An introduction to expl3

Marei Peischl

Even some long-term LaTeX users seem to be scared of expl3 — the syntax of the LaTeX3 programming layer — and think of the structure as confusing or even frightening. Perhaps with some justification:

```
\ExplSyntaxOn
\clist_map_inline:nn
    \l_tmpa_clist
    { \__ptxcd_add_item:n {#1} }
\ExplSyntaxOff
```

LaTeX3 is no longer a development for which LaTeX users have been waiting for decades. LaTeX3 has been around for a long time and nowadays is used by all LaTeX users, often without being noticed. The goal of this tutorial is to demystify expl3.

The LaTeX3 programming layer is the foundation of almost all new LaTeX development in the last years. It provides unified interfaces that can be used directly or indirectly by package authors and users to code complex mechanisms or process content much more flexibly than with classic LaTeX.

Overall, the most important goals of LaTeX3 are:

- Uniform interfaces for functions and variables
- Modernization of syntax
- Simplification of controlling expansion

and thus provide both much simpler and more powerful ways to program in LaTeX [5].

Programming is useful when a document, depending on settings, should get either a different layout or a different structure. A typical example from teaching is the creation of an exam including solutions within a single file, where solutions can be hidden. Another use case is the processing of external data. A typical example is a list to be converted to an enumeration:

```
\ExplSyntaxOn
\begin{enumerate}
    \clist_map_inline:nn
        { one, two, three }
        { \item #1 }
\end{enumerate}
\ExplSyntaxOff
```

1. one
2. two
3. three

The expl3 syntax often seems cryptic to users of other programming languages. As a combination of underscores and colons and a bunch of naming conventions form a unique structure, expl3 is a nice way to remind everyone of the fact that LaTeX, and thus expl3, is a pure macro language. Here, tokens are replaced by their meaning and no actual operations are performed, as opposed to scripting languages.

Understanding the structure of expl3 therefore requires a basic understanding of macro expansion and the concept of category codes. The following sections explain the basics of these, in addition to the syntactic structure. If the concepts are already familiar, the corresponding sections can be skipped.

1 Syntax switching in TeX, LaTeX, expl3

When TeX processes input, it not only reads individual characters, but also assigns a category to each character. This category determines how the character should be processed. The assignment is done using the so-called "category codes" or "catcodes" for short. Each input character corresponds to a character code, and each character code is assigned to a (changeable) category code.

In total, TeX knows sixteen different categories. The assignments of a character to a category can change within a document. The most common example is language-dependent behavior, such as constructed by babel [1]. For German documents, one can type `"a` to produce "ä". (In English documents, each character is processed separately.) This is done using category 13 "active". Active characters are no longer simple characters, but commands; in this case, the command to put an umlaut over the following character.

The following list shows all available categories along with explanations and examples, many of which are familiar to all TeX typists, even if you haven't heard of category codes.

0. Escape character (`\`)
1. Beginning of group (`{`)
2. End of group (`}`)
3. Math shift (`$`)
4. Alignment tab (`&`)
5. End of line (⟨return⟩)
6. Parameter (`#`)
7. Superscript (`^`)
8. Subscript (`_`)
9. Ignored character (⟨null⟩)
10. Space (␣)
11. Letter (non-ASCII only with XeTeX/LuaTeX)
12. Other character (`@`)
13. Active character (`~`)
14. Comment character (`%`)
15. Invalid character (⟨delete⟩)

doi.org/10.47397/tb/44-1/tb136peischl-expl3

1.1 The @ character in (LA)TEX macro names

LaTeX uses @ to protect internal macros from access by end users. Usually, internal macros are protected this way for a reason. So customizations should be done with caution to avoid unexpected side effects. While being aware of danger, it is possible to use @ within command names, by changing the category code of @ to 11 (letter), and back to 12 (other) when no longer needed:

```
\makeatletter
\makeatother
```

A typical example for the use of the @ character is the definition of "starred" variants for one's own commands. For this, two auxiliary macros must be defined internally, which are often also protected with an @:

```
\makeatletter
\newcommand*{\cmd}
    {\@ifstar\@cmdstar\@cmd}
\newcommand*{\@cmd}{without *}
\newcommand*{\@cmdstar}{with *}
\makeatother
```

Then the macro `\cmd` has the following output:

```
\cmd \\ \cmd*
```

without *
with *

1.2 The expl3 syntax

Since the focus of the expl3 syntax is programming, LaTeX behaves fundamentally differently when coding a command than when writing text:

- Spaces and newlines in code delimit tokens, but are otherwise ignored.
- Blank lines are not paragraph breaks.
- Tilde (`~`) characters are a normal space (catcode 10), not a tie.
- Colon (`:`) and underscore (`_`) characters are part of macro names.
- There is a syntactic difference between functions and variables.
- It is recommended to put spaces around curly braces unless they contain only one parameter.

