Recollections of a spurious space catcher

Enrico Gregorio

Abstract

Several pitfalls are waiting for us when programming in (LA)TEX. This paper will examine some and search for a solution. Shall we find the promised land? Maybe so, with `expl3`.

1 Introduction

Programming in (LA)TEX is not easy to begin with, mainly because of idiosyncrasies of the language, but also with some subtler points due to the simple fact that the language is oriented to *typeset* text. Knuth has done his best to ease typing a document without caring too much about white space: under normal circumstances, strings of spaces and tabulations are treated as a single space; line endings are converted into a single space, unless followed by another line ending with only white space intervening; white space at the beginning of lines is ignored; spaces are stripped off at the end of a line and substituted by an internal end-of-line marker.

The precise rules are explained in detail in *The TEXbook* [2] and *TEX by Topic* [1], among others, and this paper is not the place to go into the gory details. I'll return to some of the above points. Authors can insert unintentional spaces in text too, but reporting that is not the purpose here.

Some of the readers may know me by my activity on TeX.StackExchange (as user `egreg`) where, according to one of the most esteemed member of the community, I have gained the greater part of my reputation by catching spurious spaces in TEX code.[1] Nevertheless, catching spurious spaces in code is sometimes a tough task: they can be hidden in rather obscure corners and it may be necessary to resort to `\tracingall` or similar heavy machinery in order to isolate them: TEX macros call other macros, often in very involved ways.

Anyone who has undertaken the job of writing macros, be they simple or tremendously complicated, has been bitten by a spurious space left in the code. It happens! Reformatting code is perhaps one of the most frequent reasons: a line is too long and we want to improve code readability so we split it, forgetting the magical `%` at the end of the line.

Another relevant aspect of TEX's language is that under certain conditions spaces are *ignored*: they are there and TEX behaves as if they aren't, but with a very peculiar feature that will be described later.

We have to distinguish carefully between *typed spaces* and *space tokens*. Only typed spaces are subject to the above 'contraction rule', whereas *space tokens* are not. As a simple example, `\space\space` will produce two spaces: when TEX expands `\space`, it is converted to a space token.

The paper will describe the most common errors, with examples taken from questions in TeX.StackExchange or from package code. I'll give no precise references, because the purpose of the paper is not to shame anybody.[2] It will end with some considerations about methods for avoiding these quirks.

2 First examples

A very famous spaghetti western movie is "Il buono, il brutto e il cattivo" by Sergio Leone, featuring Clint Eastwood, Lee Van Cleef and Eli Wallach. The English title is "The Good, the Bad and the Ugly". I like to present examples in this form and I'll do it now.

2.1 The ugly

```
\providecommand{\sVert}[1][0]{
\ensuremath{\mathinner{
\ifthenelse{\equal{#1}{0}}{ % if
```

[1] We like to joke quite much in the site's chatroom. I'm not completely sure that that remark is a joke.
[2] Maybe I'll make a couple of exceptions.

```
\rvert}{}
\ifthenelse{\equal{#1}{1}}{ % if
\bigr\rvert}{}
\ifthenelse{\equal{#1}{2}}{ % if
\Bigr\rvert}{}
\ifthenelse{\equal{#1}{3}}{ % if
\biggr\rvert}{}
\ifthenelse{\equal{#1}{4}}{ % if
\Biggr\rvert}{}
}} % \ensuremath{\mathinner{
}
```

This code is part of a package in which no blank line separates the various macro definitions and is ugly in several respects: there is no indentation that can make clearer the various parts and the nesting of the conditionals; the code is clumsy and misses several end-of-line `%`-protections.

Remember that an end-of-line is converted to a space token; in this particular case they aren't really relevant, because most of the code is processed in math mode where space tokens are ignored. However, a user could type `\sVert` in text mode because of `\ensuremath`, ending with excess spacing around the bar: in this case the first end-of-line and the space between `}}` and `%` at the end are *not* ignored.

Using `\providecommand` is obviously wrong: users loading the package will expect the command `\sVert` do the advertised thing, not something else. A better definition would be

```
\newcommand*{\sVert}[1][0]{%
  \ifcase#1\relax
    \rvert % 0
  \or
    \bigr\rvert % 1
  \or
    \Bigr\rvert % 2
  \or
    \biggr\rvert % 3
  \or
    \Biggr\rvert % 4
  \fi
}
```

I also consider it bad programming style to place the `%` next to a control word, but it's just an opinion. Those `%` characters after `\rvert` are not necessary if no comment is used.

