TEX.StackExchange cherry picking: expl3

Enrico Gregorio

Abstract

We present some examples of macros built with expl3 in answer to users' problems presented on `tex.stackexchange.com` to give a flavor of the language and describe its possibilities. Topics include list printing, string manipulation, macro creation, and graphics.

1 Introduction

My first answer on TEX.SX using expl3, the programming language for the future LATEX3, appeared in November, 2011 and a month later I issued the first version of kantlipsum. As every regular of TEX.SX knows, I like to use expl3 code for solving problems, because I firmly believe in its advantages over traditional TEX and LATEX programming.

I'll present some selected answers I have given (sometimes with modified code), also making some comparisons with traditional coding. Some objections to expl3 may be justified: it's verbose, it needs to load a few thousand lines of code. Yes, it's verbose and in my opinion this is one of its strengths: I don't think that `\hb@xt@` is clearer and easier to interpret than `\hbox_to_wd:nn`. Loading a few thousand lines of code is done almost instantly on modern machines.

Using expl3 doesn't free the user from knowing something about *macro expansion*, because this is how TEX works to begin with, but a big advantage is that commonly used and often misunderstood `\expandafter` tricks are (almost) never needed.

Some acquaintance with the language is needed for reading this paper, but I think that having the `interface3` manual at hand would be sufficient for removing most doubts.

2 List printing

Our first toy problem is to define a macro `\ocamllist` that prints a list of items in the style of OCaml. Thus we want `\ocamllist{}` to print opening and closing brackets with a thin space in between, while `\ocamllist{aa,bb}` should print $[aa; bb]$.[1]

Such a command should be flexible enough to allow recursive calls. There are several possible solution and the one by Petr Olšák is, as usual, brilliant:

```
\def\ocamllist#1{\ocamllistA #1,,}
\def\ocamllistA#1,{[#1\ocamllistB}
\def\ocamllistB#1,{%
  \ifx,#1,#1]%
  \else
    ;#1\expandafter\ocamllistB
  \fi
}
$\ocamllist{}$\par
$\ocamllist{aa}$\par
$\ocamllist{aa,bb}$\par
$\ocamllist{aa,\ocamllist{bb,cc},dd}$
\end
```

If we try it, the output is almost as required; it only lacks the thin space:

[]
$[aa]$
$[aa; bb]$
$[aa; [bb; cc]; dd]$

Writing such code, however, requires mastery of the low-level TEX language. Can we do it without having to define three macros for doing such a thing? And, most important, using a more natural language? This is, of course, where expl3 comes into the picture.

```
\ExplSyntaxOn
\NewDocumentCommand{\ocamllist}{m}
 {
  [
  \clist_set:Nn \l_hongxu_ocamllist_clist {#1}
  \clist_if_empty:NTF \l_hongxu_ocamllist_clist
   { \, }
   { \clist_use:Nn \l_hongxu_ocamllist_clist {;} }
  ]
 }
\clist_new:N \l_hongxu_ocamllist_clist
\ExplSyntaxOff
```

Yes, one needs to learn a bunch of new names for the basic functions, but there are several advantages. For instance, suppose we want to extend the macro so it accepts a star variant for automatically sized fences and an optional command for manually choosing the size of the brackets. Very easy with expl3:

```
\ExplSyntaxOn
\NewDocumentCommand{\ocamllist}{sO{}m}
 {
  \IfBooleanTF{#1}{\left[}{\mathopen{#2[}}
  \clist_set:Nn \l_hongxu_ocamllist_clist {#1}
  \clist_if_empty:NTF \l_hongxu_ocamllist_clist
   { \, }
   { \clist_use:Nn \l_hongxu_ocamllist_clist {;} }
  \IfBooleanTF{#1}{\right]}{\mathclose{#2]}}
 }
\clist_new:N \l_hongxu_ocamllist_clist
\ExplSyntaxOff
```

Now the (silly) input

```
$\ocamllist{}$\par
$\ocamllist{aa}$\par
$\ocamllist[\Big]{aa,bb}$\par
$\ocamllist*{\sum\limits_{i=1}^n a_i,bb,cc}$\par
$\ocamllist[\big]{aa,\ocamllist{bb,cc},dd}$
```

will produce

[1] `https://tex.stackexchange.com/questions/360958/`

$[\,]$
$[aa]$
$\left[aa; bb\right]$
$\left[\sum_{i=1}^{n} a_i; bb; cc\right]$
$\left[aa; [bb; cc]; dd\right]$

What's happening? In the extended macro, `s` stands for an optional `*`, whose presence can be tested with `\IfBooleanTF` which does the necessary branching. The `O{}` bit specifies an optional argument with empty default value.