These changes allow us to structure the code quite a bit better, without changing its meaning. Switching to (or from) the expl3 programming syntax is done with these commands:

```
\ExplSyntaxOn
\ExplSyntaxOff
```

Additionally, most package authors nowadays use CamelCase for the commands for their users. This improves the readability of the code even within the classic LaTeX syntax.

2 Naming scheme

Using a naming scheme, expl3 distinguishes at a glance functions that process content or arguments from variables that simply store a value. Function names contain a : character, and variables do not:

Functions:

module_description:*arguments*

Variables:

*validity*_*module_description_datatype*

2.1 Variables

Variables store values. Expl3 provides different data types for this. The naming scheme is the same for all types. Technically, this is only a convention. However, following it ensures that users can understand the code more easily.

*validity*_*module_description_datatype*

Validity constant (c), global (g) or local (l).

Internal? Internal variables which are not intended for end-users separate the *validity* from the *module* by two underscores. Normal variables have only one underscore here.

Module Named for the package/bundle to avoid name conflicts. There is a process to register module names; see [6].

Description What does the variable store for which purpose?

Datatype What kind of values are stored in the variable? How must it be processed?

An example of an internal variable (two underscores) is the token list (`tl`):

```
\l__siunitx_complex_sign_tl
```

There are a wide variety of data types, each with its own abbreviation. A small selection is shown in Table 1. All interfaces are documented in [4].

2.2 Functions

The term "function" is perhaps a bit misleading for those familiar with other programming languages. TeX and thus expl3 is a pure macro language, which

Table 1: Selection of expl3 data types

Type	Description
`bool`	boolean (true or false)
`box`	box
`clist`	comma-separated list
`coffin`	"box with handles", a box with anchor points for relative placement with respect to other objects.
`dim`	length (dimension)
`fp`	floating point numbers (double precision)
`int`	integer
`prop`	key/value list (property list)
`seq`	sequence (queue/stack)
`skip`	extensible length (glue)
`str`	string, consisting only of letters, spaces and category 12 (other) characters
`stream`	input/output streams for reading/writing external files
`tl`	token list, string with arbitrary catcodes

gives the impression of a function that operates only by substituting strings. However, functions in expl3 can also be understood to process their contents, with no return value in the sense of conventional programming languages. The "return value" is often left in the "input stream" and thus becomes part of the document.

The naming convention for functions is:

`\`*module_description*`:`*argument specification*

The *module* and *description* for function names are identical to those for variables. However, the difference between whether a function acts locally or globally is recorded in the description. The convention prefixes a `set` for a (local) assignment and an additional "`g`" for a global assignment: `gset`. Examples are the functions for setting integer variables:

```
\int_set:Nn
\int_gset:Nn
```

In expl3 syntax, the arguments expected by the function are specified after the colon. Thus, the two examples above expect two arguments: one of type "N" and a second of type "n". Thus, one can directly look at a function for the number and types of expected arguments. This will be important for controlling the expansion process, but for now, just N and n are enough to deal with. The letter stands for "No manipulation" in each case. Thus, the argument is passed to the function without any further processing.

The difference between the upper and lower case is whether the function expects a single token or a grouped argument. Upper case letters stand for tokens; lower case for groups. Thus, our argument specification `:Nn` above expects a single token followed by a group. Altogether, the macro `\int_set:Nn` could be used as follows:

```
\int_set:Nn \l_tmpa_int { 5 }
```

The above example sets an integer variable (the first argument, here `\l_tmpa_int`) to the value "5" (specified in a group).

The function is thus similar to classic LaTeX's `\setcounter`, but the argument of `\int_set:Nn` can also be used to calculate. The function `\int_use:N` returns the value of the given variable, in this case typesetting it:

```
\ExplSyntaxOn
\int_set:Nn \l_tmpa_int { 5 + 2 * 3 }
\int_use:N \l_tmpa_int
\ExplSyntaxOff
```

```
11
```

2.3 dim: Example of a data type

This section is devoted to the "dim" data type for lengths. The most common functions are discussed. Analogous functions exist for other data types. Complete documentation is given in [4].