2.2 The bad

```
\def\@wrqbar#1{%
\ifnum\value{page}<10\def\X{\string\X}\else%
\ifnum\value{page}<100\def\X{\string\Y}\else%
\def\X{\string\Z}\fi\fi%
\def\F{\string\F}\def\E{\string\E}%
\stepcounter{arts}%
\iffootnote%
\edef\@tempa{\write\@barfile{\string%
\quellentry{#1\X{}{\thepage}}{\F{}{\thefootnote}}}}%
\else%
\edef\@tempa{\write\@barfile{\string%
\quellentry{#1\X{}{\thepage}}{\E{}{\thearts}}}}%
\fi%
\expandafter\endgroup\@tempa%
\if@nobreak \ifvmode\nobreak\fi\fi\@esphack}
```

Here we surely can't find spurious spaces. There are even too many % characters! The main problem here is that it's almost impossible to read the code.

2.3 The good

```
\def\deleterightmost#1{\edef#1{\expandafter\xyzzy#1\xyzzy}}
\long\def\xyzzy\\#1#2{\ifx#2\xyzzy\yzzyx
  \else\noexpand\\{#1}\fi\xyzzy#2}
\long\def\yzzyx#1\xyzzy\xyzzy{\fi}
```

This code by the Grand Wizard can be taken as a model of neatness. However, it's not really clear what it does: reading Appendix D of *The TEXbook* is necessary to understand it.

2.4 A surprise

```
{\tt A'C B}

\def\adef#1{\catcode`#1=13 \begingroup
 \lccode`\~=`#1\lowercase{\endgroup\def~}}
\let\oldtt\tt\def\tt{\adef'{\char"0D}\oldtt}

{\tt A'C B}

{\tt A'c B}
\bye
```

Run this code with plain TEX and you'll have quite a surprise. The purpose is to substitute the curly apostrophe with the straight one that the `cmtt10` font has in position `"0D`. The surprise is that we just get `AB` with no space in between, no quote and no `C`, but a mysterious warning in the log file:

```
Missing character: There is no ^^dc in font cmtt10!
```

This example has no spurious space; instead, it has a missing one! (Read on for details.)

2.5 Great programmers are not immune

This is an excerpt from `cleveref.sty`, a great piece of software nonetheless:

```
\cref@addlanguagedefs{spanish}{%
 \PackageInfo{cleveref}{loaded `spanish' language definitions}
 \renewcommand{\crefrangeconjunction}{ a\nobreakspace}%
 \renewcommand{\crefrangepreconjunction}{}%
 \renewcommand{\crefrangepostconjunction}{}%
 \renewcommand{\crefpairconjunction}{ y\nobreakspace}%
 [...]
```

Perhaps the author tested the language support only in some cases: the spurious space in the second line showed up when a user tried something like

```
The Spanish word for Spain is \foreignlanguage{spanish}{Espa\~na}
```

and realized that there were *two* spaces between 'is' and 'España'. Isolating the problem was far from easy, because cleveref wasn't seemingly involved. The macro `\cref@addlanguagedefs` adds to `\extrasspanish`, which is executed every time the language shifts to Spanish: so at this language change a space was produced.

See Figure 1 for another example.

3 What happens?

The rule is simple: an end-of-line is converted into a 'typed space' that *can* become a space token, but won't if it follows a control word. So, in the line

```
\iffootnote%
```