Now let's analyze the bulk of the macro. With `\clist_set:Nn` we set a variable (of type `clist`) to the specified argument. This is not just the same as doing a standard `\def`, because the input is 'normalized'; for instance, leading and trailing spaces in the items are removed. This would not be a problem here, as the macro is called in math mode, but it could be for macros called in text mode. With `\clist_use:Nn` the contents of the `clist` is delivered with the specified separator *between* items. Not adding a semicolon after the last item is cleverly done by the plain TEX macros above, while with expl3 we need not worry about it. It should also be clear that `\clist_if_empty:NTF` checks whether the variable contains an empty list or not.

For those readers who are unacquainted with expl3 syntax, let's recall the main facts. A *function* in the language has a name consisting of three parts:

1. a *module name*, here `hongxu`, that acts as a sort of "name space" indicator;
2. a *proper name*, which consists of any string of characters describing the function's action, with parts separated by underscores;
3. a *signature*, after the mandatory colon, that specifies the arguments the function expects.

When reading expl3 code, one can immediately parse the arguments to a function, because of the signature. The main argument types are

- `N`, for arguments consisting of a single token, usually a control sequence, but also a character;
- `n`, for standard braced arguments;
- `T` and `F`, for the true and false branch of a conditional function, but they're syntactically the same as `n`, so the actual arguments should be braced.

There are others, and we'll see some of them in action.

Names of variables follow a similar scheme; a name consists of

1. a *prefix*, which should be `l` (local), `g` (global) or `c` (constant);
2. a module name as with functions;
3. a proper name;
4. the variable's type.

Sticking to this convention helps in reading and debugging code.

A slightly different approach for this problem would be with

```
\seq_set_split:Nnn \l_hongxu_ocamllist_seq { , } {#1}
\seq_if_empty:NTF \l_hongxu_ocamllist_seq
 { \, }
 { \seq_use:Nn \l_hongxu_ocamllist_seq {;} }
```

Here we use another data type, namely `seq` (sequence); the first command splits the input at commas, removing leading and trailing spaces from the items. An input such as `\ocamllist{,}` would be treated differently by the two versions: with `clist` it would produce an empty `clist` because of normalization; with `seq` the sequence would have two items. The choice depends on the needs at hand.

2.1 Two-row matrix input

A similar problem is typesetting a two-row matrix with the entries specified as comma-separated pairs *firstrow*/*secondrow*,[2] so the input `1/5, 2/6` means the matrix $\begin{smallmatrix}1&2\\5&6\end{smallmatrix}$. Cleverly written recursive macros are possible, but here I'll present an expl3 version.

```
\documentclass{article}
\usepackage{amsmath}
\usepackage{xparse}
\setcounter{MaxMatrixCols}{20} % or maybe more

\ExplSyntaxOn
\NewDocumentCommand{\twolinematrix}{O{}m}
 {
  \twoline_matrix:nn { #1 } { #2 }
 }

\seq_new:N \l__twoline_i_seq
\seq_new:N \l__twoline_ii_seq

\cs_new_protected:Nn \twoline_matrix:nn
 {
  \seq_clear:N \l__twoline_i_seq
  \seq_clear:N \l__twoline_ii_seq
  \clist_map_function:nN { #2 } \twoline_add:n
  \begin{#1matrix}
  \seq_use:Nn \l__twoline_i_seq { & }
  \\
  \seq_use:Nn \l__twoline_ii_seq { & }
  \end{#1matrix}
 }
\cs_new:Nn \twoline_add:n
 {
  \__twoline_add:w #1 \q_stop
 }
\cs_new_protected:Npn \__twoline_add:w #1/#2 \q_stop
 {
  \seq_put_right:Nn \l__twoline_i_seq { #1 }
  \seq_put_right:Nn \l__twoline_ii_seq { #2 }
```

[2] https://tex.stackexchange.com/questions/393053/

```
 }
\ExplSyntaxOff
\begin{document}
\[
\twolinematrix{1/6, 2/7, 3/8, 4/9, 5/10}
\qquad
\twolinematrix[b]{
  1/6, 2/7, 3/8, 4/9, 5/10,
  6/11, 7/12, 8/13, 9/14, 10/15
}
\]
\end{document}
```

It is recommended to transfer action to an expl3 command as soon as the code is more than a few lines, as we do here. The optional argument defaults to empty; this exploits the uniform syntax of the `matrix`-like environments of amsmath, so we can use `b` for brackets or `p` for parentheses and so on.