2.3.1 Initialization: N/n arguments

```
\dim_new:N
\dim_const:Nn
```

`\dim_new:N` is used to create a new variable. It then exists globally, but can also be assigned locally. Here, the naming convention is crucial. To create a local and a global variable for our present tutorial, we do:

```
\dim_new:N \l_tugboat_test_dim
\dim_new:N \g_tugboat_test_dim
```

It is thus possible for two variables with the same *description* to exist, one of which is to be assigned locally and one globally.

When defining a constant, its value is also directly specified:

```
\dim_const:Nn \c_tugboat_test_dim { 5cm }
```

For variables, which are changeable, the assignment is done separately with `\dim_set:Nn` (or `_gset`):

```
\dim_set:Nn \l_tugboat_test_dim { 3cm }
\dim_gset:Nn \g_tugboat_test_dim
                          { 1cm + 5mm }
```

Analogously to what we saw with counters, calculation is also possible here. Precision is limited to that of TEX for lengths. The smallest unit is 1 sp = 0.000 02 pt. And thus definitely small enough to have more than sufficient accuracy for print production.

Besides assignments, there are also commands for addition and subtraction of lengths. The name structure remains identical here. All details can be looked up in [4].

2.3.2 Conditionals and loops: T/F/TF arguments

Another basic part of programming is the ability to compare values of variables, and create loops based on such conditions. For lengths, the simplest way to do this in expl3 is to use the command:

`\dim_compare:nTF`

This macro can be used to compare lengths with each other. The possible comparisons are:

Equal	`=` or `==`
Greater-equal	`>=`
Larger	`>`
Less-equal	`<=`
Smaller	`<`
Unequal	`!=`

There are other variants of the `\dim_compare:` function that allow only some of the operators; they are processed faster. Here, we discuss only the simplest and most general command. As a complete example, our just-created and assigned global and local lengths can be compared with each other:

```
\ExplSyntaxOn
\dim_compare:nTF { \g_tugboat_test_dim >=
                   \l_tugboat_test_dim }
   { is~greater~or~equal }
   { is~smaller }
\ExplSyntaxOff
```

is smaller

The specification TF expects one group or token per letter according to the previous description. Here, the “T” stands for “true”, the “F” for “false”.

Expl3 has a special feature here: you are allowed to specify only the branch that is needed. If the function should output something only if the query returns the value “false”, the T argument can simply be omitted:

```
\ExplSyntaxOn
\dim_compare:nF
       { \g_tugboat_test_dim >
         \l_tugboat_test_dim }
   { not~greater }
\ExplSyntaxOff
```

not greater

2.3.3 Debugging outputs

In more complex programming, it may be necessary to display values of variables on the fly. Expl3 provides commands to output the current value of a variable in the terminal or to write it to the log file.

```
\dim_show:N
\dim_show:n
\dim_log:N
\dim_log:n
```

The variants with type “n” arguments expect a length expression instead of a length variable. Here you can calculate again or evaluate a macro which contains for example a centimeter value.

2.4 Argument specifications

In addition to the argument specifications already mentioned, there are several others, most of which process arguments differently than in standard arguments in classic LATEX. Table 2 shows all of them and their description.

`N` and `TF` have already been explained. Type `c` follows next, and section 4 will explain the specifications `o`, `f`, `x`, `e`, and `V`. The types `p` and `w` are less common, especially for beginners, and won't be discussed here; they're listed for the sake of completeness. Explanations can be found in [5].

Table 2: Argument specifications for expl3 functions

Token	Description
`wN/n`	No manipulation
`TF/T/F`	True/False
`c`	Csname
`V/v`	Value
`o`	expand Once
`x`	eXhaustive expansion
`e`	Exhaustive expansion, but the macro might be expandable
`f`	Full expansion to first unexpandable token
`p`	Parameters as for TEX definitions
`w`	Weird

3 Csname/Endcsname: c arguments

A fundamental concept of LaTeX is the ability for macro names to be created dynamically:

`\csname` *name*`\endcsname`

One can roughly describe the functionality of this construction as prefixing a backslash:

```
\LaTeX{}
=
\csname LaTeX\endcsname
```

LaTeX = LaTeX

A typical example of practical use is references. Here, a macro is defined internally that contains the argument of the `\label` command:

```
\label{frame:csname}
\expandafter\meaning
    \csname r@frame:csname\endcsname
```

macro:->{3}{91}{Csname\slash Endcsname: \texttt {c} arguments}{section.3}{}

A reference is thus created as a macro containing the element number — in this case empty because not numbered — and the page number. For `\ref`, the first value is used. `\pageref` uses the second. For more information on the topic, including several examples, see [3].

