Functions and expl3

Enrico Gregorio

Abstract

In this tutorial we discuss expl3 functions, their role, definition and variants, also touching on variables.

1 Introduction

The term *function* is not used in standard LaTeX, but is a very important concept in the expl3 language. The language itself had no precise name until a couple of years ago, when it was eventually decided that its name would be expl3 just like the package that provided it (now merged in the LaTeX kernel).

In the language, care is taken to distinguish between *variables* and *functions*. Variables store some value that can change during the LaTeX run, whereas functions perform some action.

A special kind of variable is the *constant*, whose value is supposed not to change during the run. Well, the name seems to conflict with the nature, but mathematicians are used to this kind of stretched terminology: you who are not a mathematician, don't worry and carry on, just smile at mathematicians' bizarre way of thinking.

We'll be mostly interested in functions, but variables can be the staple food of functions, so we'll also need to know a bit about them.

Let me give an example using legacy concepts. The standard document classes, and most nonstandard ones as well, have the *command* `\title`. How does this work? This very paper has

```
\title{Functions and \expliii}
```

at its start. When LaTeX processes this instruction, it will do something like

```
\gdef\@title{Functions and \expliii}
```

so it will be able to use `\@title` when doing its typesetting job related to `\maketitle`.

There is a big conceptual difference between `\title` and `\@title`. The former performs an action, the latter is simply a container. In expl3 terms, the former will be a function, the latter a variable: this function's action is to store a value in the specified variable.

At the user level the distinction is blurred; with

```
\newcommand{\CC}{\mathbb{C}}
```

is `\CC` be a function or a variable? Fortunately, it's not important to decide, because this is essentially a user's *shorthand* and at this level the distinction is almost irrelevant.

Simultaneously published in Italian for the GuIT 2020 conference, guitex.org.

The issue comes up when *programming*. In 'correct' LaTeX programming we have a user level command which calls a function that performs an action:

```
\NewDocumentCommand{\title}{m}
 {
  \example_title:n { #1 }
 }
%
\tl_new:N \g_example_title_tl
%
\cs_new_protected:Nn \example_title:n
 {
  \tl_gset:Nn \g_example_title_tl { #1 }
 }
```

Then the user level command `\maketitle` will use the value stored in the variable, via other functions.

This (imaginary) example shows many of the concepts we'll be discussing. We define a user-level command in terms of a function; this function has one argument (the given title) and its job is to set the value of a particular variable to the specified value. The variable has been declared in advance:

- `\title` is a user-level command; these are not the subject of this paper;
- `\g_example_title_tl` is a variable, defined using `\tl_new` (*tl* = token list);
- `\example_title:n` is a function, defined using `\cs_new_protected` (*cs* = control sequence).

We'll be discussing all of this in detail.

2 Naming conventions

A common problem with TeX is that it has no concept of *namespace*, which only became common in computer science circles much later. Name conflicts were frequent in the olden days of LaTeX, and such conflicts still appear now and then. It might be appealing for a package to use `\@x` and `\@y` for coordinates, but package authors should be aware that if a simple name appeals to them, other authors have probably thought the same.

In expl3, variables should have a name of the form

`\l_`⟨*prefix*⟩`_`⟨*proper name*⟩`_`⟨*type*⟩
`\g_`⟨*prefix*⟩`_`⟨*proper name*⟩`_`⟨*type*⟩
`\c_`⟨*prefix*⟩`_`⟨*proper name*⟩`_`⟨*type*⟩

where the distinct parts are *important* and *necessary*:

- `l`, `g` and `c` declare that the variable is local, global or constant, respectively;
- ⟨*prefix*⟩ should be a unique string of letters for the package we're writing or the code in the document;
- ⟨*proper name*⟩ is an arbitrary string of letters possibly split into parts separated by an underscore;

- ⟨*type*⟩ is the type of variable.

The most common types of variables are:

- `tl`, for *token list*;
- `seq`, for *sequence*;
- `clist`, for *comma list*;
- `prop`, for *property list*;
- `int`, for *integer*;
- `dim`, for *dimension*;
- `box`, for *box*;
- `fp`, for *floating point*.

There are several others, but as the purpose of this tutorial is to talk about functions, I'll skip the more esoteric ones for now, only touching them when need comes.