The main function clears two `seq` variables and proceeds to repopulate them. Since the argument is a comma-separated list, it's convenient to use `\clist_map_function:nN` that passes each item to the specified function, in this case `\twoline_add:n`, which will add the parts to the two sequences. The function separates the two parts by using an auxiliary function `\__twoline_add:w`, whose definition is clear: the first argument is whatever precedes the slash, the second argument is what's behind it. The "quark" `\q_stop` is customary here, the analog of `\@nil` in the LaTeX kernel.

After having populated the two sequences, we can deliver them to form the two rows of the matrix. Since `\seq_use:Nn` delivers its output in one swoop, there's no problem of already being in an alignment: typesetting will only start when the action of `\seq_use:Nn` has ended.

You might wonder why the functions are defined with the `protected` variant of `\cs_new`. It's because these functions perform assignments, in this case adding items to sequences, so they should not be expanded in a full expansion context. The unadorned `\cs_new:Nn` function should only be used for functions that can be fully expanded. This helps in avoiding the ever-loved *fragile commands*.

Another point is the declaration of function parameters: `\cs_new_protected:Nn` can deduce the parameter text from the function name being defined. On the other hand, `\cs_new_protected:Npn` needs the parameter text to be fully spelled out. In the case of `\__twoline_add:w` it is necessary because we use delimited arguments; the signature is thus `w`, for *weird*.

It would take too long to fully explain the double underscores; the idea is that functions or variables whose names start with a double underscore are *private*, whereas the others are *public*. For personal macros the distinction is a bit foggy, but it becomes important for package code: package writers are allowed to use the public functions of another package, but not the private ones, under the assumption that the syntax and action of the public functions are stable, whereas the private functions realizing the actual implementation may vary with time.

If the first row always has a sequence of integers in the natural order, we can simplify the input:

```
\NewDocumentCommand{\twolinematrix}{O{}m}
 {
  \twoline_matrix:nn { #1 } { #2 }
 }

\clist_new:N \l__twoline_row_clist

\cs_new_protected:Nn \twoline_matrix:nn
 {
  \clist_set:Nn \l__twoline_row_clist { #2 }
  \begin{#1matrix}
  1 % start at 1
  \use:x
   {
    \int_step_function:nnnN
     { 2 }
     { 1 }
     { \clist_count:N \l__twoline_row_clist }
     \__twoline_addindex:n
   }
  \\
  \clist_use:Nn \l__twoline_row_clist { & }
  \end{#1matrix}
 }
\cs_new:Nn \__twoline_addindex:n
 {
  & #1
 }
\ExplSyntaxOff
```

The calls would then be like

```
\twolinematrix{6, 7, 8, 9, 10}
\twolinematrix[b]{
  6, 7, 8, 9, 10,
  11, 12, 13, 14, 15
}
```

Here we exploit the fact we can access the number of items in a stored `clist` so we can easily generate the tokens `&2&3&...&`n by fully expanding with `\use:x` the function `\int_step_function:nnnN`. I leave filling in the details as an exercise to the reader. Time to move on.

3 String manipulation

LaTeX users sometimes have weird ideas like setting the title of a document based on the file name.[3] For instance, from a file name such as

`Chapter-Name_of_Section.tex`

the document should be titled

Chapter: Name of Section

[3] https://tex.stackexchange.com/questions/394489/

Not that I recommend such an approach, but at least it provides an occasion for describing some useful expl3 functions. We need to replace, in an expandable way, the hyphen with a colon plus space, and underscores with spaces.

expl3 defines `\c_sys_jobname_str` as its alias for the TEX primitive `\jobname`; we now make our acquaintance with another data type, namely `str` (string). The tokens in a `str` variable are stored 'literally'.[4] We also see here an example of a *constant*, that is, a variable whose value should never change.

Our approach is to examine the tokens in the file name one by one and output a replacement if needed:

```
\ExplSyntaxOn
\NewExpandableDocumentCommand{\massagedjobname}{}
 {
  \str_map_function:NN \c_sys_jobname_str
                       \ddas_jobname:n
 }
\cs_new:Nn \ddas_jobname:n
 {
  \str_case:nnF { #1 }
   {
    { - }{ :~ }
    { _ }{ ~ }
   }
   { #1 }
 }
\ExplSyntaxOff
```

Then one can do `\title{\massagedjobname}`. We pass each token to `\ddas_jobname:n`, which does the comparison: if the token is in the list in the second argument, then the corresponding replacement is done; in case of no match, the `F` branch is followed and, in this case the token itself is output. As all kernel function with `TF` branching, also `\str_case:nnTF` actually comes in four flavors

```
\str_case:nnTF
\str_case:nnT
\str_case:nnF
\str_case:nn
```

The true branch is followed when there is a match, but here we don't need it, so we can omit it using the third variant.

