dowith.sty

—

Apply Command to Each Item in a List of Arguments in "TEX's Gullet"*

Uwe Lück†

November 14, 2015

Abstract

This package provides macros for applying a "command" to all items in a "list of possible macro arguments," and also for extending and reducing macros storing such lists. "Brace groups" are single items of such lists, as opposed to token lists. Iteration is implemented within TEX's expansion processor, so works within `\write` as with blog.sty. Loop and list macros in other packages are discussed in the documentation. There is no need for ε-TEX to which some of them refer.

The package is "generic," i.e., should also work with Plain TEX or even other formats, relying on the plainpkg package for some minimal LATEX-like behaviour.

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

Keywords: programming structures; macro programming, loops, list macros

Contents

*This document describes version v0.3 of dowith.sty as of 2012/11/05.

†http://contact-ednotes.sty.de.vu

1 Usage and Features

1.1 Installing and Calling

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] The packages plainpkg [2] and stacklet (catcodes)[3]catcodes must be installed as well.

As to calling (loading): dowith is a "plainpkg package" in the sense of the plainpkg documentation that you may consult for details. So roughly,

- load it by `\usepackage{dowith}` if you can,
- otherwise by `\RequirePackage{dowith}`
  (perhaps from within another "plainpkg package"),
- or by `\input␣dowith.sty`

[1] http://www.tex.ac.uk/FAQ-inst-wlcf.html
[2] http://ctan.org/pkg/plainpkg
[3] http://ctan.org/pkg/catcodes

- or even by `\input{dowith.sty}` ...

1.2 What It Does With What Lists

The single commands that the package provides are described in the implementation section below. What follows here is some general background about how the commands work.

The term 'list' may refer to various things and need clarification here.

First of all, we are not referring to LaTeX `list` environments such as `enumerate` or `itemize`; neither to "TODO" lists of what needs to be done soon.

Rather, dowith allows you to abbreviate

⟨*cmd*⟩⟨*arg-1*⟩⟨*cmd*⟩⟨*arg-2*⟩ ... ⟨*cmd*⟩⟨*arg-n*⟩

by

`\DoWith`⟨*cmd*⟩⟨*arg-1*⟩⟨*arg-2*⟩ ... ⟨*arg-n*⟩`\StopDoing`

or by

`\DoWithAllOf`⟨*cmd*⟩`{`⟨*arg-1*⟩⟨*arg-2*⟩ ... ⟨*arg-n*⟩`}`

With small n, one may doubt whether this really is an abbreviation ... ; anyway,

⟨*arg-1*⟩⟨*arg-2*⟩ ... ⟨*arg-n*⟩

was an attempt to refer to the kind of lists we are dealing with.

⟨*arg-1*⟩, ⟨*art-2*⟩, ..., ⟨*arg-n*⟩

are the "items" of the list. The question is: what counts as an item?

We might say that `aa` is a list of *two* items, ⟨*arg-1*⟩ being `a` and ⟨*arg-2*⟩ being `a`, too.

When we do *three* keystrokes to get `a␣a` instead of `aa`, we still have *two* items, ⟨*arg-1*⟩ being `a` and ⟨*arg-2*⟩ being `a` too. Strange, isn't it?

Also, when in `aa` we replace the first `a` by a backslash, `\`, we get `\a`, and this is a list of a *single* item, ⟨*arg-1*⟩ = `\a` ...

You shouldn't believe these stories of mine entirely. What I am alluding to is that the *"items"* dowith is about are determined in terms of *TeX's tokens*, and the relation between the "characters you type" and *TeX's tokens* is not entirely straightforward.

1.3 The Notion of Arglists for LaTeX Users

Still, it may suffice to clarify what counts as an ⟨*arg-i*⟩ without speaking of *tokens* explicitly: It is simply what a *one-parameter macro* (where the parameter is *not delimited* in terms of *The TeXbook* pp. 203f.) can take as an *argument.*

The lists dowith is about then are lists *of possible arguments* in the previous sense—let me call them *"arglists."*[4] The single *items* of such lists are those single

[4] Not to be confused with German Arglist.

possible arguments. They become *actual* arguments beginning from the leftmost *possible* one when dowith presents them to that ⟨*cmd*⟩ mentioned earlier—where ⟨*cmd*⟩ *should* be a one-parameter macro (or some TEX primitive parsing arguments similarly).