Function names are similar:

\⟨*prefix*⟩_⟨*proper name*⟩:⟨*signature*⟩

The ⟨*prefix*⟩ and ⟨*proper name*⟩ are the same as before, but the ⟨*signature*⟩ must be explained. It can be an arbitrary string of characters among

```
c e f n N o v V w x T F
```

Each character given, except `w`, denotes an argument to the function. We'll be going into details soon.

Mathematical functions can depend on one or more arguments (well, also zero, but then they're constant functions) and the same is true for expl3 functions. The purpose of the signature is to precisely specify how many arguments the function depends on and their type. For instance, the commonly used function

```
\seq_set_split:Nnn
```

takes three arguments, one of type `N` and two of type `n` *in that order*. An argument of type `N` should be a single token, the nature of which depends on the function; in the above case, it should be a sequence variable. An argument of type `n` should be a braced list of tokens. In our example, the title of the paper is an `n`-type argument to `\title`. This is a bit stretched, but should explain the concept.

The `w` type is an exception, because it specifies nothing except that the arguments to the function are *weird*, and one must refer to the package/code documentation in order to know how many there are and what syntax they have. Generally speaking, `w`-type arguments should only appear in low-level functions.

A call to the previously mentioned function might be something like

```
\seq_set_split:Nnn
  \l_example_test_seq
  { || }
  { a || b || c }
```

(more likely on one line in a source file; split here because of the paper's formatting). It doesn't matter now to know what this code does; `seq_set_split` is usually called as part of other processing and receives the arguments from other calls. The important thing is to see that the arguments follow the naming convention: the first one is a single token, the other two are braced lists of tokens.

A function can have no arguments, but the colon is still required. Although TEX will not balk if you define a function with a nonconforming name, sticking to the convention will help to avoid conflicts and to have more easily parsable code. A typical function with no arguments is `\scan_stop:`, which is nothing other than our old friend `\relax`. Maybe the expl3 name is less poetic, but it expresses what the function's main purpose is.

3 Defining functions

There are many kernel functions which define functions. All of them share the `cs` prefix. The main ones are

```
\cs_new:Nn
\cs_new_protected:Nn
```

We'll discuss the others later. According to the naming conventions given, we can see that both have two arguments: the first argument is a single token, the name of the function to be defined, the second being the *replacement text*, that is, the code that will be substituted to the function's call.

While expl3 tries hard to emulate a functional language, it is still implemented in TEX, which only knows primitives, macros and registers. This fact needs to be kept in mind when programming. On the other hand, the new language makes for simpler constructions, avoiding the clumsy (or fun, depending on the programmer's attitude) chains of `\expandafter` or `\noexpand` often seen in traditional TEX, understanding which is often quite hard.

A simple example is the internal function for managing the document's title, which we saw earlier:

```
\cs_new_protected:Nn \example_title:n
 {
  \tl_gset:Nn \g_example_title_tl { #1 }
 }
```

Since the function's name has signature `:n` (strictly speaking, the colon is not part of the signature, but it's convenient to use it as a marker), expl3 knows that the replacement text can use `#1` to refer to the argument supplied at call time.

Why are we using the `protected` instruction and not the simpler `cs_new`? Because our function will *set* the value of a variable. This is something

that for years has frustrated a horde of LaTeX programmers and has required the distinction between robust and fragile commands. Nowadays the issue is less relevant because almost all fragile commands have been 'robustified', but it can still bite.

What is the problem? If we define, in legacy LaTeX, something like

```
\newcommand{\foo}[1]{%
  \renewcommand{\baz}{#1}%
}
```

which is the common way to store a value into a macro, and then somehow `\foo` ends up in `\write` or `\edef`, even under their wrappers `\protected@write` or `\protected@edef`, a long list of error messages appears. The traditional way of avoiding this is to use `\DeclareRobustCommand` instead of `\newcommand`.

Any function that works by setting variables or calling other protected functions should generally be protected itself. In case of doubt, protect.

Beware! The signature of the function to be defined can only consist of the characters `N` or `n`. Well, `T` and `F` are also allowed, but these are a special topic that we'll touch later on. How do the other characters listed above get into signatures? This is a good question!