Again, there are classical TEX methods, but this has the big advantage of not requiring nested conditionals which would become very cumbersome when more than a couple of replacements are needed.

One might object that what we get doesn't have the correct category codes. Here's a different approach that also resets letters to category code 11.

```
\ExplSyntaxOn
\str_new:N \g_ddas_jobname_str
\NewDocumentCommand{\computetitle}{m}
 {
  \str_gset_eq:NN
   \g_ddas_jobname_str
   \c_sys_jobname_str
  \str_greplace_all:Nnn
   \g_ddas_jobname_str
   { - } { :~ }
  \str_greplace_all:Nnn
   \g_ddas_jobname_str
   { _ } { ~ }
  \tl_gset_rescan:Nnx
   #1
   { }
   { \str_use:N \g_ddas_jobname_str }
 }
\ExplSyntaxOff
\computetitle{\massagedjobname}
```

This is not expandable, but that is not a problem here, since what we need is a macro that holds the title. We perform the replacements in a more efficient fashion and then rescan the string so that the right category codes are assigned. This has to be done outside the scope of `\ExplSyntaxOn`, where the colon is a letter and the space is ignored. Global assignments are used, because the macro should be globally available; it wouldn't make a big difference here, but following a scheme is always best. The working of `\str_greplace_all:Nnn` should be clear from the function's name, and the syntax is as uniform as possible: `str` refers to the 'string' module, `greplace` stands for 'global replace'; the first argument is the `str` variable in which we want to do the replacement, the second argument is the search string, and the final one is the replacement string. There's a similar set of functions for `tl` variables. The second argument to `\tl_gset_rescan:Nnn` is for local assignments of category codes, but we need none.

Wait! Why is it `\tl_gset_rescan:Nnx`? This is a great feature of expl3. We can define *variants* of already existing functions. The argument type `x` means: a normal braced argument that is fully expanded before being passed to the main function. The expl3 kernel provides a definition for `\tl_gset_rescan:Nnn` and then has

```
\cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nnx }
```

Thus, suppose we have a macro `\foo` that takes two arguments and we want to call it by first fully expanding the second argument. The classical approach would be:

```
\newcommand{\fooexpii}[2]{%
  \edef\@tempa{#2}%
  \expandafter\fooexpii@aux\expandafter{\@tempa}{#1}%
}
\newcommand{\fooexpii@aux}[2]{%
  \foo{#2}{#1}%
}
```

[4] For TEX hackers: as characters with category code 12, except for spaces that have their usual category code of 10.

Indeed, the variant defined above does essentially this, but the nice thing is that we don't need to know the details, just enjoy the result.

3.1 Colorizing capital letters

Another funny request is to change the color of capital letters in a given token list.[5] For this a different approach is needed, with regular expressions: expl3 has a powerful regular expression engine, tailored for the special quirks of TEX.

```
\documentclass{article}
\usepackage{xparse}
\usepackage{xcolor}

\ExplSyntaxOn
\NewDocumentCommand{\colorcap}{ O{blue} m }
 {
  \sheljohn_colorcap:nn { #1 } { #2 }
 }

\tl_new:N \l__sheljohn_colorcap_input_tl
\cs_new_protected:Npn \sheljohn_colorcap:nn #1 #2
 {
  % store the string in a variable
  \tl_set:Nn \l__sheljohn_colorcap_input_tl { #2 }
  \regex_replace_all:nnN
    % search a capital letter (or more)
    { ([A-Z]+ | \cC.\{?[A-Z]+\}?) }
    % replace the match with \textcolor{#1}{<match>}
    { \c{textcolor}\cB\{#1\cE\}\cB\{\1\cE\} }
    \l__sheljohn_colorcap_input_tl
  \tl_use:N \l__sheljohn_colorcap_input_tl
 }
\ExplSyntaxOff

\begin{document}
\colorcap{\`Once \r{U}pon a Time}

\colorcap[red]{Once Upon a Time}
\end{document}
```

We store the input in a `tl` (token list) variable and then proceed to search for capital letters with `[A-Z]+` (one or more) or sequences formed by

1. any control sequence, denoted by `\cC.`,
2. an optional open brace, `\{?`,
3. one or more capital letters, and
4. an optional close brace, `\}?`.

A match is replaced by `\textcolor{#1}{`⟨*match*⟩`}`. The syntax of the replacement text is admittedly peculiar, but it's necessary for getting the correct tokens with the likewise peculiar TEX properties. Our output is:

Ònce Ůpon a Time
Once Upon a Time

Variables of type `tl` are simply containers of TEX tokens.