The reader perhaps has an *intuitive* understanding of what can be an argument of a one-parameter macro. A *strict LATEX* user may think that such an argument ⟨*arg-i*⟩ just has form `{`⟨*ark-i*⟩`}`, i.e., ⟨*arg-i*⟩ = `{`⟨*ark-i*⟩`}` for some ⟨*ark-i*⟩. Such arguments are also called *"brace groups"*. (LATEX's *optional* arguments `[`⟨*extra*⟩`]` do not count as possible arguments here, they are not macro arguments in the sense of *The TEXbook*.) In this restricted LATEX sense, arglists consist of brace groups

`{`⟨*ark-1*⟩`}{`⟨*ark-2*⟩`}` ... `{`⟨*ark-n*⟩`}`,

and each single brace group is an *item* of it.

The *TEX macro writer*, by contrast, knows that a macro argument doesn't need outer braces. In an intuitive sense, a single "command" can be a macro argument, too. "Command" may be understood as "control sequence" (starting with a backslash), but some authors also have considered single *characters* (character *tokens*?) "commands." Blank spaces, by contrast, are ignored when a macro looks for its argument.

1.4 Anatomy of TEX

The documentation of v0.22 as of 2012-06-04 said that the package is about "lists in TEX's mouth." However, this was very wrong. I believed it following Alan Jeffrey's paper "Lists in TEX's Mouth",[5] in whose Section 2 you read:

> *TEX's programming facilities come in two forms—there are TEX's* macros *which are expanded in its mouth, and some additional* assignment *operations like* `\def` *which take place in the stomach.*

The macros that Jeffrey lists and describes in that article can be obtained as a CTAN package lambda-lists.[6] If you follow the link given here (in the footnote), you currently (2012-11-03) read about this package:

> *These list-processing macros avoid the reassignments employed in the macros shown in Appendix D of the TeXbook: all the manipulations take place in what Knuth is pleased to call "TeX's mouth".*

But Knuth doesn't. On page 267 of *The TEXbook*, you read:

> *Chapter 7 has described the process by which input files are converted to lists of tokens in TEX's "mouth," and Chapter 20 explained how expandable tokens are converted to unexpandable ones in TEX's "gullet" by a process similar to regurgitation.*

[5] Alan Jeffrey: "Lists in TEX's Mouth," TUGboat Vol. 11 (1990), No. 2, pp. 237–245), `http://tug.org/TUGboat/tb11-2/tb28jeffrey.pdf`.

[6] `http://ctan.org/pkg/lambda-lists`

I.e., the "mouth" is TEX's "tokenizer," the inner part of what van Eijkhout calls TEX's "input processor" on, e.g., p. 15 of his *TEX by Topic*.[7] The exact rules the tokenizer follows are described on pp. 46f. (Chapter 8!) of *The TEXbook*. *Macro expansion* takes place in TEX's *"gullet"*, which van Eijkhout calls TEX's "expansion processor" (p. 16). Abrahams, Hargreaves, and Berry follow Knuth's terminology on pp. 16 and 46f. of their *TEX for the Impatient*.[8]

TEX's gullet has been called "TEX's mouth" also in the documentation of my bitelist[9] package and in the documentation of the package bibleref-mouth[10].

Moreover, I should have clarified that Jeffrey's paper deals with "lists" in some general, rather abstract sense, different from the kind of lists the present documentation tries to characterize as the objects for dowith.

1.5 TEX's Tokens

The dowith package is a tool that affects the order of tokens in TEX's gullet.

The "characters you type" enter "TEX's mouth" line by line, in a slightly modified appearance. Each line forms a *string*. TEX takes initial substrings away from it and turns them into *tokens* that are appended to the right of TEX's *expansion buffer* ("gullet").