3.1 Generating variants

Suppose we're doing a general purpose function for setting `tl` (token list) variables to contain some material we need to use at later points. The user interface would utilize `\setvar` for storing the value and `\usevar` for delivering the value.

We face a problem: how can the user specify the name of a variable inside the document where the expl3 names are not allowed? Indeed, in normal text, the underscore cannot be used in a command name, so something like

```
\setvar{\l_example_var_a_tl}{something}
```

would bomb out. We'd like instead that the user types in

```
\setvar{a}{something}
\usevar{a}
```

(at different points of the document, of course). Let's proceed at a slow pace. We'll define the user interface afterwards. First we define a function that allocates a variable and stores a value in it; then a function that delivers the contents of a variable:

```
\cs_new_protected:Nn \example_setvar:Nn
 {
  \tl_clear_new:N #1
  \tl_set:Nn #1 { #2 }
 }
\cs_new:Nn \example_usevar:N
 {
  \tl_use:N #1
 }
```

The `tl_clear_new` instruction clears a possible preceding value or allocates a new variable. Then the `tl_set` function does the setting job; it's not protected because it does no dangerous processing. However, this does not solve the problem we face. Here's where the concept of *variants* comes into the scene. We do

```
\cs_generate_variant:Nn \example_setvar:Nn
  { cn }
\cs_generate_variant:Nn \example_usevar:N
  { c }
```

which effectively defines two *new* functions named

```
\example_setvar:cn
\example_usevar:c
```

What does `c` mean? It means that the new functions expect a braced argument, which it will build a command name from (in this case the name of a variable, in other cases it could be the name of a function). So now we can define the user interface:

```
\NewDocumentCommand{\setvar}{mm}
 {
  \example_setvar:cn
    { l_example_var_#1_tl }
    { #2 }
 }
\NewExpandableDocumentCommand{\usevar}{m}
 {
  \example_usevar:c { l_example_var_#1_tl }
 }
```

Sites on LaTeX are plagued with questions about code doing nasty things such as `\def\c{something}`, asking why this breaks.

With the approach just outlined we set up a *namespace* for our variables, which can just be called by their 'outer' name, leaving to the implementation the details about how to avoid conflicts.

[We could certainly define the user level commands in legacy LaTeX. A typical implementation would be

```
\newcommand{\setvar}[2]{%
  \expandafter
    \def\csname example@var@#1\endcsname{#2}%
}
\newcommand{\usevar}[1]{%
  \csname example@var@#1\endcsname
}
```

I won't quarrel with people maintaining this is simpler. But I'll remain with my opinion that it isn't.]

Let's add something to the game: now we want to allow the user to copy the value of a variable into another, say by doing `\copyvar{b}{a}`, where `b` is the new one and `a` the existing one. We only need a new variant, namely

```
\cs_generate_variant:Nn \example_setvar:Nn
  { cv }
```

and then define the user interface with

```
\NewDocumentCommand{\copyvar}{mm}
 {
  \example_setvar:cv
    { l_example_var_#1_tl }
    { l_example_var_#2_tl }
 }
```

We now have at our disposal another function, namely `\example_setvar:cv`, which takes two braced arguments. The second one is scanned like `c`, producing a symbolic token which should be a variable of some kind and then will deliver *the contents* of the variable as a braced argument to the main function.

[I leave as an easy exercise on `\expandafter` how to do this in legacy LaTeX programming (hint: use `\let` and two `\expandafter`s, cleverly positioned).]

It's not necessary to write two distinct calls for defining the variants, we can create them both at once with:

```
\cs_generate_variant:Nn
  \example_setvar:Nn
  { cn, cv }
```

The `v` variant is a special case of the `V` variant, which is almost the same, but the capital letter reminds us that a single token (a variable's name) should be used without braces. Suppose we have a function that does something with its argument:

```
\cs_new:Nn \example_foo:n { -- #1 -- }
```

(just some nonsense to illustrate the concept). However, in some cases we need to pass the function something that has been stored in a `tl` variable: nothing simpler, because we can do

```
\cs_generate_variant:Nn \example_foo:n
  { V }
```

allows us to do

```
\example_foo:V \l_tmpa_tl
```

If, say, `\l_tmpa_tl` has been set to contain `abc`, then calling `\example_foo:V \l_tmpa_tl` is exactly the same as doing `\example_foo:n {abc}`.