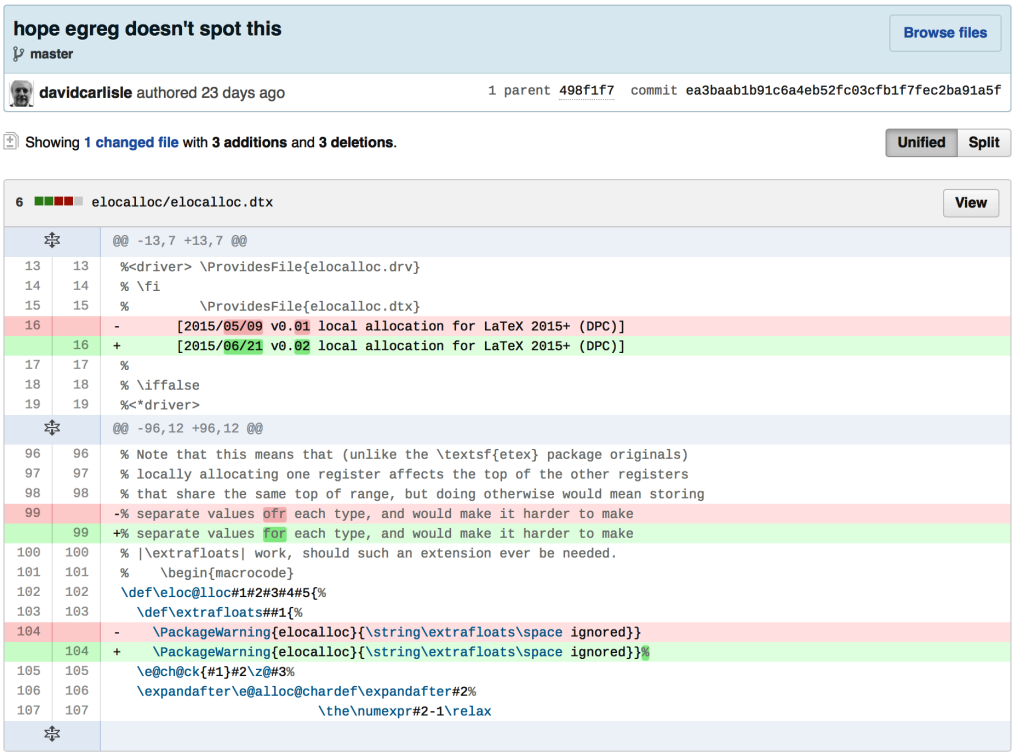

Figure 1: Another example of expert TEXnicians not being immune

the comment character is not necessary because the typed space resulting from the end-of-line will be ignored, being after a control word. To the contrary, the end-of-line in

```
\PackageInfo{cleveref}{loaded ‘spanish’ language definitions}
```

becomes a regular space token in the replacement text of the macro being defined and will disappear only if TEX is in a good mood when the macro is used.

Being in a good mood about space tokens means that the current mode is vertical (between paragraphs, basically) or math: in these cases, space tokens do nothing. If TEX is typesetting a regular paragraph (or a horizontal box), space tokens are generally *not* ignored and this is the cause for the mysterious effect presented in section 2.5.

Here are some examples of code written by people probably accustomed to other programming languages where spaces are used much more freely to separate tokens and are mostly irrelevant (unless in a string, of course).

```
\newcommand{\smallx}[1]{
     \begin{center}
        \begin{Verbatim}[commandchars=\\\{\}]

            \code{#1}

        \end{Verbatim}
    \end{center}
 }
============================
\include{fp}
\newcommand\entryOne[1]{
\ifnum #1 = 100 #1 \fi
```

Figure 2: The inscription at the base of the *Columna Traiana* in Rome

```
}
\newcommand\entryTwo[1]{
\FPeval{\result}{#1}
\ifnum \result = 100 \result \fi
}
==========================
\newcommand{\trellis}[4]{
  \def \STATES {#1}
  \def \PSK {#2}
  \def \XDISTANCE {#3}
  \def \YDISTANCE {#4}
  \FPupn\NGROUPS{\STATES{} \PSK{} div 0 trunc}
  \multido{\ryA=0+-\YDISTANCE,\nA=1+1}{\STATES}{%
    \dotnode(0,\ryA){dotA\nA}
    \dotnode(\XDISTANCE,\ryA){dotB\nA}
  }
  \multido{\nG=1+1,\nOffset=1+\PSK}{\NGROUPS}{%
    \multido{\nStart=\nG+\NGROUPS}{\PSK}{%
      \multido{\nArrows=\nOffset+1}{\PSK}{%
        \ncline{dotA\nStart}{dotB\nArrows}
      }
    }
  }
}
```

The authors of the first two examples have no notion of spurious space and write TEX code as if it was, say, C. The third example mixes end-of-line protection with 'free form code'. Counting the number of spurious spaces so produced is a funny exercise.

Unfortunately, TEX is not free form! Spaces *are* important in typesetting, unless we want to write texts like the ancient Romans did: in Figure 2 we can see an emulation of the inscription at the base of the *Columna Traiana* in Rome.[3] (The font is Trajan by Peter Wilson, `http://ctan.org/pkg/trajan`.)

The ancients did not use spaces for several reasons. The suffixes helped in dividing a word from the next one, but perhaps the most important reason was saving space: marble and parchment were very expensive. Only the introduction of print and of less expensive paper allowed for using spaces between words for better clarity and ease of reading.

Well, one says, why don't we add `%` at the end of each line in macro code and forget about the whole thing, even if the code is a bit harder to read?