There are *two kinds of tokens* here: *named* tokens and *character* tokens. "Named" tokens usually are referred to as "control sequence tokens" or just "control sequences"—I really want to avoid those horrible confusions from *The TEXbook*. There never are any "parameter tokens" in TEX's gullet (perhaps unless one considers a one-step macro expansion a two-or-more-step procedure). The character(s) *after* the escape character until some delimiting character form a *string* that is the *name* of the token that is formed—a *named* token, as I am saying. *Character* tokens are formed by removing a character from the beginning of the character buffer and appending it to the token buffer paired with its *category* code.

For every *string of characters*, there is exactly one (possible) *named token* whose name the string is.[11] It is so common (starting from *The TEXbook*) to denote the token whose name is ⟨*string*⟩ by '\⟨*string*⟩'. For instance, the token whose name is `input` is denoted by '`\input`'. On the other hand, on page 7 of *The TEXbook* '`\input`' is a "string of characters." With this notation, it is already difficult to explain what the LATEX command `\DeclareRobustCommand` does or what the difference between a starred LATEX command and a starred LATEX environment is.[12] *The TEXbook* makes it worse by saying on page 39: *"A control sequence is considered to be a single object that is no longer composed of a sequence of symbols."* So "it depends" whether '`\input`' is a string of characters or not—it *is before* tokenization, but *no longer* afterwards. So if you have two computers and start a TEX run on each of them with a little difference in time, there will be a moment where '`\input`' is a string on the one computer but not on the other? This appears to me like saying *"When we apply the square root function to the number 4, the number 4 will no longer be the number 4, it will be the number 2 instead."*

[7] It is available as a CTAN package texbytopic at `http://ctan.org/pkg/texbytopic`.

[8] It is available as CTAN package impatient, `http://ctan.org/pkg/impatient`.

[9] `http://ctan.org/pkg/bitelist`

[10] `http://ctan.org/pkg/bibleref-mouth`

[11] "Possible" refers to the fact that TEX does not store named tokens anywhere before they appear in its gullet, maybe apart from "primitive" tokens that have a "pre-assigned meaning" when a TEX run starts.—What is more bad with my claim is that the TEX program by design cannot extend its memory arbitrarily—even not using the "cloud"—, so it doesn't support tokens whose name lengths are above a certain limit.

[12] A reader knowing LATEX only thinks that '\\␣' is the result of typing a double backslash and a space and that '`\equation*`' is the "command" `\equation` followed by a `*`.

The TEXbook does offer an alternative notation for named tokens: "boxing;" so the token whose name is `input` can be denoted by the rather "graphical" notation '`input`' (used only exceptionally).[13] *I* would suggest something like '`ntok(input)`' for clarity and '`?input`' for brevity.[14]

Named tokens may get into TEX's gullet by "tokenization" as described above, i.e., they are drawn from the character buffer. But they also can appear in TEX's gullet "from within," by the manipulation inside TEX's gullet.

More formally, those manipulations are called "expansion," and TEX's gullet can be conceived of as a *token buffer* that is feeded to the right (or end) by tokenization from the character buffer. Expansion means that certain tokens in the token buffer are substituted by other ones. This way tokens may get into TEX's gullet that emerged from tokenization a "long time ago", maybe in a previous run that created the *format* (TEX's variant `INITEX`); or tokens may appear by some hardwired expansion function.

However, *named tokens* may get into TEX's gullet also by *expansion*, never having been drawn by tokenization and not being hardwired. This happens by the `\csname` construct. The input *code* may contain

```
\csname␣tupni\endcsname
```

This may be converted into 7 tokens entering TEX's gullet, the first one being `ntok(csname)`, the last one `ntok(endcsname)`, and five character tokens in between. Due to some *function* (which I would denote as `*csname`) originally associated with the token `ntok(csname)`, those seven tokens then are replaced by `ntok(tupni)`, the named token whose name is `tupni`. It is not required that the TEX program knows about a token `ntok(tupni)`, neither anybody must type '`\tupni`' in any file.[15]