While tokens in a `tl` variable are usually stored with their category code, we can rescan them. An example where this is useful is for splitting Windows-style paths, which can use the backslash instead of the slash of other operating systems.[6]

3.2 Menu sequences

We'd like to be able to say `\menu{1,2,3,4}` and treat specially the first and last item, with provision for a single item. The macro should be able to specify a different separator, for cases such as

```
\menu[/]{C:/A/B/C}
\menu*{C:\A\B\C}
```

The *-variant will use the backslash as separator. The code is rather longish, but instructive.

```
\documentclass{article}
\usepackage{xparse}

\ExplSyntaxOn

% user level commands
\NewDocumentCommand{\setmenuseparator}{ m }
 {
  \tobi_menu_setsep:n { #1 }
 }
\NewDocumentCommand{\menu}{ s o m }
 {
  \group_begin:
  \IfValueT{#2}{ \tobi_menu_setsep:n { #2 } }
  \IfBooleanTF{#1}
   {
    \tobi_menu_process_rescan:n { #3 }
   }
   {
    \tobi_menu_process:n { #3 }
   }
  \group_end:
 }

% variables
\seq_new:N \l_tobi_menu_seq
\tl_new:N \l_tobi_menu_sep_tl
\tl_set:Nn \l_tobi_menu_sep_tl { , } % default
\tl_new:N \l_tobi_first_tl
\tl_new:N \l_tobi_last_tl
\tl_new:N \l_tobi_input_tl

% internal functions
\cs_new:Nn \tobi_menu_setsep:n
 {
  \tl_set:Nn \l_tobi_menu_sep_tl { #1 }
 }

\cs_new:Npn \tobi_menu_process:n #1
 {
  \seq_set_split:NVn \l_tobi_menu_seq
   \l_tobi_menu_sep_tl
   { #1 }
  \tobi_premenu:
  \int_case:nnF { \seq_count:N \l_tobi_menu_seq }
   {
    { 0 } { EMPTY }
```

[5] https://tex.stackexchange.com/questions/173209/

[6] https://tex.stackexchange.com/questions/44961/

```
    { 1 } { \tobi_singlemenu:n { #1 } }
   }
   {
    \seq_pop_left:NN \l_tobi_menu_seq \l_tobi_first_tl
    \seq_pop_right:NN \l_tobi_menu_seq \l_tobi_last_tl
    \tobi_firstmenu:V \l_tobi_first_tl
    \seq_map_function:NN
     \l_tobi_menu_seq
     \tobi_midmenu:n
    \tobi_lastmenu:V \l_tobi_last_tl
   }
  \tobi_postmenu:
 }

\cs_new_protected:Npn \tobi_menu_process_rescan:n #1
 {
  \group_begin:
  \tl_set_eq:NN \l_tobi_menu_sep_tl \c_backslash_str
  \tl_set_rescan:Nnn \l_tobi_input_tl
   { \char_set_catcode_other:N \\ }
   { #1 }
  \tobi_menu_process:V \l_tobi_input_tl
  \group_end:
 }
\cs_generate_variant:Nn \seq_set_split:Nnn { NV }
\cs_generate_variant:Nn \tobi_menu_process:n { V }

% customize to suit
\cs_new_protected:Nn \tobi_premenu:
 { \fbox{\strut pre} }
\cs_new_protected:Nn \tobi_postmenu:
 { \fbox{\strut post} }
\cs_new_protected:Nn \tobi_firstmenu:n
 { \fbox{\strut #1~(first)} }
\cs_generate_variant:Nn \tobi_firstmenu:n { V }
\cs_new_protected:Nn \tobi_midmenu:n
 { \fbox{\strut #1~(mid)} }
\cs_new_protected:Nn \tobi_lastmenu:n
 { \fbox{\strut #1~(last)} }
\cs_generate_variant:Nn \tobi_lastmenu:n { V }
\cs_new_protected:Nn \tobi_singlemenu:n
 { \fbox{\strut #1~(single)} }

\ExplSyntaxOff

\begin{document}

\menu{1,2,3,4}\par\medskip
\menu{Single Element}\par\medskip
\menu{A,B,C,D,E}\par\medskip
\menu[/]{A/B/C/D/E}\par\medskip

\setmenuseparator{/}
\menu{C:/A/B/C}\par\medskip

\menu*{C:\A\B\C}

\end{document}
```

We start off with an interesting application of input splitting; we can set a `seq` variable to the items we obtain from breaking the input at the specified sequence of tokens. The function for this is `\seq_set_split:Nnn`, whose second argument is the chosen separator. However, in this application the separator is variable, so we define a variant `\seq_set_split:NVn`. The `V` argument type means "brace the contents of the specified variable and pass it as if it were a normal argument". In classical terms the action is similar to

```
\expandafter\foo\expandafter\xyz
\expandafter{\baz}{arg}
```

where `\foo` is the three-argument macro and `\baz` is a container.