Bear in mind that variants do not come into existence without first generating them. Kernel functions come predefined with several variants that have proven to be useful; they're listed together with the main function in `interface3.pdf`. What if we don't know whether a variant has already been defined? No problem at all! The generation will be silently ignored and, even if it weren't, there should be no problem either, because variants are generated in a uniform way.

We could avoid generating variants. For instance the job of `\example_setvar:cv` could be accomplished by

```
\exp_args:Ncv \example_setvar:Nn
```

(which is actually how the variant is defined), but there's no point in complicating our life this way. Modern TeX and friends' implementations have lots of memory available, and the times when memory was in *very* short supply and tricks saving just a few tokens in order to spare memory were necessary are only remembered by old dinosaurs like Frank, Chris, David and myself. I remember well the time I saw the dreaded "This can't happen" error message because I was using PiCTeX.

Let's go back to theory. Are there variants to the function-defining functions? Yes, of course there are! There are times when we want to define a function whose name is decided at runtime. The example below is a bit silly, but should give the idea: the call

```
\cs_new:cn { example_foo:n } { -- #1 -- }
```

is the same as the definition of `\example_foo:n` above, because the name, including the signature, will be formed before the underlying `\cs_new:Nn` function does its job. One can use anything inside the braces corresponding to the `c` argument type, so long as the final result, after *full expansion*, just consists of characters. Oh! Expansion! What is it? Be patient. First there are other bits to discuss.

3.2 Local, global, and ⟨*extra*⟩s

Everybody should know about *local* and *global*. In LaTeX, if we perform some command definition inside an environment, it is local to the environment and will disappear when it ends. Other assignments of meaning are instead global: operations on counters, for instance. We don't need a full discussion on local versus global, but there are aspects of the problem that are relevant for functions.

All `\cs_new`⟨*extra*⟩`:Nn` declarations are always *global*. Even if performed inside a group, their effect will also be carried on at the outer levels. Further, they will check whether the function is already defined and issue an error message if so. The ⟨*extra*⟩ part will be described next. Variable allocation (not setting) is also always global: the instruction

```
\tl_new:Nn \l_example_foo_tl
```

defines the variable at all levels and will balk if the variable already exists.

However, sometimes we need *locally defined* auxiliary functions, that have no fixed meaning and need to be redefined according to the context. For these there is the family:

`\cs_set`⟨*extra*⟩`:Nn`

The syntax is exactly the same as with `\cs_new`, as are the available variants.

Which one to prefer? The `new` or the `set` family? The answer is easy: the former, unless the function is *required* to change its definition according to the context. Rarely, if ever, will a high level function be defined with `set`.

Thus, the very nature of the language invites programmers to code in layers. For instance, we could have done:

```
\NewDocumentCommand{\setvar}{mm}
 {
  \tl_clear_new:c
    { l_example_var_#1_tl }
  \tl_set:cn
    { l_example_var_#1_tl }
    { #2 }
 }
```

without defining `\example_setvar:Nn` at all. I discourage this kind of programming: our code should sit *on top* of expl3 and provide APIs for the programmer to employ. As I said before, there is no point in sparing a function, even more so if we consider that once our API is available, we can easily define variants thereof for particular jobs.

What's the complete list of ⟨*extra*⟩s available after `\cs_new` or `\cs_set`? Here it is:

```
\cs_new:Nn
\cs_new_protected:Nn
\cs_new_nopar:Nn
\cs_new_protected_nopar:Nn
\cs_set:Nn
\cs_set_protected:Nn
\cs_set_nopar:Nn
\cs_set_protected_nopar:Nn
\cs_gset:Nn
\cs_gset_protected:Nn
\cs_gset_nopar:Nn
\cs_gset_protected_nopar:Nn
```

In all cases, the first argument is the name of the function to be defined and the second argument is the replacement text.

You probably already have an idea of what `protected` does: it arranges things so that the function is not expanded when *full expansion* is enforced. In particular, a `protected` function cannot be used in a `c`-type argument, because it wouldn't be expanded and it is not a character.