1.6 Arglists vs. Lists of Tokens—Example

Let us reconsider the examples from Sections 1.2 and 1.3, and pack them into a single example. If you type a file line

```
a␣a\a{a}                                                    (1)
```

(*eight* keystrokes), it should usually be converted into this *seven*-item list of tokens:

$$\texttt{a}_{11}\ \texttt{␣}_{10}\ \texttt{a}_{11}\ \texttt{ntok(a)}\ \texttt{\{}_{1}\ \texttt{a}_{11}\ \texttt{\}}_{2} \qquad (2)$$

—with notation from Section 1.5 and *The TEXbook*'s notation $\langle char \rangle_{\langle cat \rangle}$ for the *character token* that TEX's tokenizer forms from $\langle char \rangle$ in the character buffer when $\langle char \rangle$'s category code is $\langle cat \rangle$.

[13] The box notation is introduced on page 38 without explanation, as if it explained something.

[14] I am suggesting the question mark for named tokens since TEX "must look up the current definition" of a named token according to *The TEXbook* p. 39, while the meaning of character tokens rather is "fixed," at least according to *The TEXbook* p. 39. However, *active*-character such as .~ are in the same situation as named tokens as to this respect. The dot notation may be fine for them, though.

[15] These considerations may not be essential here, rather a draft for a paper. Using `dowith`, one better just thinks of the arglist items one actually lists.

It turns out that the token list in 2 provides an arglist of *four* items: The token $\mathtt{a}_{11}$ at the first and third place, the named token ntok(a), and the entire token list $\{_1\mathtt{a}_{11}\}_2$ as a single item—a "brace group." The space token is ignored.[16]

You can try this after \renewcommand{\a}{A}[17] with dowith:

```
\DoWith\typein␣a␣a\a{a}\StopDoing                    (3)
```

Then LaTeX shows a, a, A from \a, and another a from within the braces—\typein (as any macro with arguments) removes them.

I have avoided saying 2 *were* an arglist of 4 items. The mathematical basic way of writing lists—understood as finite *sequences*—as "commma-separated lists" within brackets may clarify the difference (that the juxtaposition notation tends to conceal). The *token* list is

$$(\mathtt{a}_{11}, \sqcup_{10}, \mathtt{a}_{11}, \mathtt{ntok}(\mathtt{a}), \{_1, \mathtt{a}_{11}, \}_2) \tag{4}$$

while the list of macro arguments is

$$(\mathtt{a}_{11}, \mathtt{a}_{11}, \mathtt{ntok}(\mathtt{a}), (\{_1, \mathtt{a}_{11}, \}_2)). \tag{5}$$

2 or 4 simply is *not* an arglist (since neither $\{_1$ nor $\}_2$ can be a macro argument), and the arglist 5 "provided" by the list of tokens is *not* a list of *tokens*—its final item is a three-item list of tokens, and a token cannot be a list of two or more tokens itself(*!?*).

1.7 Another Notation and the Example's Steps

To write token lists easier and hopefully easier to read, I would suggest writing '.⟨*char*⟩' for the character token that the tokenizer "usually" forms from character ⟨*char*⟩, i.e., adding the *standard* category code as in *The TEXbook* (page 37). Then 2 would read[18]

```
.a .␣ .a ?a .{ .a .}                                 (6)
```

and the corresponding arglist is

```
(.a, .a, ?a, (.{ .a .}))                             (7)
```

In "retrospect," the result of tokenizing 2 should be

```
?DoWith ?typein .a .␣ .a ?a .{ .a .} ?StopDoing       (8)
```

and the intention is that it works like

```
?typein .{ .a .} ?typein .{ .a .} ?typein .{?a .} ?typein .{ .a .}   (9)
```

(The definition of \DoWith in Section 3.3.1 indeed adds surrounding braces, if missing.) However, TeX rather tries to work with as few tokens ahead as possible. When it finds ?DoWith and the latter's meaning is the one intended by dowith, it first looks for nothing more than

[16] *The TEXbook* p. 201: "TeX doesn't use single spaces as undelimited arguments."
[17] Otherwise \a is a one-parameter macro that breaks dowith's control.
[18] See Section 1.5 for the question mark.