Sorry, no. Here's another example that shows this is not possible: there are two diseases that I call the 'spurious space syndrome' and the 'missing space syndrome'. The latter is the more serious one.

```
\documentclass{article}
\newcount\monthlycount
```

[3] When writing the paper, I was victim of a spurious space sneaking in the text of the inscription, but fortunately I noticed it before submission.

```
\newcommand{\monthlytodo}[1]{\par%
  \fbox{%
    \parbox{10cm}{%
      \monthlycount=1%
      \loop\ifnum\monthlycount<13%
        #1--\number\monthlycount\hrulefill\par%
        \advance\monthlycount by 1%
      \repeat%
    }%
  }%
}
\begin{document}
\monthlytodo{2013}
\end{document}
```

This code should print a boxed numbered lists, with items consisting of a rule preceded by year and month number. Running it will make TEX stop, after several seconds with no sign of activity, showing

```
! TeX capacity exceeded, sorry [main memory size=5000000].
<to be read again>
                   -
l.15 \monthlytodo{2013}
```

Quite surprising, isn't it? Well, the author of the code had in the past been a victim of the spurious space syndrome, so he started to add `%` at the end of each line, even in the document part, not only in macro code.

Let's see what happens: the replacement text of the macro would be shown by TEX as

```
... \loop \ifnum \monthlycount <13#1--\number \monthlycount ...
```

and, when `\monthlytodo{2013}` is called, the test will be

```
... \loop \ifnum \monthlycount <132013--\number \monthlycount ...
```

No wonder now that it takes so long to end the loop and that TEX runs out of memory, because it's trying to build an `\fbox`!

A clear case of 'missing space syndrome'. The solution is *not* having `%` after `13`. Since the constant is part of a numeric test, we are in the special case where TEX *ignores* a space token after it.

The code in section 2.4 suffers from the same syndrome; when `{\tt A'C B}` is processed, the apostrophe becomes active and expanded like a macro, leading to the token list

```
A\char"0DC B
```

and this is where things go wrong: `\char` looks for a number in hexadecimal format because of `"` and finds the digits `DC`; since no character lives in that slot, the error message about a missing character `^^dc` is issued. The correct code should have either a space or `\relax`

```
\let\oldtt\tt\def\tt{\adef'{\char"0D }\oldtt}
\let\oldtt\tt\def\tt{\adef'{\char"0D\relax}\oldtt}
```

Which one is a stylistic decision: both the space and `\relax` stop the search for further digits; `\relax` would do nothing, but the space would be ignored. There is another possibility, that is,

```
\let\oldtt\tt\def\tt{\adef'{\active}\oldtt}
```

but this exploits the incidental fact that `"0D` = 13 and that `\active` is defined with `\chardef` to point at character 13, so it's not good programming.

Here is a worse example discovered by Frank Mittelbach some days after the TUG meeting.[4]

```
2337              \advance\@tempcnta-2%
2338              \ifnum \thevpagerefnum =\@tempcnta%
```

[4] http://tex.stackexchange.com/questions/257100/varioref-and-previous-page

Remember the spurious spaces in `cleveref.sty` discussed in section 2.5? After I reported the bug, the author went on and added `%` at the end of lines. In this case it is a very bad case of 'missing space syndrome': when TEX is looking for a numerical constant, it looks for a space token after it or a token that can't be interpreted as a digit. In doing this it *expands* tokens, so it evaluates the conditional before having set the value of `\@tempcnta` because it still doesn't know whether the number is ended. The result is that the reference will not be correct. This code is part of a redefinition of a command in `varioref.sty` and, of course, the original code (pretty much identical otherwise) has no `%` after `-2`.

The precise rule is that when a syntactical construct allows for ⟨*one optional space*⟩, TEX will look for it with expansion. Quoting *The TEXbook* [2, p. 208]

> For best results, always put a blank space after a numeric constant; this blank space tells TEX that the constant is complete, and such a space will never "get through" to the output. In fact, when you don't have a blank space after a constant, TEX actually has to do more work, because each constant continues until a non-digit has been read; if this non-digit is not a space, TEX takes the token you did have and backs it up, ready to be read again.

One could maintain that a macro writer using `\loop` should know the details of TEX programming, as this macro is not officially supported by LATEX, so the memory exhaustion problem would not show up. However, a gross estimate of the number of packages using `\loop` is 378. I didn't even try checking how many use `\ifnum`: both are essential devices in the (LA)TEX programmer's toolbox.