We then branch according to the number of items in the sequence. When we have more than one item, we separate off the first and the last to receive special treatment: with `\seq_pop_left:NN` we remove the leftmost item from the `seq` and store it in a `tl` variable. Then we process the first item, the middle items, and the last. Again, defining a variant is handy for processing the special items: we define a function for the explicit argument and then a `V` variant thereof.

For the *-variant, the input is first rescanned, making the backslash a printable character, and the separator is set to `\` using `\c_backslash_str`, a predefined string. Then `\tl_menu_process:V` is used so as to 'recycle' the standard function without having to bother with `\expandafter`.

Everything is done in a group in order to allow local setting of the separator, which can also be set (conforming to the standard scoping rules) by `\setmenuseparator`. Here's the output:

pre	1 (first)	2 (mid)	3 (mid)	4 (last)	post

pre	Single Element (single)	post

pre	A (first)	B (mid)	C (mid)	D (mid)	E (last)	post

pre	A (first)	B (mid)	C (mid)	D (mid)	E (last)	post

pre	C: (first)	A (mid)	B (mid)	C (last)	post

pre	C: (first)	A (mid)	B (mid)	C (last)	post

3.3 Doubling backslashes in auxiliary file

Another example of input manipulation is the request for writing to an auxiliary file, but doubling all backslashes, for feeding to an external program.[7]

```
\documentclass{article}
\usepackage{xparse}

\ExplSyntaxOn
\NewDocumentCommand{\setupstream}{ O{default} m }
 {
  \iow_new:c { g_mblanc_dbswrite_#1_iow }
  \iow_open:cn { g_mblanc_dbswrite_#1_iow } { #2 }
  \AtEndDocument
   {
```

[7] https://tex.stackexchange.com/questions/402011/

```
    \iow_close:c { g_mblanc_dbswrite_#1_iow }
   }
 }

\NewDocumentCommand{\dbswrite}{ s O{default} m }
 {
  \IfBooleanTF { #1 }
   {% argument is a macro
    \mblanc_dbswrite:nV { #2 } #3
   }
   {% argument is explicit
    \mblanc_dbswrite:nn { #2 } { #3 }
   }
 }

\tl_new:N \l__mblanc_dbswrite_text_tl

\cs_new_protected:Nn \mblanc_dbswrite:nn
 {
  \tl_set:Nx
   \l__mblanc_dbswrite_text_tl
   { \tl_to_str:n { #2 } }
  \tl_replace_all:Nxx \l__mblanc_dbswrite_text_tl
   { \c_backslash_str }
   { \c_backslash_str \c_backslash_str }
  \iow_now:cV
   { g_mblanc_dbswrite_#1_iow }
   \l__mblanc_dbswrite_text_tl
 }
\cs_generate_variant:Nn \mblanc_dbswrite:nn { nV }

\cs_generate_variant:Nn \tl_replace_all:Nnn { Nxx }
\cs_generate_variant:Nn \iow_now:Nn { cV }

\ExplSyntaxOff

\setupstream{\jobname.TESTFILE}
\setupstream[secondary]{\jobname.TESTFILESEC}

\newcommand{\test}{%
  Here are my contents: \UndefinedMacro and \\%
}

\begin{document}

\dbswrite{%
  Here are my contents:
  \UndefinedMacro and \\
}
\dbswrite[secondary]{%
  Here are my contents:
  \UndefinedMacro and \\%
}

\dbswrite*{\test}
\dbswrite*[secondary]{\test}

\end{document}
```

We find here still another data type, `iow` (input/output write). This is a good place to discuss a nice feature of expl3 regarding input and output streams. Since conventional TEX engines have a very limited number of streams (16), a new stream is allocated from a pool and when the stream is closed that stream is available again, in contrast to what happens in current LATEX (and plain TEX). Here this is irrelevant, as the stream is only closed at the end of the document, but it can be useful in other applications.

With `\iow_new:N` we can allocate a new write stream, but here we may need more than one, with a symbolic name. Thus we use a variant with the `c` type, so that the braced argument is turned into a control sequence, the counterpart of the classical

```
\expandafter\foo\csname baz\endcsname
```

This way, we can easily use a variable name. The optional argument defaults to `default`, but we can set up as many as we want. Thus the macro `\dbswrite` takes an optional argument for the symbolic name of the stream, and also has a *-variant for the case when we want to pass a `tl` argument (here a classical parameterless macro).