the two arguments required by our definition of `\DoWith`. A few moments later, the token buffer's content will just be[19]

?`typein`.{`.a.`}?`expandafter`?`DoWith`?`expandafter`?`typein`?`fi` (10)

Next ?`typein`.{`.a.`} is expanded according to the code for `\typein` in `latex.ltx`. Some unexpandable tokens will emerge and be moved into the "instruction buffer," and you should get a screen message with `a` and a prompt. When you have entered something, the remaining ?`expandafter` tokens and the ?`fi` will be removed from the character buffer, and it contains only

?`DoWith`?`typein` (11)

Another token is ordered from the tokenizer to provide a second argument for expanding ?`DoWith`. The token .␣ comes in, but that doesn't serve as a macro argument. It is removed, and the next token is `.a`. The same story as before happens, until the named token ?`a` is found ...

1.8 Summary of Possible Arglist Items

For $0 \le i \le 15$, let X_i be the set of character tokens of category code i. X_1 is the set of tokens working like {$_1$, and X_2 is the set of tokens working like }$_2$.

Let E be the set $\{3, 4, 6, 7, 8, 11, 12, 13\}$. These numbers are the category codes for *math*, *align*, *parameter*, *super*, *sub*, *letter*, *other*, *active* respectively. Let X_E be the set of character tokens of category code in E (so $\mathrm{X}_E = \bigcup_{i\in E} \mathrm{X}_i$).

Let $\circ$ be the *concatenation* operation among token lists.[20]

The following kinds of token lists form a single arglist item, i.e., can serve as an argument for an undelimited parameter:

1. a *named* token, or the single-token list consisting of it, if you prefer that;
2. a *character* token from X_E or the list consisting of it;
3. a *brace group*. That is a token list meeting the following conditions: (*i*) its *first* token is in X_1, (*ii*) its *last* token is in X_2, (*iii*) it has as many occurrences of tokens from X_1 as from X_2, (*iv*) if it is split as $\lambda \circ \rho$, there are not more X_2 occurrences in λ than X_1 occurrences in ρ ("don't close before opening").

The second claim can be checked with

`\DoWith\typein$#^_a1~\StopDoing` (12)

as to what works. (The claim is not affected by one or two surprises.)[21] Characters with different category codes either are not converted into a character token[22] or are not accepted as macro arguments. The latter applies to "brace" tokens in X_1, X_2 and to the single space token ␣$_{10}$.

As to *"brace groups"*, the third and fourth condition above are intended to say that what is between the two outer tokens is ⟨balanced text⟩ in the sense of *The TEXbook* pp. 275f. and 385; i.e., for two tokens a, b and a token list β, $(a) \circ \beta \circ (b)$ is a brace group exactly if a is from X_1, b is from X_2, and β is ⟨balanced text⟩. The conditions are more formal than what I can find in *The TEXbook*, but still they don't give me an idea of all possibities. This should be improved by the following recursive definition:

[19] If you use `\DoWithAllOf\typein{a␣a\a{a}}` instead, the entire token sequence 8 will appear in the token buffer "at once."

[20] TODO: Define for representations by maps, or: "Concatenation is about as basic as natural numbers and is understood in terms of axioms rather than by a definition."—See notes from 2011 (even with attempts with Category theory) the English Wikipedia for sequences—German article too much restricted to maps.

[21] Moreover, `\DoWith\typein#1\StopDoing` tells something about "parameter tokens."

[22] *The TEXbook* p. 47.

B1. The empty list is balanced text. B2. For any token t not in X_1 or X_2, the single-item token list (t) is balanced text. (Such a token is either a *named* token or a *character* token from X_E or *the space token* $\sqcup_{10}$.) B3. If α and β are balanced texts, then $\alpha \circ \beta$ is balanced text. B4. If β is balanced text, a is from X_1, and b is from X_2, then $(a) \circ \beta \circ (b)$ is balanced text. (This is a brace group, and the only way of getting a brace group.) B5. Nothing else is balanced text.