Surely a wannabe macro writer should be aware of these issues, but they are very subtle and, as we saw, even very experienced programmers can be victim of the bad syndrome. I could give plenty of similar examples.

4 Solutions

A possible way out of the space related syndromes might be fencing macro code with statements equivalent to doing

```
\catcode‘ =9 \endlinechar=-1 \makeatletter
```

(the last command for LATEX management of internal macros) with an appropriate restore action at the end. Something like

```
\edef\restorecodes{%
  \catcode32=\the\catcode32
  \endlinechar=\the\endlinechar
  \noexpand\makeatother
}
\catcode32=9 \endlinechar=-1 \makeatletter

... macro code ...

\restorecodes
```

but this wouldn't cure the missing space syndrome, but rather aggravate it because we can no longer add the optional space after constants! Moreover, sometimes we need space tokens in macro code. Well, we could add `\space` where we want a space, but this is cumbersome. Another bright idea comes to mind: use `~` as space. Let's try it.

```
\edef\restorecodes{%
  \catcode32=\the\catcode32
  \catcode126=\the\catcode126
  \endlinechar=\the\endlinechar
  \noexpand\makeatother
}
\catcode32=9 \catcode126=10 \endlinechar=-1\makeatletter

\newcount\monthlycount
```

```
\newcommand{\monthlytodo}[1]{\par
  \fbox{
    \parbox{10cm}{
      \monthlycount=1~
      \loop\ifnum\monthlycount<13~
        #1--\number\monthlycount\hrulefill\par
        \advance\monthlycount by 1~
      \repeat
    }
  }
}

\restorecodes
```

This is indeed more like free form. Recall that category code 9 means 'ignored' and category code 10 means 'space'. But wait a moment! The rule says that spaces are stripped at end of lines and substituted with the internal end-of-line marker, in this case nothing because the parameter `\endlinechar` is set to −1. And maybe we wouldn't have 'real spaces' in the replacement text of our macros. Again a couple of quotations from *The TEXbook* solve the issue. First about the stripping of spaces [2, p. 46]

> TEX deletes any ⟨*space*⟩ characters (number 32) that occur at the right end of an input line. Then it inserts a ⟨*return*⟩ character (number 13) at the right end of the line [...]

and this is supplemented by the fact that `\endlinechar` usually has the value 13. So the category code 10 tilde will not be stripped, because its character code is not 32.[5]

About the second issue, look at [2, p. 47]

> If TEX sees a character of category 10 (space), the action depends on the current state. If TEX is in state *N* or *S*, the character is simply passed by, and TEX remains in the same state. Otherwise TEX is in state *M*; the character is converted to a token of category 10 whose character code is 32, and TEX enters state *S*. The character code in a space token is always 32.

Thus we can count on `~` being converted to a real space token when the replacement text of the macro is stored in memory. Of course we lose the character to denote a tie, but when writing macros this is rarely needed and we can always resort to `\nobreakspace` in those cases.

Why not use `\relax` instead? It *can* be used, of course, but we lose full expandability that, in several cases, is what we need. Note that integer registers or constants defined with `\chardef` or `\mathchardef` are already 'full' numbers on their own and no space is looked for after them.

5 LaTeX3 and expl3, a new way around these problems and much more

Whoever is so kind to follow my answers at TeX.StackExchange will have probably understood where I'm going: the LaTeX3 project started more than twenty years ago, but has been stalling for a long time until a few years ago, when it became really possible to use the new programming paradigms it introduced. At the time the LaTeX team started studying it, computing resources were too scant: for instance, running LaTeX 2ε on emTEX left very little space for labels and personal macros. Nowadays, we can make presentations, plots, complex drawings with very short computing time. I remember without any nostalgia the times when drawing a mildly complicated diagram using PICTEX could take minutes! Making a 37 frame presentation from this paper just took seconds, and it involved compiling twice with a run of PythonTEX in between.

Thus the overhead of loading several thousand lines of code is not much of a problem as it could have been years ago. When these lines of code are included in the format, the loading time will be reduced to a few milliseconds.

[5] It must be mentioned that the current TEX implementations of TEX Live and MiKTEX also strip off tabs (character code 9).

The expl3 programming environment provides 'code block fences' similar to the ones I described earlier, but it also adds _ and : as characters that can be used in control sequence names. Think of _ as the traditional @ (that's not allowed in control sequence names); the use of the colon is rather interesting, and I'll return to it after having given some examples.

I'm aware that changing one's programming paradigm can be difficult at first, because habits are hard to die. But a couple of toy problems may be able to get your attention.