The `nopar` variety disallows `\par` tokens in the function's argument (when called). Unless we're dealing with special situations where `\par` does not make sense in a function's argument, there is no need to use it.

The `gset` family is almost the same as `new`, but no check is performed about the function being defined.

[For the TEX gurus that are reading these notes, `new` and `gset` use `\gdef`, whereas `set` uses `\def`; the `\long` prefix is added except with `nopar`.]

What are the available variants? Here's the complete list for all of the above functions:

```
Nn  cn  Nx  cx
```

What's this mysterious `x`? It's intended to bring to mind *fully e**x**panded*. There will be a section later on the topic.

3.3 Parameters

Some people will now be complaining that they have seen different ways to define functions and they're right. There *is* a whole new family like the one above but where the signature has a strange `p` between `N` and `n`, namely

```
\cs_new:Npn
\cs_new_protected:Npn
\cs_new_nopar:Npn
\cs_new_protected_nopar:Npn
\cs_set:Npn
\cs_set_protected:Npn
\cs_set_nopar:Npn
\cs_set_protected_nopar:Npn
\cs_gset:Npn
\cs_gset_protected:Npn
\cs_gset_nopar:Npn
\cs_gset_protected_nopar:Npn
```

The `p` is a reserved argument type just for these functions and variants thereof: all of them come along with the variants

```
Npn  cpn  Npx  cpx
```

and stand for ***p**arameter text*. The two lines below are completely equivalent:

```
\cs_new:Nn \example_usevar:n {...}
\cs_new:Npn \example_usevar:n #1 {...}
```

The same for the other functions. In the second instance, the *parameter text* is explicitly written out. Remember that when the expl3 programming conventions are in force, spaces are ignored, so for two parameters we can have

```
\cs_new:Nn \example_foo:nn {...}
\cs_new:Npn \example_foo:nn #1 #2 {...}
\cs_new:Npn \example_foo:nn #1#2 {...}
\cs_new:Npn \example_foo:nn #1 #2{...}
\cs_new:Npn \example_foo:nn #1#2{...}
```

and the last four lines are completely equivalent. Personally, I prefer the first way that's 'more logical'; others prefer the second way. Beware! The second way doesn't check for consistency of the signature with the parameter text and it even allows for 'wrong'

signatures, but this fact should not be exploited: LaTeX will not balk if you type

```
\cs_new_protected:Npn
  \example_setvar:cn #1 #2
  {...}
```

but this doesn't mean that you can avoid the two-step procedure of first defining `\example_setvar:Nn` and then creating the variant. Doing the definition this way is *wrong*. The second family of functions even allows for no signature at all, actually, so they can be used for defining user level commands, although the path with `\NewDocumentCommand` (or siblings) is recommended.[1]

The `p` way is necessary when the parameter text is 'nonstandard', in the sense that we're defining a function with delimited arguments; in this case, the signature should be `w`. If we want a function that sets three variables to the year, month and day given an ISO-format date such as 2020-10-15, we can do

```
\int_new:N \l_example_year_int
\int_new:N \l_example_month_int
\int_new:N \l_example_day_int
\cs_new_protected:Nn \example_setdate:n
 {
  \__example_setdata:w #1 \q_stop
 }
\cs_new_protected:Nn
  \__example_setdate:w #1-#2-#3 \q_stop
 {
  \int_set:Nn \l_example_year_int { #1 }
  \int_set:Nn \l_example_month_int { #2 }
  \int_set:Nn \l_example_day_int { #3 }
 }
```

Here I introduce another useful convention: if the ⟨*prefix*⟩ is preceded by a double underscore, the function is considered lower level than the others and should *never* be called outside its specific uses by standard functions (without the double underscore). The idea is that the standard functions are the 'programmer's interface', whereas the others are auxiliary whose actual implementation should not concern the programmer. The distinction when writing personal code is not so important, but it is crucial for package writers. Standard functions (without the double underscore) *can* be used by other packages, whereas one should not count on the lower level ones (with the double underscore) to even be defined in later versions of the package.

This should clarify why the code above splits the work into two levels; we have the high level function `\example_setdate:n` function which relies on a lower level one to do the dirty work. Maybe the package writer will discover a better way to accomplish the task, but this would only influence the lower level and not the main function, which will be possible to call forever. Maybe the definition of `\example_setdate:n` will change in the future, but this won't affect code that uses it.