In expl3, this concept is the foundation for type `c` arguments — `c` as in "csname". Here, we've seen the first command, `\dim_set:Nn`, which sets `\l_tmpa_dim` to `1cm`. The second command, using `\dim_set:cn`, similarly sets `\l_tmpb_dim` to `2cm`, but the variable is given as a name instead of a literal control sequence. The third uses names for both the variable name and the value.

```
\dim_set:Nn \l_tmpa_dim { 1cm }
\dim_set:cn { l_tmpb_dim } { 2cm }
\dim_set_eq:cc { l_tmpa_dim }
               { l_tmpb_dim }
```

4 Controlling expansion: o/x/e arguments

Expansion essentially means replacing a command with its meaning. For classic commands in LaTeX, the expansion of a macro can be seen by the commands

`\meaning\`*command*
`\show\`*command*

The result can be displayed in the text or output to the terminal. For example:

```
\newcommand*{\myVariable}{myvalue}
\meaning\myVariable\\
\newcommand*{\myFunction}[1]{%
    function with argument (#1)
}
\meaning\myFunction
```

macro:->myvalue
macro:#1->function with argument (#1)

The macro `\myVariable` is thus simply replaced by the string "myvalue". The macro `\myFunction` accepts an argument and places this behind the text in parentheses.

To implement more complex structures, other macros are often used within macro definitions. Then, multi-level expansion is necessary.

4.1 Multiple-level expansion

We use the following definitions as an example to illustrate how LaTeX handles commands.

```
\newcommand\one{a}
\newcommand\two{\one,b}
\newcommand\three{\one,\two,c}
```

Imagine each macro as a box with different content depending on its definition:

Without expansion, TeX sees only a token. If this is then to be processed further, the box is unpacked:

Since there is only the word "one" inside this macro, this process cannot be repeated. The macro is already fully and exhaustively expanded.

Compared to the once-expandable macro `\one`, `\three` can be expanded multiple times. See Figure 1 for all steps.

Expl3 allows for arguments to explicitly control how far boxes should be unpacked before further processing. This allows for exact control.

Accordingly, we can now distinguish the argument specifications. We see this using an expl3 construct directly: `\tl_to_str:n` directly prints the argument, as a string, into the document. To get variants of the expansion levels, one can use `\exp_args:No` or other expansion levels. Here the first argument is not expanded and the second one is expanded according to the argument:

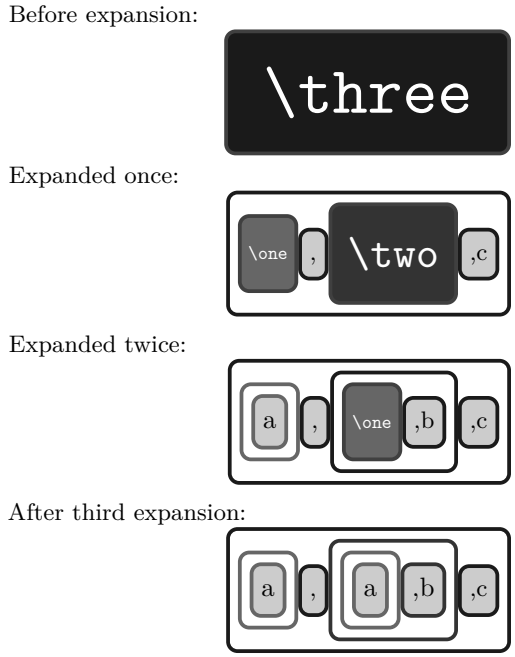

Figure 1: Visualization of expansion steps in boxes

```
\ExplSyntaxOn
unexpanded:\hfill
\tl_to_str:n { \three } \\
 expanded~once:\hfill
\exp_args:No \tl_to_str:n { \three } \\
fully~expanded:\hfill
\exp_args:Nf \tl_to_str:n { \three } \\
exhaustively~expanded~(x)\hfill
\exp_args:Nx \tl_to_str:n { \three } \\
exhaustively~expanded~(e)\hfill
\exp_args:Ne \tl_to_str:n { \three }
\ExplSyntaxOff
```

unexpanded:	\three
expanded once:	\one ,\two ,c
fully expanded:	a,\two ,c
exhaustively expanded (x)	a,a,b,c
exhaustively expanded (e)	a,a,b,c