First problem: we want to obtain the ratio between two lengths in an expandable way to use, for instance, as a factor in front of another length parameter and we want this with as high accuracy as possible.

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

\ExplSyntaxOn % start the programming environment
\DeclareExpandableDocumentCommand{\dimratio}{ O{5} m m }
 {
  \fp_eval:n
   {
    round ( \dim_to_fp:n { #2 } / \dim_to_fp:n { #3 } , #1 )
   }
 }

\ExplSyntaxOff % end the programming environment
```

Apart from line breaks added for readability, this is essentially one line of code! And it can be 'free form'! I use the \fp_eval:n function that produces the result of a computation, together with data type conversion functions. If we try \dimratio{\textwidth}{\textheight} or \dimratio[2]{\textwidth}{\textheight} we obtain, respectively, 0.62727 and 0.63 and we could even say

```
\setlength{\mylength}{\dimratio{\textwidth}{\textheight}\mylength}
```

in order to scale the value of the parameter \mylength by this ratio.

Second toy problem. We want to parse a semicolon-separated list, applying to each item some formatting macro \dosomething. First some nice code by Petr Olšák:

```
\def\ls#1{\lsA#1;;}
\def\lsA#1;{\ifx;#1;\else \dosomething{#1}\expandafter\lsA\fi}
\def\dosomething#1{\message{I am doing something with #1}}

\ls{(a,b);(c,d);(e,f)}
```

This is a well-known technique which does its job nicely, but has some shortcomings that I'll illustrate after showing the corresponding expl3 code:

```
\ExplSyntaxOn
\NewDocumentCommand{\dosomething}{m}
 {
  I ~ am ~ doing ~ something ~ with ~ #1
 }
\seq_new:N \l_manual_ls_items_seq
\NewDocumentCommand{\ls}{m}
 {
  \seq_set_split:Nnn \l_manual_ls_items_seq { ; } { #1 }
  \seq_map_function:NN \l_manual_ls_items_seq \dosomething
 }
\ExplSyntaxOff

\ls{(a,b); (c,d) ;(e,f)}
```

One of the new data types introduced in `expl3` is the *sequence* type: an ordered list of items, which can be accessed as a whole or by item number. The first function does the splitting and loads the declared variable with the items; then `\seq_map_function:NN` passes each item as the argument to the function `\dosomething`, which is just the same as Petr Olšák's macros.

Oh, wait! If you look carefully, in my example there are spaces surrounding the middle item, which would sneak into Petr's code, while they don't with the `expl3` code, because the splitting function automatically trims off leading and trailing spaces around an item. So users can even type the input as

```
\ls{
  (a,b);
  (c,d);
  (e,f)
}
```

if they deem it convenient.

Let's go back to the Grand Wizard's code shown in section 2.3:

```
\def\deleterightmost#1{\edef#1{\expandafter\xyzzy#1\xyzzy}}
\long\def\xyzzy\\#1#2{\ifx#2\xyzzy\yzzyx
  \else\noexpand\\{#1}\fi\xyzzy#2}
\long\def\yzzyx#1\xyzzy\xyzzy{\fi}
```

It's supposed to remove the last item from a sequence. In Appendix D [2, p. 378], sequences, called *list macros*, are implemented as macros with a replacement text such as `\\{(a,b)}\\{(c,d)}\\{(e,f)}`, so one could do

```
\def\myitems{\\{(a,b)}\\{(c,d)}\\{(e,f)}}
```

and the `\deleterightmost` macro is supposed to be called like

```
\deleterightmost\myitems
```

in order to remove the last item, so leaving the same as if the definition had been

```
\def\myitems{\\{(a,b)}\\{(c,d)}}
```

Compare the code above with the `expl3` code

```
\seq_pop_right:NN \l_manual_ls_items_seq \l_tmpa_tl
```

which has also the significant advantage that the last item is still available in the token list scratch variable `\l_tmpa_tl` (or any token list variable we choose to use). Knuth's macro simply discards it.

Here's a new version for the `\monthlytodo` macro:

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

\ExplSyntaxOn
\NewDocumentCommand{\monthlytodo}{ O{10cm} m }
 {
  \par \noindent \fbox { \manual_monthlytodo:nn { #1 } { #2 } }
 }
\cs_new_protected:Nn \manual_monthlytodo:nn
 {
  \parbox { \dim_eval:n { #1 - 2\fboxsep - 2\fboxrule } }
   {
    \int_step_inline:nnnn { 1 } { 1 } { 12 }
     {
      #2--\int_to_arabic:n { ##1 }\hrulefill\par
     }
   }
 }
```

```
\ExplSyntaxOff
\begin{document}
\monthlytodo{2013}
\monthlytodo[\textwidth]{2015}
\end{document}
```

Note that there's no `\loop` any more, but the much more convenient `\int_step_inline:nnnn` function that accepts as its first three arguments, respectively, the starting point, the step and the ending point; the fourth argument contains code that should use the current value, available as `#1` which in this case must become `##1` because we're doing a definition. I also added an optional argument to set the width of the box and show `\dim_eval:n`.