The argument is first so-called "stringified" with `\tl_to_str:n`, then backslashes are doubled with `\tl_replace_all:Nxx`, and finally, the contents are written out. The expl3 kernel doesn't provide every possible variant, so we need to do

```
\cs_generate_variant:Nn \tl_replace_all:Nnn {Nxx}
\cs_generate_variant:Nn \iow_now:Nn {cV}
```

It's no problem if some other code, maybe from a package we load, does the same, because an already existing variant will cause the code above to do nothing and the variants are defined in a uniform way. Similarly we need `\mblanc_dbswrite:nV` for the *-variant. Here we can see why we want `\NewDocumentCommand` to generally do only "argument parsing and normalization" and then pass control to an internal public function: we just need to concentrate on `\mblanc_dbswrite:nn` and a variant will cope with the other case.

The generated files will be identical and contain

```
Here are my contents: \\UndefinedMacro and \\\\
Here are my contents: \\UndefinedMacro and \\\\
```

4 Macro factory

In several cases one has to build several macros following a certain scheme. The mapping facilities of expl3 help in writing compact code.

The first example is about defining macros that expand to the items in a given list.[8] To begin, from `\DefinitionVariables{abc,def}` we'd like to define `\variableI` and `\variableII` expanding to `abc` and `def` respectively. Here's the code:

```
\NewDocumentCommand{\DefinitionVariables}{m}
 {
  \int_zero:N \l_tmpa_int
  \clist_map_inline:nn { #1 }
   {
```

[8] https://tex.stackexchange.com/questions/367335/

```
    \int_incr:N \l_tmpa_int
    \tl_clear_new:c
     {
      variable \int_to_Roman:n { \l_tmpa_int }
     }
    \tl_set:cn
     {
      variable \int_to_Roman:n { \l_tmpa_int }
     }
     { ##1 }
   }
 }
```

Actually, we're slightly abusing the language for defining a 'user level' macro with (a variant of) `\tl_set:Nn`.

The given list is mapped by passing each item to the second argument, where the current item is referred to as `#1`; here the hash mark needs to be doubled because we're in the body of a definition. Compare this with the standard `\@for` cycle, where the current item is stored in a macro, which typically needs to be expanded, often in an awkward way.

We can avoid allocating a new `int` (integer) variable and use the scratch one provided by the kernel. An alternative way could be

```
\NewDocumentCommand{\DefinitionVariables}{m}
 {
  \int_step_inline:nnnn
   { 1 } % start
   { 1 } % step
   { \clist_count:n { #1 } } % end
   {
    \tl_clear_new:c
     {
      variable \int_to_Roman:n { ##1 }
     }
    \tl_set:cx
     {
      variable \int_to_Roman:n { ##1 }
     }
     { \clist_item:nn { #1 } { ##1 } }
   }
 }
```

but this is less efficient because the `clist` needs to be scanned at each step. However, this is a nice way to show how we can do integer-based cycles.

4.1 Symbol abbreviation macro sets

A possibly better example of a macro factory is the following: we want to define `\CC` to stand for `\mathbb{C}` and also `\cS` to stand for `\mathcal{S}`. Of course we'd like to add other similar symbols with as little burden as possible.[9]

```
\ExplSyntaxOn
\NewDocumentCommand{\makeabbrev}{mmm}
 {
  \yoruk_makeabbrev:nnn { #1 } { #2 } { #3 }
 }
\cs_new_protected:Nn \yoruk_makeabbrev:nnn
 {
  \clist_map_inline:nn { #3 }
   {
    \cs_new_protected:cpn { #2 } { #1 { ##1 } }
   }
 }
\ExplSyntaxOff

\makeabbrev{\mathbb}{#1#1}{C,N,Q,Z,D,R,T}
\makeabbrev{\mathcal}{c#1}{A,B,C,S}
```

Tricky code, isn't it? If we try it, we'll find that `\CC` indeed expands to `\mathbb{C}`. The trick is that when `#2` is picked up, the hash marks are doubled (this is a general TEX feature). When performing each cycle in the first call of `\makeabbrev`, what TEX sees is

```
\cs_new_protected:cpn { #1#1 } { \mathbb { #1 } }
```

(because double hash marks are reduced to single during macro expansion) and, when the current item is `C`, this becomes

```
\cs_new_protected:cpn { CC } { \mathbb{C} }
```

which is exactly what we need. Here we must use the `:cpn` signature for `\cs_new_protected` because the macro we're defining has no signature itself, so the (empty) parameter text is mandatory, as it cannot be deduced. As before, `c` stands for 'make a control sequence out of the argument'.