In other words, a token list is a brace group if and only if it is balanced text and starts with a token from X_1 and ends with a token from X_2.[23]

1.9 Summary: "Commands" Usable with dowith

In the implementation section, you learn about

`\DoWith`⟨*cmd*⟩, `\DoWithAllOf`⟨*cmd*⟩, and `\DoWithAllIn`⟨*cmd*⟩.

(LaTeX users may type `{`⟨*cmd*⟩`}` instead.) What ⟨*cmd*⟩s are allowed?

1. All **one-parameter macros** ⟨*cmd*⟩ work this way, unless there are programming mistakes outside dowith (also thinking of arguments that take over control from dowith commands before the argument list is finished).
2. **Other one-parameter** "commands" ⟨*cmd*⟩ such as TeX **primitives** may work—you must think of the fact that surrounding *braces* are added.[24] So the **primitives** `\hbox` and `\vbox` work, for instance. `\show` is an example that doesn't work at all, it takes the single starting brace token and then confuses `\DoWith`.
3. Some ⟨*cmd*⟩s taking **no argument** may make sense, e.g., for getting
   (a) apples,
   (b) pears,
   (c) peaches

   from

   ```
   \begin{enumerate}
     \DoWithAllOf{\item}{{apples,}{pears,}{peaches}}
   \end{enumerate}
   ```

   Recall that `\item` at most takes an *optional* argument.
4. ⟨*cmd*⟩ must **not take more than one** parameter. A different package will support multi-parameter macros.

[23] Again, this may be more of a draft for a paper, or notes for it, than package documentation.
[24] TODO: in the future, variants not adding braces could be added.

2 Similar Commands in other Packages

2.1 "Heavy" 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. Moreover, Ahmed Musa describes such commands as "Parsing 'tsv' lists" in documenting his catoptions package. moredefs (Matt Swift) provides list handling commands like the few that are here.[25] —In October 2012, Ahmed Musa's loops appeared on CTAN, offering loops of several "categories" about as those that are listed below, very elaborate.—I do not want to load that much. I need and only need something excessively simple, very few lines of code, as presented in Section 3. The next sections somewhat point out single features of loop constructs that I do not want to have.

2.2 Separators

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 package options). With comma-separated lists, a "string" of characters counts as an item when it is delimited by commas, or by a comma and the list "border," or spaces may be used as separators additionally. However, when LaTeX analyzes such lists (in "TeX's gullet"), it uses representations by *character tokens* of them.

The more recent lmake (Shengjun Pan) 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. Those lists are comma-separated.

2.3 "For" Loops vs. "Foreach" Loops

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 (neither helpful) when a list literally is *enumerated*—such loops are distinguished as "foreach loops."

[25] arrayjobxprovides somewhat "exotic" handling of "lists".

2.4 Iterators

So `\DoWith` and `\@tfor` rather provide "foreach" loops. A major difference between them is that the latter uses a "loop variable" or "iterator" to which the elements of the list are assigned. `\DoWith`⟨*cmd*⟩ does not use such a loop variable or such assignments and thus is "expandable" at least when ⟨*cmd*⟩ (and the elements, depending on ⟨*cmd*⟩) 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). I wondered whether behind LaTeX's `\@tfor` (and `\@for`) there was an "ideological" consideration such as "A loop must have a loop variable!" ...

Hopefully more clearly on "loop variable" vs. our approach: In order to run

⟨*code-before*⟩⟨*item*⟩⟨*code-after*⟩

on each ⟨*item*⟩ of a ⟨*list*⟩, *we here*

define `\do` as `#1` → ⟨*code-before*⟩`#1`⟨*code-after*⟩ (13)

and then run `\do{`⟨*item*⟩`}` for each ⟨*item*⟩ in ⟨*list*⟩,[26]

always replacing `\do{`⟨*item*⟩`}` by `\do{`⟨*item*⟩`}\do`. (14)

(`\do` is only an example command that dowith supports especially.) In `latex.ltx` instead, we find things like