No need to be obsessed by adding spaces after constants, because objects are clearly separated from each other. If a function needs a numeric argument, it will be given in braces and the function's *signature* tells us how many arguments are expected (of course, one has to know what type of argument should be given).

Just to show the power of `expl3`, let me make a bimonthly calendar:

```
\cs_new_protected:Nn \manual_bimonthlytodo:nn
 {
  \parbox { \dim_eval:n { #1 - 2\fboxsep - 2\fboxrule } }
   {
    \int_step_inline:nnnn { 1 } { 2 } { 12 }
     {
      #2 --
      (\int_to_arabic:n { ##1 } -- \int_to_arabic:n { ##1 + 1 })
      \hrulefill\par
     }
   }
 }
```

where the step is two; only six steps will be performed, because at the next the value would be above the stated upper bound: let TEX do the computations.

In `expl3` a careful distinction is made between *functions* and *variables*. They are of course all realized as macros or registers; the distinction is in what they are for: functions do something, whereas variables store tokens.

The naming scheme makes it easy to distinguish between them: a function has a signature made of zero or more characters, separated from the name by a colon. The introductory manual [7] explains what characters are valid and what they mean. The name of a variable should start with `l_`, `g_` or `c_`, meaning *local*, *global* or *constant*; `expl3` provides distinct functions to act on local and global variables; constants should just be allocated and given a value. A common source of head scratching is memory exhaustion due to filling up the save stack; always adding in the correct way to variables should avoid the problem. The interface manual [6] describes all available kernel functions; there are also other manuals for the still-experimental packages such as the one for regular expressions [8], and not to forget the higher level command defining package xparse [9].

6 Advantages and disadvantages

Powerful tools always have pros and cons. Let me make a short list of the good parts I've found in `expl3`.

1. Consistent interface: the team is striving for a set of basic functions in such a way that the name suggests the action.
2. Several new data types: TEX basically has only macros, but `expl3` builds higher level structures that help in keeping different things apart.
3. Tons of predefined functions: the most common tasks when operating on data structures are provided and the team is usually responsive when uncovered use cases are shown.

4. Ongoing development: the state of `expl3` is quite stable; the good thing is that it hides the implementation details of data types and basic functions, so changes at the lower level will have no effect on higher level constructs (apart from speed and efficiency).

   Here's a list of the new data types:

token lists are generic containers for tokens (think of `\chaptername`);

sequences are ordered sets of token lists;

comma separated lists are the same, more user interface oriented;

property lists are unordered sets of token lists, addressed by key;

floating point numbers adhere to the IEEE standard;

regular expressions are as near to the POSIX standard as possible;

coffins are ordinary TEX boxes but with many more handles.

The available data types of course include booleans, integers, boxes, lengths, skips (rubber lengths), input and output streams as in standard TEX.

Several papers in *TUGboat* have touched on the problem of case switching macros; here's a simple example of how this is done in `expl3`:

```
\str_case:nnTF { <string> }
 {
  {a}{Case~a}
  {b}{Case~b}
  {z}{Case~z}
 }
 {Additional code if there is a match}
 {Code if there is no match}
```

The ⟨*string*⟩ mentioned in the first argument is usually the argument to a function/macro. This also shows the purpose of function signatures: the function expects four braced arguments, the last two denote code to execute if the test returns true (in this case that the first argument is found in the list specified in the second argument) or false, respectively. Also `\str_case:nn`, `\str_case:nnT` and `\str_case:nnF` are provided, for cases when the ⟨*false code*⟩ or ⟨*true code*⟩ are not needed, respectively: an example of what I mean by 'consistent interface'.

