dowith.sty

—

Apply Command to Elements of Lists without Separators — and without Iterator*

Uwe Lück†

May 10, 2012

Abstract

This package provides macros for applying a command to all elements of a list without separators, such as `\DoWithAllIn{`⟨*cmd*⟩`}{`⟨*list-macro*⟩`}`, and also for extending and reducing macros storing such lists. Applications in mind belonged to LaTeX, but the package should work with other formats as well. Loop and list macros in other packages are discussed. There is an emphasis on expandability (no iterator), without relying on ε-TeX.

Related packages: etextools, etoolbox, forarray, forloop, multido, moredefs, lmake, texapi, xfor, xspace

Keywords: macro programming, loops, list macros

Contents

*This document describes version v0.2 of dowith.sty as of 2011/11/19.

†`http://contact-ednotes.sty.de.vu`

1 Usage

The file dowith.sty is provided ready, installation only requires putting it somewhere where TEX finds it (which may need updating the filename data base).[1]

With LATEX, you load dowith.sty (as usually) by

```
\usepackage{dowith}
```

below the `\documentclass` line(s) and above `\begin{document}`. However, the package can also be used with other formats, just

```
\input dowith.sty
```

The single commands that the package provides are described below together with their implementation.

2 Similar Commands in other Packages

The ε-TeX-related packages etextools (Florent Chervet), etoolbox (Philipp Lehman), and texapi (Paul Isambert) seem to include and (very much) extend the functionality of dowith. Also the `\ForEach...` macros of forarray (Christian Schröppel) seem to extend the present `\DoWith...` commands. moredefs (Matt Swift) provides list handling commands like the few that are here.[2] (I do not want to load that much.)

Regarding LATEX macros in `latex.ltx`, the basic macro `\DoWith` of the present package resembles `\@tfor` very much, which likewise deals with lists without separators. By contrast, LATEX's `\@for` deals with comma-separated lists (such as lists of options).

A major difference between `\DoWith` and `\@tfor` is that the latter uses a "loop variable" or rather "iterator" to which the elements of the list are assigned. `\DoWith`$\langle cmd\rangle$ does not use such a loop variable or such assignments and thus is "expandable" at least when $\langle cmd\rangle$ (and the elements, depending on $\langle cmd\rangle$) are expandable. On the other hand, `\@tfor` applies some procedure to the list elements without needing a *name* for the procedure (or a *macro* storing the procedure).

What about forloop (Nick Setzer), multido (Timothy Van Zandt, Rolf Niepraksch, Herbert Voß), and xfor (Nicola Talbot)?

xfor is just a reimplementation of `\@for`. forloop and multido are more close to "real 'for' loops" (cf. *Wikipedia*). Loops of the latter kind go through a certain set as well, but such sets rather consist of *numbers* and are exhausted by incrementing (or also decrementing) variables (counters). This is essentially not needed when a list literally is *enumerated*—such loops are distinguished as "foreach loops." I wondered whether behind LATEX's `\@tfor` (and `\@for`) there was an "ideological" consideration such as "A loop must have a loop variable!"

[1] `http://www.tex.ac.uk/cgi-bin/texfaq2html?label=inst-wlcf`

[2] arrayjobx provides somewhat "exotic" handling of "lists".

However, avoiding usage of a macro name and a macro parameter may have been a good reason.

Commands like `\DoWith` also save tokens thinking of list macros (in LaTeX/`latex.ltx`) that use a *separator macro* which may be used as a *command* to be applied to the list elements. One example is `\dospecials` that already is in Plain TeX and expands to

```
\do\␣\do\\\do\{\do\}\do\$\do\&\do\#\do\^\do\_\do\%\do\~
```

An important application of `\dospecials` is temporarily switching off the "special" functionality of the "elements" in `\dospecials`. With LaTeX, this may happen thus:

```
\let\do\@makeother\dospecials
```

With dowith, you can do the same with a shorter variant `\specials` of `\dospecials`, defined by

```
\def\specials{\␣\\\{\}\$\&\#\^\_\%\~}
```

and then

```
\DoWithAllIn\@makeother\specials
```

`latex.ltx` uses `\@elt` instead of `\do` for its own list macros.

There also is `\loop`⟨*loop-body*⟩`\repeat` in Plain TeX and a refined version of it in `latex.ltx`. It is *not* expandable since it starts with an assignment for `\body` (Plain TeX) or `\iterate` (`latex.ltx`). As to the programming structure, it is so simple and general that you cannot immediately see what kind of loops it addresses. However, the applications I have seen have been "for" or (rather) "while" loops. "While" loops can "emulate" "for" and "foreach" loops by having the "incrementation" method or the "enumeration" method in their body. This is quite obvious for "for" loops, not quite so for "foreach" loops; which for practical application (in my view) means that neither LaTeX/TeX's `\loop` macro nor in general "while" loops is/are very helpful for implementing "foreach" loops, as rather `\DoWith` and similar constructions are. The reason for this is (as it seems to me) is that you (a human being) can much more easily enumerate ("list") the items of a list (you have in mind) than define the *method* that (allegedly) is behind your enumeration. *Example:*

```
\DoWithAllOf{\printsamplearea}{\red\green\blue}
```

—*how* (according to what "method"?) did you "proceed" from `\red` to `\green` and from `\green` to `\blue`?