[1] expl3 can also be used with plain TeX, and in this case this is the only way to define user level commands.

4 Expansion

There can be no full understanding of TeX without some knowledge on how expansion works. In functional programming languages, if `g` is a function of one variable returning an array of three data, whereas `f` is a function of three variables, a call

```
f(g(x))
```

would be permissible (from a mathematical point of view, at least, maybe a particular functional language requires some tweak). This is not the same in TeX, which goes from outside to inside, rather than conversely.

If we have a one-argument `\example_a:n` function that returns three braced lists of tokens, and another function `\example_b:nnn` that takes three arguments, a call such as

```
\example_b:nnn { \example_a:n {x} }
```

would fail miserably. That's how TeX works and no clever code can change this. The outer function will look for three arguments and the given braced list of tokens is just one.

As an example of how to handle this, suppose that `\example_a:n` can be fed a date in ISO format and from `2020-10-15` it returns `{2020}{10}{15}`, whereas `\example_b:nnn` takes three arguments and produces a date in a different format, say "day 'name of the month', year": in this case it should output "15 October, 2020".

We need an indirect approach, in order to allow feeding an ISO date to the general function outputting the date in that format. Let's see how the general function might be defined:

```
\cs_new:Nn \example_date:nnn
 { % #1 = year, #2 = month, #3 = day
  #3~
  \int_case:nn { #2 }
   {
    {1}{January}
    {2}{February}
    ...
    {12}{December}
   }
  ,~#1
 }
```

The `\int_case:nn` function examines its first argument against the list given as its second argument (the code is incomplete, but you can guess how to

fill it in) consisting of pairs of braced items; the first contains an integer, the second something to output when a match is found. The ~ here is not a nonbreaking space in the expl3 programming environment, but a normal space.

We also need a function that is given a date in ISO format (2020-10-15) and returns it split into the three constituent parts, {2020}{10}{15}:

```
\cs_new:Nn \__example_isodate:n
 {
  \__example_isodate:w #1 \q_stop
 }
\cs_new:Npn
  \__example_isodate:w #1-#2-#3 \q_stop
 {
  {#1}{#2}{#3}
 }
```

The input to the second function is split at the hyphens and at the terminator, so we're using *delimited arguments*, a detail I'll skim over. Here, we just need to know that the call

```
\__example_isodate:n {2020-10-15}
```

will *eventually* return {2020}{10}{15} to the input stream.

How do we combine these? There are several ways, but all of them require understanding the concept of *full expansion*. TEX only knows macros; when it finds one, it knows how many arguments it takes and looks for them in the input stream; upon finding them, it replaces the whole sequence of tokens so found with the replacement text of the macro.

The main problem is that most of the time we don't know how many steps of expansion it will take to get from __example_isodate:n {2020-10-15} to {2020}{10}{15}. In this case it would be easy to count them, but this is just a simple example. If we knew, a suitable chain of \expandafter commands would suffice, but this is prone to errors and inconsistencies if the implementation changes.

What we'd like is for __example_isodate:n to go all the way down to the final result in one swoop. Here's a way:

```
\exp_last_unbraced:Ne
  \example_date:nnn
  { \__example_isodate:n {2020-10-15} }
```

What \exp_last_unbraced does is *fully* ***e****xpand* its second argument and return the result in the input stream with no braces around it.

There are other ways. One is to define a new helper function:

```
\cs_new:Nn \example_date:n
 {
  \__example_date:Ne
    \example_date:nnn
    { \__example_isodate:n { #1 } }
 }
\cs_new:Nn \__example_date:Nn
 {
  #1 #2
 }
\cs_generate_variant:Nn
  \__example_date:Nn { Ne }
```

upon which \example_date:n {2020-10-15} would produce the intended result.

There are still more ways, but here the idea is to present how we can exploit the full expansion variants.

In sum, there are three kinds of them, namely e, x and f. The last of these is the most restricted, because it performs recursive expansion of the tokens as soon as they are placed in the input stream and ends at the first unexpandable token it finds. Notwithstanding this limitation it has several uses.

