and the name with two Xs attached. For

```
\def\aaXX   ...
  % .. after this come the parameters
  % .. and definition
```

we conclude with

```
  \arg#1{\bnon XX}}
```

Phew.

What? You want more?

Well, the assumption that an omitted optional argument should take on the value of the first present argument is a bit limiting. You may want it to take on some default value. For instance, the syntax

```
\defoptargcomm[4]\def\bb[#1]#2{%
  Opt: [#1] Req: [#2]\par}
```

would mean that the value taken in absence of an optional argument is '4'. The input

```
\bb [1]2

\bb 2
```

then gives

```
Opt: [1] Req: [2]
Opt: [2] Req: [2]
```

That is not very hard to do: we need yet another application of `\futurelet`.

```
\def\defoptargcomm{%
  \futurelet\next\defoptargcommX}
```

```
\def\defoptargcommX{%
  \ifx[\next
    \expandafter\defoptargcommXX
  \else \def\optarg{[########1]}%
    \expandafter\defoptargcommXXX
  \fi}
\def\defoptargcommXX[#1]{%
 \def\optarg{[#1]}\defoptargcommXXX}
```

We are thus saving the value of the optional argument in a control sequence `\optarg`. Note the sequence of eight hash characters, which I will not further explain[1].

Now the macro `\defoptargcommXXX` has to use the value of `\optarg`. For this we change only a small part. For the equivalent of

```
\def\bbXXX#1{%
  \bbXX[optarg]{#1}}
```

```
  \edef\anon{%
   \nxarg#1{\bnon XXX}####1{%
     \nxname{\bnon XX}\optarg{####1}}%
           }\anon
```

Tada!

Now, if you've followed this exposition carefully, you'll have noticed that this ultra-powerful macro can still not do something that we could do by hand: let any argument be optional, not just the first. Let us say that we want to write

```
\PACdefoptargncomm3%
 \def\cc#1#2[#3]#4{First: [#1,#2]
    Opt: [#3] Req: [#4]\par}
\cc 12[3]4
\cc 124
```

and get

```
First: [1,2] Opt: [3] Req: [4]
First: [1,2] Opt: [4] Req: [4]
```

Deep breath. Here comes the final version of our macro for defining macros with optional arguments.

I will explain this one bottom-up, instead of top-down. Our first problem is that we need to get the first couple of fixed arguments out of the way before we can look at the optional argument. Suppose we have a macro `\firstarg` that expands to the arguments before the optional one, in this case `#1#2`, and a macro `\savedarg` with the same, but in braces: `{#1}{#2}`, then to get the equivalent of

```
\def\cc#1#2{\def\ccsaved{{#1}{#2}}%
    \futurelet\next\ccX}
```

we write — and compare this with the above —

```
 \edef\anon{\nxarg#1{\bnon}\firstarg
     {\def\nxname{\bnon saved}%
           {\savedarg}%
      \futurelet\noex\next
      \nxname{\bnon X}}}\anon
```

Now that we have the first arguments set aside, we can look for the square bracket. If is is not there, we have to call a macro that duplicates the next argument; if is is there, we re-insert the saved arguments, and call the final macro. This would read

```
\expa\ccXX\ccsaved
```

but because it occurs in a conditional it becomes

```
\expa\expa\expa\ccXX\expa\ccsaved
            \else
```

and because it happens inside an `\edef` there are `\noexpand`s interspersed everywhere:

```
  \edef\anon{\nxarg#1{\bnon X}{%
    \noex\if[\noex\next
      \noex\expa\noex\expa\noex\expa
        \nxname{\bnon XX}%
          \noex\expa\nxname
                {\bnon saved}%
    \noex\else
      \noex\expa\nxname{\bnon XXX}%
    \noex\fi}}\anon
```

By comparison, the macro to duplicate the argument after the omitted optional argument is child's play:

```
  \edef\anon{%
     \nxarg#1{\bnon XXX}####1{%
      \noex\expa\nxname{\bnon XX}%
       \nxname{\bnon saved}%
        [\optarg]{####1}}}\anon
```

Note the `\optarg`, which contains either the tokens `####1`, or a default value.