Another example of consistent interface is the following. Suppose we have to do something with two token lists, one of which is explicitly given and the other one is sometimes only available as the contents of a token list variable.

```
\cs_new_protected:Nn \manual_foo:nn
 {
  Do~something~with~#1~and~#2
 }
\cs_generate_variant:Nn \manual_foo:nn { nV , Vn , VV }

\manual_foo:nn { Bar } { Foo }
\manual_foo:nV { Bar } \l_manual_whatever_tl
\manual_foo:Vn \l_manual_whatever_tl { Foo }
\manual_foo:VV \l_manual_whatever_a_tl \l_manual_whatever_b_tl
```

The *variant* is defined in a consistent way, giving us four similarly named functions that perform the same action, only on differently specified arguments. Doing the same in standard LATEX requires juggling arguments and well placed `\expandafter` tokens: a nice game to play, but usually leading to undesired code duplication. The `V` type argument means that the argument is expected to be a variable, whose value is obtained and placed in a braced argument for processing with the base function.

Now let's turn to the cons. The code is much more verbose, rather like C code is much more verbose than assembler; a price to pay when the pool of available base function is larger. One has never (well, almost) to use `\expandafter`, the preferred toy of all TEX programmers;

in particular, no `\expandafter\@firstofone`. (Er, just joking.) Coding in `expl3` still requires understanding how macro expansion works and error messages can be quite cryptic, as they often refer to TEX's lowest level; however, also programming in plain TEX can lead to inscrutable errors, so this is already a problem.

7 The real advantages

The `.../tex/latex` subtree in TEX Live contains more than 2000 directories. If we look at those packages, we can easily see that several of them reinvent the wheel many times. It's not rare to even see LATEX kernel provided functions/macros reimplemented! The `\ls` macro described above, with its auxiliary macro `\lsA` is an example: the code is good, but it can be found umpteen times in small variations, for every parsing task. The same can be said for dozens of typical constructs, so having a wide base of functions for common tasks is going to make code much more readable and understandable. Studying the `\xyzzy` example is certainly fun and instructive for grasping concepts related to recursive macros, particularly tail recursion. But why redo the work the LATEX team has already done for us?

Frank Mittelbach's papers (with Rainer Schöpf and Chris Rowley) [3, 4, 5] about LATEX3 are very interesting reading for understanding the aim of the project.

So it's not just about 'avoiding spurious spaces' or not 'forgetting required spaces': we have available, and in a quite mature state, an almost *complete* (and extendable) programming environment that frees us from thinking about the low-level stuff, while concentrating on the real programming task.[6]

Since the meeting has taken place in Darmstadt, Germany, I'd like to finish in the German language, with apologies to David Hilbert:

Aus dem Paradies,
das das LATEX3 Team uns geschaffen,
soll uns niemand vertreiben können.

(From the paradise that the LATEX team created for us, no one can expel us.)

References

[1] Victor Eijkhout. *TEX by Topic, A TEXnician's Reference*. Addison-Wesley, Reading, MA, USA, 1992. `http://eijkhout.net/texbytopic/`.

[2] Donald E. Knuth. *The TEXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

[3] Frank Mittelbach and Chris Rowley. LATEX 2.09 ↪ LATEX3. *TUGboat*, 13(1):96–101, April 1992. `http://tug.org/TUGboat/tb13-1/tb34mittl3.pdf`.

[4] Frank Mittelbach and Chris Rowley. The LATEX3 Project. *TUGboat*, 18(3):195–198, September 1997. `http://tug.org/TUGboat/tb18-3/l3project.pdf`.

[5] Frank Mittelbach and Rainer Schöpf. Towards LATEX 3.0. *TUGboat*, 12(1):74–79, March 1991. `http://tug.org/TUGboat/tb12-1/tb31mitt.pdf`.

[6] The LATEX Project. The LATEX3 interfaces, 2015. `http://mirror.ctan.org/macros/latex/contrib/l3kernel/interface3.pdf`.

[7] The LATEX Project. The expl3 package and LATEX3 programming, 2015. `http://ctan.org/pkg/l3kernel`.

[8] The LATEX Project. The l3regex package: Regular expressions in TEX, 2015. `http://ctan.org/pkg/l3regex`.

[9] The LATEX Project. The xparse package: Document command parser, 2015. `http://ctan.org/pkg/xparse`.

◇ Enrico Gregorio
Università di Verona, Dipartimento di Informatica
Strada le Grazie 15, Verona, Italy
`enrico dot gregorio (at) gmail dot com`

[6] In a conversation at the TUG meeting, Stefan Kottwitz asked me to remark on the best advancements with `expl3`; I answered 'floating point computations' and 'regular expressions'. These are of course implemented with TEX primitives, but my opinion is that they wouldn't be so powerful if `expl3` had not been developed.