This ability to control exactly in which order arguments of functions should be expanded makes it possible to look inside boxes before they are finally processed. Thus, for example, you can check in which format a date is passed and proceed accordingly:

```
\cs_new:Nn \__tugboattut_parse_date:n {
    % split at "-"
    \seq_set_split:Nnn \l_tmpa_seq
        { - } {#1}
    % more than one item
    % -> there was a dash
    \int_compare:nTF
        { \seq_count:N \l_tmpa_seq > 1 }
    {
        % assuming ISO format
        \seq_item:Nn \l_tmpa_seq { 3 } .
        \seq_item:Nn \l_tmpa_seq { 2 } .
        \seq_item:Nn \l_tmpa_seq { 1 }
    } {
        % alternative checks possible
        #1
    }
}
```

The macro now checks if the argument contains a hyphen. If it does, the date is interpreted as an ISO date (YYYY-MM-DD). Otherwise it is assumed that the date already has the format DD.MM.YYYY. (Further checking could be done.)

As an example, let's imagine the date that is passed at the beginning of the document via `\date` for the title line is to be processed with this. Internally, this value is stored in the command `\@date`. However, this macro does not contain a hyphen, which means that it has to be expanded first.

One way would be to use the `\exp_args:` command used above, but expl3 additionally provides a mechanism to create variants of a base command:

```
\cs_generate_variant:Nn
    \__tugboattut_parse_date:n
    { x }
```

Now the variant `\__tugboattut_parse_date:x` also exists. This allows the following construct:

```
\ExplSyntaxOn
\__tugboattut_parse_date:n { 23.06.2022 }
=
\__tugboattut_parse_date:n { 2022-06-23 }
=
\__tugboattut_parse_date:x
    { \use:c { @date } }
\ExplSyntaxOff
```

23.06.2022=23.06.2022=23.06.2022

5 Summary/outlook

LaTeX3 or expl3 extends and simplifies the ability to write more flexible macros despite — or maybe even because of — its somewhat weird syntax. In addition to the basic data type of "Token", structured data types are introduced, and a differentiation is made between data processing and data storage. Moreover, expl3 makes the control of macro expansion much more transparent.

It is thus possible to process different contents automatically and to pack significantly more functionality into a macro depending on the arguments and their structure. Furthermore, the uniform structure of the interfaces makes it easier to read and understand code by other authors.

Very many helper macros, which have been defined in many packages again and again, are obsolete, since there now are functions within the LaTeX kernel to replace them. Whole groups of packages are replaced by LaTeX3 standard modules, one of the most common examples being the key–value parsing packages [2]. In the long run, this — along with other extensions to the kernel — will reduce conflicts between packages and thus make the overall LaTeX system even more stable and increase the usability for the end user.

This article provides only a very small glimpse of the possibilities. Although more or less anything could be done in classic LaTeX, it often required low-level patches and hacking. Expl3 has made it much easier for me to do programming tasks directly in LaTeX.

I sincerely hope this article can help to reduce some confusion and fear, on the road to more expl3.

`\prg_do_nothing:` or just `\relax`.

References

[1] J.L. Braams, J. Bezos. Babel: Localization and internationalization. `ctan.org/pkg/babel`

[2] CTAN. `keyval` topic. `ctan.org/topic/keyval`

[3] A. Hendrickson. The joy of `\csname... \endcsname`. *TUGboat* 33(2):219–224, 2012. `tug.org/TUGboat/tb33-2/tb104hendrickson.pdf`

[4] LaTeX Project. The LaTeX3 interfaces. `mirrors.ctan.org/macros/latex/contrib/l3kernel/interface3.pdf`

[5] LaTeX Project. The expl3 package and LaTeX3 programming. `mirrors.ctan.org/macros/latex/contrib/l3kernel/expl3.pdf`

[6] J. Wright. Registering expl3 module[s]. `texdev.net/2012/11/04/registering-expl3-module/`

◇ Marei Peischl
Gneisenaustr. 18
20253 Hamburg
Germany
`marei (at) peitex dot de`
`https://peitex.de`