The x type is nowadays less important because all TEX engines supporting LATEX have the primitive \expanded, which is itself expandable. Only Knuthian TEX (the engine that one launches with tex on the command line) lacks it, since it is kept with no extensions, according to Knuth's desiderata.

[What does \expanded do? It is essentially like \edef, with the difference that no macro is defined. The argument to it is subject to full recursive expansion which *doesn't* stop when an unexpandable token is found, but just jumps over it and continues from the next token, until exhausting the supplied token list. The result is then placed on the input stream (without braces).]

4.1 Full expansion with e

The e argument type tells LATEX to first fully expand the given argument and then supply the result to the original function. This is very important if the argument contains a variable which we want to deliver the value of at call time.

We can see an example in a post on TEX.StackExchange.[2] The question is about adding to endnotes, via the endnote package, the page number where the endnote actually appears. We want to use \pageref through an automatically supplied label:

```
\NewDocumentCommand{\MyEndNote}{m}
 {
  \polyv_myendnote:ne
    { #1 }
    { \int_eval:n { \arabic{endnote}+1 } }
 }

\cs_new_protected:Nn \polyv_myendnote:nn
 {
  \endnote
```

[2] https://tex.stackexchange.com/a/438715/. The code there uses f, because e wasn't available yet.

```
   {
    #1~(page\nobreakspace
    \pageref{#2:endnote})
   }
  \label{\arabic{endnote}:endnote}
 }
\cs_generate_variant:Nn
  \polyv_myendnote:nn
  { ne }
```

What's the problem to be solved? The endnote number is incremented after the endnote is processed, so a `\label` command gets this new number. So we can't use

```
\pageref{\arabic{endnote}:endnote}
```

because this would refer to the *previous* endnote. Thus the internal function uses `e` expansion in order to generate the successor to the current value of the `endnote` counter. Without this full expansion, all the endnotes declared with `\MyEndNote` would contain the equivalent of

```
\pageref{%
  \int_eval:n{\arabic{endnote}+1}:endnote
}
```

and so the final result would be undefined cross references, because `\arabic{endnote}` would always expand to the *final* value of the counter. For instance, if the last endnote was number 10, we'd end up with `\pageref{11:endnote}`. Instead, with full expansion, the current value is used and passed to the main function. At the first endnote, the counter has value 0, so the end result is the same as

```
\endnote{The text of the endnote
  (page\nobreakspace\pageref{1:endnote})}
\label{1:endnote}
```

We could as well have used `f` or `x` for this particular application, but `e` is the most efficient of the lot. The difference from `x` is that functions using `x` are *not* expandable, so they have to be of the `protected` kind. Indeed the process is a two-step one: first a temporary token list is set using

```
\tl_set:Nx \l__exp_internal_tl {...}
```

(which internally uses good old `\edef`).

The introduction of `e`-type full expansion has been a significant step forward, because it allows for things that were almost impossible before.

However, as seen above, there are uses for `x`: for instance there is no `\cs_new:Ne` variant and it would be *less* efficient than `\cs_new:Nx` (which is just `\edef`).

5 Another essential variant: `V` for variables

In the list of argument types above there are `V` and `v`, which have been touched upon briefly. Now it's time to discuss the former in greater detail.

Type `v` is nearly the same as `V`; it just adds the ability of building the name of the variable by supplying data at runtime. The main one is `V`.

Again, let's suppose we have our favorite function that splits an ISO date into components and outputs the date in another format, and that it's named `\example_date:n`. (Its implementation is irrelevant.)

Suppose now we have a date stored in a `tl` variable. Since this is just a macro under cover, at the beginning of expl3, the way to process this was

```
\cs_generate_variant:Nn
  \example_date:n { o }
```

to be called like

```
\example_date:o { \l_tmpa_tl }
```

but this is bad because it depends on the knowledge of the implementation of `tl` variables. Also, it is not generalizable to other kinds of variables. The `o` variants do a single expansion step in the braced argument; while this works with the straightforward implementation of token list variables it would fail spectacularly with `fp` variables (which contain floating point numbers).

The best method is to do

```
\cs_generate_variant:Nn
  \example_date:n { V }
```

to be called like

```
\example_date:V \l_tmpa_tl
```

This will deliver the contents of the variable, suitably braced, to the main function. So if we did

```
\tl_set:Nn \l_tmpa_tl { 2020-10-15 }
```

then the call `\example_date:V \l_tmpa_tl` would be equivalent to `\example_date:n {2020-10-15}`.