What if we wanted to define `\AA` to `\ZZ` in one fell swoop and maybe also `\abf` to `\zbf` to stand for `\mathbf{a}` and so on? We can use the command `\int_step_inline:nnnn` to populate a `clist` and then do the same; a check whether we're redefining an existing control sequence is added as otherwise we'd get errors for `\AA` and `\SS`.

```
\ExplSyntaxOn
\NewDocumentCommand{\makeabbreviations}{mmmm}
 {% #1 = wrapper macro
  % #2 = template
  % #3 = starting letter
  % #4 = ending letter
  \clist_clear:N \l_tmpa_clist
  \int_step_inline:nnnn
   { `#3 } % start
   { 1 }   % step
   { `#4 } % end
   {% populate a clist
    \clist_put_right:Nx
     \l_tmpa_clist
     { \char_generate:nn { ##1 } { 12 } }
   }
  \clist_map_inline:Nn \l_tmpa_clist
   {
    \cs_if_exist:cTF { #2 }
     {
      \msg_term:n
       {
        Not~redefining~\c_backslash_str#2
       }
     }
     {
```

[9] https://tex.stackexchange.com/questions/207985/

```
     \cs_new_protected:cpn { #2 } { #1 { ##1 } }
    }
  }
}
\ExplSyntaxOff

\makeabbreviations{\mathbb}{#1#1}{A}{Z}
\makeabbreviations{\mathbf}{#1bf}{a}{z}
```

In the log file and on the console we'd see

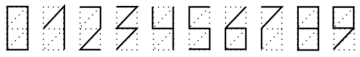
```
************************************************
* Not redefining \AA
************************************************
************************************************
* Not redefining \SS
************************************************
```

5 Graphics

I'd like to end this showcase with some new features of expl3 regarding graphic inclusion. The team (primarily Joseph Wright) is currently working on a set of APIs for the graphics driver meant to implement the same APIs as PGF, with different names, of course.

Some basic calls are already provided by the current kernel (release of 21 February, 2018, at this writing).

A funny question about printing numbers in the style required by the Soviet Union postal service appeared in October 2017.[10] These numerals look like this:



The idea is to use another expl3 data type, namely prop (property list). A property list is a container where data is identified by a key, in this case the digit. We can extract an item, typically consisting of code, by using its key.

So we start with

```
\prop_new:N \g_torcli_sovietdigits_prop

\prop_gput:Nnn \g_torcli_sovietdigits_prop { 0 }
 {
   ⟨code for 0⟩
 }
```

and so on for the other digits. Then we define an interface

```
\NewDocumentCommand{\postalcode}{O{}m}
 {
  \mbox
   {
    \keys_set:nn { torcli/sovietdigits } { #1 }
    \torcli_sovietdigits_print:n { #2 }
   }
 }
```

and now it's down to using the code for the various digits stored in the prop. I'll not go into the details of the key-value interface, suffice it to say that the code is defined in terms of some parameters, using the new LaTeX3 graphics commands to draw the necessary lines.

In the fully expl3 version, the code for 0 and 6 is

```
\prop_gput:Nnn \g_sovietdigits_prop { 0 }
 {
  \__sovietdigits_moveto:nn {0}{0}
  \__sovietdigits_lineto:nn {1}{0}
  \__sovietdigits_lineto:nn {1}{2}
  \__sovietdigits_lineto:nn {0}{2}
  \driver_draw_closestroke:
 }

\prop_gput:Nnn \g_sovietdigits_prop { 6 }
 {
  \__sovietdigits_moveto:nn {1}{2}
  \__sovietdigits_lineto:nn {0}{1}
  \__sovietdigits_lineto:nn {0}{0}
  \__sovietdigits_lineto:nn {1}{0}
  \__sovietdigits_lineto:nn {1}{1}
  \__sovietdigits_lineto:nn {0}{1}
  \driver_draw_stroke:
 }
```

where the functions __sovietdigits_moveto:nn and __sovietdigits_lineto:nn are simply syntactic sugar around the basic calls:

```
% Syntactic sugar
\cs_new_protected:Nn \__sovietdigits_moveto:nn
 {
  \driver_draw_moveto:nn
   { #1 \l_sovietdigits_width_dim }
   { #2 \l_sovietdigits_width_dim }
 }
\cs_new_protected:Nn \__sovietdigits_lineto:nn
 {
  \driver_draw_lineto:nn
   { #1 \l_sovietdigits_width_dim }
   { #2 \l_sovietdigits_width_dim }
 }
```

Happy LaTeX3ing!

◇ Enrico Gregorio
Dipartimento di Informatica
Università di Verona
and
LaTeX Team
enrico.gregorio@univr.it

[10] https://tex.stackexchange.com/questions/394616/