`\@tfor\@tmp:=`⟨*list*⟩`\do{`⟨*code-before*⟩`\@tmp`⟨*code-after*⟩`}` (15)

where `\@tmp` is a *macro* that is set to be ⟨*item*⟩ at each iteration of the loop, by

`\def\@tmp{`⟨*item*⟩`}` (16)

within `\@tforloop`. After that,

⟨*code-before*⟩`\@tmp`⟨*code-after*⟩ (17)

from 15 is run.—17 like 15 is stored in a larger macro. `\do` in 15 does not act as a macro, it just delimits a macro parameter in order give a feeling of some familiar programming structure. This organisation of macros is fine when the loop body code is only used by the containing macro, while the dowith approach to store the "loop body" in an own macro has been useful when the loop body code also is used for different purposes or when it has been introduced before I thought of using it in a loop.

Note that this only was an example. In general, ⟨*item*⟩ may appear more than once in the "loop body."

"Expandability" by *avoiding* something iterating `\def\@tmp{`⟨*item*⟩`}` and doing iteration in TeX's gullet (`\do` or so must have been defined earlier) is essential especially within `\write`. Assignments do not work there. A major motivation for developing dowith developed with the blog package that `\write`s HTML code. Assignments happen in "TeX's stomach." That place might be called the "instruction buffer" to which the "expansion processor" moves items from the incoming token buffer that cannot be expanded (any more).

[26] Cf. description of procedure in terms of tokens in Section 1.7.

2.5 Separator Macros

Commands like `\DoWith` also could 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.

2.6 Ye Olde `\loop`

There also is `\loop`⟨*loop-body*⟩`\repeat` in Plain TeX and a refined[27] version of it in `latex.ltx`. It is *not* expandable since it starts with an assignment for `\body` (Plain TeX) or `\iterate` (`latex.ltx`), and then some assignments are needed to stop the loop, such as incrementing or decrementing a *counter*. As to the programming structure, it is very simple and general, I think any kind of loop can be implemented by this (apart from nested loops). E.g., I realize[28] that even a "foreach" loop could be implemented by managing a list macro, e.g., using LaTeX's internal `\@next`.

2.7 Without Iterator and Separators

In LaTeX's tools bundle, xspace was developed in the nineties by David Carlisle. It had a rather fixed exception list implemented by a deeply nested conditional. In 2004 Morton Høgholm joined, and now xspace has a list macro `\@xspace@exceptions@tlp` without separators. It is handled like here, except that it "breaks" the loop when an item is found that applies. After the "next" token is stored by the usual `\futurelet`, the exception list is searched without using an iterator. Addition and removal commands are provided as well.

[27] Using Kabelschacht's suggestion, cf. Section 4
[28] 2012-05-20

3 Implementation

3.1 Package File Header (Legalese)

```
1  \def\filename{dowith}     \def\fileinfo{simple list loop (UL)}
2  \def\filedate{2012/11/05} \def\fileversion{v0.3}
3  %%
4  %% Copyright (C) 2011 2012 Uwe Lueck,
5  %% http://www.contact-ednotes.sty.de.vu
6  %% -- author-maintained in the sense of LPPL below --
7  %%
8  %% This file can be redistributed and/or modified under
9  %% the terms of the LaTeX Project Public License; either
10 %% version 1.3c of the License, or any later version.
11 %% The latest version of this license is in
12 %%     http://www.latex-project.org/lppl.txt
13 %% We did our best to help you, but there is NO WARRANTY.
14 %%
15 %% Please report bugs, problems, and suggestions via
16 %%
17 %%   http://www.contact-ednotes.sty.de.vu
18 %%
```

3.2 Proceeding without LaTeX

v0.3 mainly replaces imitating the german.sty approach to genericity by plainpkg:

```
19 \input plainpkg
20 \ProvidesPackage{\filename}[\filedate\space
21                   \fileversion\space \fileinfo]
22 \PushCatMakeLetterAt
```