In LaTeX's tools bundle, xspace has a list macro `\@xpspace@exceptions@tlp` without separators. It is handled like here, except that it "breaks" the loop when an item is found that applies.

The more recent lmake provides a key-value syntax for printing lists of complex mathematical expressions easily (using some assignments) as well as defining commands according to a pattern from a list.

3 Implementation

3.1 Package File Header (Legalese)

```
\def\filename{dowith}     \def\fileinfo{simple list loop (UL)}
\def\filedate{2011/11/19} \def\fileversion{v0.2}
%% Copyright (C) 2011 Uwe Lueck,
%% http://www.contact-ednotes.sty.de.vu
%% -- author-maintained in the sense of LPPL below --
%%
%% This file can be redistributed and/or modified under
%% the terms of the LaTeX Project Public License; either
%% version 1.3c of the License, or any later version.
%% The latest version of this license is in
%%     http://www.latex-project.org/lppl.txt
%% We did our best to help you, but there is NO WARRANTY.
%%
%% Please report bugs, problems, and suggestions via
%%
%%   http://www.contact-ednotes.sty.de.vu
```

3.2 Proceeding without LaTeX

A little LaTeX as in Bernd Raichle's ngerman.sty:

```
\begingroup\expandafter\expandafter\expandafter\endgroup
```

I need `\ProvidesPackage` for fileinfo, my package version tools.

```
\expandafter\ifx\csname ProvidesPackage\endcsname\relax
```

When `\ProvidesPackage` is not defined, we provide a version of LaTeX's `\in@` (an old version that may wrongly claim to have found an occurrence of a sequence, but is correct for single tokens) for checking token list macros. LaTeX must not see `\ifin@` when it parses the `\ifx` conditional:

```
  \expandafter\newif\csname ifin@\endcsname
  \def\in@#1#2{%
    \def\in@@##1#1##2##3\in@@{%
      \ifx\in@##2\in@false\else\in@true\fi}%
    \in@@#2#1\in@\in@@}
```

readprov stops reading the file at `\Provides...`, therefore ...

```
  \long\def\@gobble#1{} \expandafter\@gobble
\else
  \expandafter\@firstofone
\fi
{ \ProvidesPackage{\filename}[\filedate\space
                    \fileversion\space \fileinfo] }
\chardef\atcode=\catcode`\@
\catcode`\@=11 % \makeatletter
```

3.3 Applying a Command

3.3.1 Core

`\DoWith{`⟨*cmd*⟩`}`⟨*list*⟩`\StopDoing` applies ⟨*cmd*⟩ to all elements of ⟨*list*⟩. An element of ⟨*list*⟩ (after tokenizing) may be either a single token or a group `{`⟨*balanced*⟩`}`.

```
\def\DoWith#1#2{%
    \ifx\StopDoing#2%
        \else#1{#2}\expandafter\DoWith\expandafter#1\fi}
```

`\StopDoing` delimits the list:

```
\let\StopDoing\DoWith
```

. . . something arbitrary that is not expected to occur in a list. With

```
\let\StopDoing*
```

instead, the star would end lists.

`\DoWithAllOf{`⟨*cmd*⟩`}{`⟨*list*⟩`}` works like

`\DoWith{`⟨*cmd*⟩`}`⟨*list*⟩`\StopDoing` :

```
\def\DoWithAllOf#1#2{\DoWith#1#2\StopDoing}
```

3.3.2 `\do` being the Command

When the ⟨*list*⟩ is worked at a single time in the TEX run where assignments are possible, instead of introducing a new macro name for ⟨*cmd*⟩ you can use `\do` for ⟨*cmd*⟩ as a "temporary" macro and define it right before

`\DoWith{\do}`⟨*list*⟩`\StopDoing`

However, we provide

`\DoDoWith{`⟨*cmd*⟩`}`⟨*list*⟩`\StopDoing`

as a substitute for the former line that at least saves one token. For the definition of `\do`, we provide `\setdo{`⟨*def-text*⟩`}`. It works similarly to

`\renewcommand{\do}[1]{`⟨*def-text*⟩`}`,

so ⟨*def-text*⟩ should contain a `#1`:

```
\def\setdo{\long\def\do##1}
```

With `\letdo`⟨*cmd*⟩ that is provided next where ⟨*cmd*⟩ is defined elsewhere, you could type

`\letdo`⟨*cmd*⟩`\DoDoWith`⟨*list*⟩`\StopDoing`

It seems to me, however, that you better type

\dowith⟨*cmd*⟩⟨*list*⟩\StopDoing

instead. So I provide \letdo although I consider it useless here. It is provided somewhat for the sake of "completeness," thinking that it might be useful at other occasions such as preceding \dospecials.

```
\def\letdo{\let\do}
```