All this is inside a macro

```
\def\defoptargcommXXX#1#2{%
  \def\protect{}%
  \edef\bnon{\stringcsnoescape#2}%
  < ... the above ... >
  \arg#1{\bnon XX}}
```

This is the macro that handles the explicit default value:

```
\def\defoptargcommXX[#1]{%
  \def\optarg{#1}\defoptargcommXXX}
```

[1] Okay, just a little bit then. If TEX sees one hash character followed by a letter — which can only happen in a macro — it replaces it by the corresponding macro argument. Two hash characters in a row are replaced by a single, which is further left untouched. Unless, that is, it is scanned again. Since in the end the sequence here will be scanned three times we need to write eight hash characters in order to get #1 in the input stream.

We're getting close to the interesting bits. This the macro that tests for a default value:

```
\def\defoptargcommX{%
 \ifx[\next
   \expandafter\defoptargcommXX
 \else
  \edef\anon
   {\def\noex\optarg{\protect\hash1}%
   }\anon
  \expandafter\PACdefoptargcommXXX
 \fi}
```

Note the occurrence of a macro `\hash`, which we will define in a minute. Here is the old macro for an optional first argument:

```
\def\PACdefoptargcomm{%
  \def\PACfirstarg{}\def\PACsavedarg{}%
  \def\protect{\noex\protect\noex}%
  \def\hash{########}%
  \futurelet\next\defoptargcommX}
```

This is the nasty one: the macro that accepts the location of the optional argument and builds the `\firstarg`, `\savedarg` macros. We use two token lists, which gradually get build inside a loop.

```
\def\defoptargncomm#1{%
 \toksa={}\toksb={}\counta=#1\relax
 \def\protect{\noex\protect\noex}%
 \def\hash{########}%
 {\count1=1 \count2=#1
  \loop
   \edef\PACanon{
     \global\PACtoksa={\the\PACtoksa
       \protect\hash\number\count1}%
     \global\PACtoksb={\the\PACtoksb
      {\protect\hash\number\count1}}%
                }\PACanon
   \advance\count1 by 1\relax
  \ifnum\count1<\count2 \repeat
 }%
 \edef\anon{\def\noex\firstarg
             {\the\toksa}}\PACanon
 \edef\anon{\def\noex\savedarg
             {\the\PACtoksb}}\PACanon
 \futurelet\next\defoptargcommX}
```

And that's it. You can get this monster from CTAN as `PAC_utils.tex`; you also need `CS_auxs.tex`.

Finally, here are the auxiliary macros. I will leave out mentioning various conditions on the functioning of these macros; normally they will be satisfied. To convert a control sequence to a string of characters, purely by expansion:

```
\let\expa\expandafter
\let\noex\noexpand
\def\stringcsnoescape#1{%
  \expa\gobbleescape\string#1}
{\escapechar-1
\expa\expa\expa\gdef
  \expa\expa\expa\CSgobblearrow
    \expa\string
      \csname macro:->\endcsname{}
}
\def\gobbleescape#1{%
  \ifnum`\\=`#1 \else #1\fi}
```

Here are various macros to build a control sequence out of a string of characters, and subsequently to protect the control sequence from further expansion:

```
\def\name#1{\csname#1\endcsname}
\def\arg#1#2{%
  \expa#1\csname#2\endcsname}
\def\csarg#1#2{%
  \name{#1\expa}\csname#2\endcsname}
\def\nxarg#1#2{%
  \expa#1\expa\noex
    \csname#2\endcsname}
\def\nxname#1{%
  \expa\noex\csname#1\endcsname}
```

◇ Victor Eijkhout
Computer Science Department
University of Tennessee
Knoxville, TN 37996-1301 U.S.A.
victor@eijkhout.net