If LaTeX is not present ...

```
23 \ifltx \else
```

... an old version of its `\in@` is introduced. It is bad as a subword test (false positive cases, cf. fifinddo documentation), but dowith will check for single tokens only. If LaTeX *is* present, on the other hand, `\ifin@` is recognized while skipping `false` parts of conditionals, without being matched by some `\fi` before the next `\else`, so I hide it by `\csname`:

```
24   \expandafter\newif\csname ifin@\endcsname
25   \def\in@#1#2{%
26     \def\in@@##1#1##2##3\in@@{%
27       \ifx\in@##2\in@false\else\in@true\fi}%
28     \in@@#2#1\in@\in@@}
29 \fi
```

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*⟩`}`.

```
30  \def\DoWith#1#2{%
31      \ifx\StopDoing#2\empty
```

The previous `\empty` (replacing `%`) is a bug fix as of v0.22 (June 2012), while in my extension draft I already had it in January 2012. It allows "empty" arglist items '`{`$_1$`}`$_2$'. Before v0.22, such an empty brace group would have resulted in comparing `\StopDoing` with `\else`, so '`{`$_1$`}`$_2$' would have had the same effect as `\StopDoing`, the token text after `\else` until `\fi` would have been skipped. Instead, the user may have a reason to allow empty arguments/brace groups.

```
32          \else#1{#2}\expandafter\DoWith\expandafter#1\fi}
```

`\StopDoing` delimits the list:

```
33  \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` :

```
34  \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`:

```
35  \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`.

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

`\DoDoWith` has been described above:

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

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

```
38  \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:

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

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

```
41  \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:

```
\ifltx                                                  %% v0.3
  \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}}
\else
  \def\InitializeListMacro#1{\let#1\empty} %% not \@empty 2011/11/07
  \let\ReInitializeListMacro\InitializeListMacro
\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 ...

```
58      \TestListMacroForToken#1#2%
59      \ifin@
60        \def\RemoveThisToken##1#2{##1}%
61        \expandafter \DefExpandStart
62          \expandafter #1\expandafter {%
63            \expandafter\RemoveThisToken #1}%
```

TODO warning otherwise?

```
64      \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@{{}}{{}}`).

```
65  \def\InTokenListMacroProvide#1#2{%
66      \TestListMacroForToken#1#2%
67      \ifin@ \else                    %% TODO warning?
68          \ToListMacroAdd#1#2%
69      \fi}
```

3.5 Leaving and History

```
70  \PopLetterCatAt                                        %% v0.3
71  \endinput
72
73  VERSION HISTORY
74  v0.1    2011/06/23/28  stored separately
75  v0.2    2011/11/02     simpler, documented
76          2011/11/03     corrected \if/\else for init
77          2011/11/07     \TestListMacroForToken, \InListMacroProvide;
78                         doc.: \pagebreak, structure
79          2011/11/19     modified LaTeX supplements
80  v0.21   2012/05/14     fix for "generic" and ‘typeoutfileinfo’:
81                         @ before ...!
82  v0.21a  2012/05/19     \labels sec:apply, sec:core; \pagebreak?
83  v0.22   2012/06/04     allow {} items
84  v0.3    2012/11/05     updating copyright, using ‘plainpkg’,
85                         rewording documentation there
86
```

4 Ack.: 25 Years of Kabelschacht's \expandafter

The essential idea of dowith and \DoWith is

\if⟨*code*⟩\expandafter⟨*one-token*⟩\fi

It was described by Alois Kabelschacht as "\expandafter vs. \let and \def in Conditionals and a Generalization of PLAIN's \loop" in TUGboat Vol. 8 (1987), No. 2, pp. 184f. (a little more than one column).[29] See some German biographical notes on Kabelschacht in the German Wikipedia.[30] It seems to me that Knuth didn't note this application of \expandafter in *The TEXbook*.[31] It was then applied in many macros of latex.ltx, cf. source2e.pdf.

[29] http://tug.org/TUGboat/tb08-2/tb18kabel.pdf

[30] http://de.wikipedia.org/wiki/Benutzer:RolteVolte/Alois_Kabelschacht

[31] However, the paper *'uses the fact that the expansion of both* \else ... \fi *and* \fi *is empty.'* In *The TEXbook* I only find *'The "expansion" of a conditional is empty'* on page 213.