\DoDoWith has been described above:

```
\def\DoDoWith{\DoWith\do}
```

By analogy to \DoWithAllOf, we provide \DoDoWithAllOf{⟨*list*⟩}:

```
\def\DoDoWithAllOf{\DoWithAllOf\do}
```

3.3.3 Expand List Macro

The former facilities may be quite useless as such a ⟨*list*⟩ will not be typed at a single place in the source code, rather the items to run ⟨*cmd*⟩ on may be collected occasionally when some routines run. The elements may be collected in a macro ⟨*list-macro*⟩ expanding to ⟨*list*⟩. So we provide

\DoWithAllIn{⟨*cmd*⟩}{⟨*list-macro*⟩}

(or \DoWithAllIn⟨*cmd*⟩⟨*list-macro*⟩). There is no need to type \StopDoing here:

```
\def\DoWithAllIn#1#2{%
    \expandafter\DoWith\expandafter#1#2\StopDoing}
```

\DoDoWithAllIn{⟨*list-macro*⟩} saves a backslash or token for \do as above in Sec. 3.3.2:

```
\def\DoDoWithAllIn{\DoWithAllIn\do}
```

3.4 Handling List Macros

3.4.1 Initializing

Here is some advanced \let⟨*cmd*⟩\empty, perhaps a little irrelevant for practical purposes. Both

\InitializeListMacro{⟨*list-macro*⟩}

and

\ReInitializeListMacro{⟨*list-macro*⟩}

attempt to "empty" ⟨*list-macro*⟩, and when we don't believe that LaTeX has been loaded, both do the same indeed. Otherwise the first one complains when ⟨*list-macro*⟩ seems to have been used earlier while the second complains when ⟨*list-macro*⟩ seems *not* to have been used before:

```
\expandafter\ifx\csname @latex@error\endcsname\relax
  \def\InitializeListMacro#1{\let#1\empty} %% not \@empty 2011/11/07
  \let\ReInitializeListMacro\InitializeListMacro
\else
  \def\InitializeListMacro#1{\@ifdefinable#1{\let#1\empty}}
  \def\ReInitializeListMacro#1{%
    \edef\@tempa{\expandafter\@gobble\string#1}%
    \expandafter\@ifundefined\expandafter{\@tempa}%
        {\@latex@error{\noexpand#1undefined}\@ehc}%
        {\let#1\empty}}
\fi
```

`\ToListMacroAdd{`⟨*list-macro*⟩`}{`⟨*cmd-or*⟩`}` appends ⟨*cmd-or*⟩ to the replacement token list of ⟨*list-macro*⟩. ⟨*cmd-or*⟩ may either be tokenized into a single token, or it is some `{`⟨*balanced*⟩`}`.

```
\def\ToListMacroAdd#1#2{\DefExpandStart#1{#1#2}}
\def\DefExpandStart#1{\expandafter\def\expandafter#1\expandafter}
```

3.4.2 Testing for Occurrence of a Token

`\TestListMacroForToken{`⟨*list-macro*⟩`}{`⟨*cmd*⟩`}` sets `\in@true` when ⟨*cmd*⟩ occurs in ⟨*list-macro*⟩ and sets `\in@false` otherwise:

```
\def\TestListMacroForToken#1#2{%
    \expandafter \in@ \expandafter #2\expandafter{#1}}
```

Indeed I removed an earlier `\IfTokenInListMacro`, now it's a kind of compromise between having a shorthand macro below and a generalization for users of the package.

3.4.3 Adding and Removing

`\FromTokenListMacroRemove{`⟨*list-macro*⟩`}{`⟨*cmd*⟩`}` removes the token corresponding to ⟨*cmd*⟩ from the list stored in ⟨*list-macro*⟩ (our parsing method does not work with braces):

```
\def\FromTokenListMacroRemove#1#2{%
```

I am not happy about defining *two* parser macros, but for now ...

```
    \TestListMacroForToken#1#2%
    \ifin@
      \def\RemoveThisToken##1#2{##1}%
      \expandafter \DefExpandStart
        \expandafter #1\expandafter {%
          \expandafter\RemoveThisToken #1}%
```

TODO warning otherwise?

```
    \fi}
```

... but this only removes a single occurrence ...

`\InTokenListMacroProvide{`⟨*list-macro*⟩`}{`⟨*cmd*⟩`}`

avoids multiple entries of a token by *not* adding anything when ⟨*cmd*⟩ already occurs in ⟨*list-macro*⟩ (again, this does not work with braces, try `\in@{{}}{{}}`).

```
\def\InTokenListMacroProvide#1#2{%
    \TestListMacroForToken#1#2%
    \ifin@ \else                    %% TODO warning?
        \ToListMacroAdd#1#2%
    \fi}
```

3.5 Leaving and History

```
\catcode`\@=\atcode
\endinput

VERSION HISTORY
v0.1    2011/06/23/28  stored separately
v0.2    2011/11/02     simpler, documented
        2011/11/03     corrected \if/\else for init
        2011/11/07     \TestListMacroForToken, \InListMacroProvide;
                       doc.: \pagebreak, structure
        2011/11/19     modified LaTeX supplements

```