An example with a different kind of variable, namely `int`. We want to feed in such a variable and the result should pad it with zeros to get four digits:

```
\cs_new:Nn \example_pad:n
 {
  \prg_replicate:nn
    { 4 - \tl_count:n { #1 } }
    { 0 }
  #1
 }
\cs_generate_variant:Nn
  \example_pad:n { V }

\int_set:Nn \l_tmpa_int { 43 }
\example_pad:V \l_tmpa_int
```

This will print 0043. This may seem of academic interest only, but with the sibling `v` type, we can transform this into

```
\cs_generate_variant:Nn
  \example_pad:n { v }
\example_pad:v { c@page }
```

knowing that `\c@page` is the LaTeX name of the register containing the page number and an application can be immediately thought of. This exploits the fact that standard TeX counters are exactly like expl3 `int` variables. Using the `V` or `v` variants we're passing the main function the actual value as a list of digits, so we can count it, which would be impossible with the 'abstract value'.

What variable types can be used this way? Several: `tl`, `int`, `fp`; also `clist` and others more esoteric. Essentially, all variables that can deliver some sensible output. This cannot be expected from `seq` or `prop` variables and, indeed, using those will crash.

I've sometimes found it useful to define the `\cs_set:NV` variant for `\cs_set:Nn` in order to use a `tl` variable where the desired replacement text has been stored and modified via some regular expression replacement.[3] By the way, it is not possible to have a `\cs_set:NpV` variant, because the generator `\cs_generate_variant:Nn` can only accept a function with a signature consisting of `N` or `n` characters.

6 True or false?

There are two other interesting argument types: `T` and `F`, and the title of the section should suggest that they're connected with truth and falsehood.

Exactly so! They are argument specifiers in the signature of *conditional* functions. Example:

```
\int_compare:nTF
```

is a function that takes three standard braced arguments; the first is a numeric relation between integers to test, the second the code to execute if the relation is true, the third the code to execute if the relation is false. So

```
\int_compare:nTF { 0<1 } { A } { B }
\int_compare:nTF { 0>1 } { A } { B }
```

will result in printing A and B respectively. So, why isn't it more simply `\int_compare:nnn`? Indeed, it used to be this way in the first versions of expl3, but it was realized that having different argument specifiers is handier, because we can also have

```
\int_compare:nT
\int_compare:nF
```

when we have nothing to execute for the false or true branch respectively. Of course

```
\int_compare:nF { ⟨relation⟩ } { B }
\int_compare:nTF { ⟨relation⟩ } { } { B }
```

are completely equivalent, but the former shows more clearly that we want to do nothing if the ⟨*relation*⟩ turns out to be true. With the `:nnn` signature, empty arguments would be always required. Also, the presence of either `T` or `F` (or both) immediately alerts us that the function is a conditional.

All kernel conditional functions are available with ending `TF`, `T` or `F`; some conditional-like functions even have the version with neither. For instance we can see in `interface3.pdf` that there is `\str_case:nn`, but also `\str_case:nn`*`TF`*. The strange-looking *`TF`* means that all three combinations `TF`, `T` and `F` available.

Why such a "pseudo-conditional"? The function `\str_case:nn` is not a conditional (being related to letter case), but we can use the extended version to output something if there is, or is not, a match. The version which is most likely to be used is `\str_case:nnF` to output something in case of no match, maybe an error message or a default output.

To generate proper conditionals, variants should be defined with

```
\prg_generate_conditional_variant:Nnn
```

rather than with `\cs_generate_variant:Nn`. For instance, if we plan to store some ⟨*relation*⟩s for `\int_compare:n`*`TF`* into `tl` variables, the correct way to generate the variant is

```
\prg_generate_conditional_variant:Nnn
  \int_compare:n { V } { p, TF, T, F }
```

This will at once generate the variants

```
\int_compare:VTF
\int_compare:VT
\int_compare:VF
```

as well as the 'predicate form'

```
\int_compare_p:V
```

to be used in boolean expressions. But this is outside the scope of the present paper.

◇ Enrico Gregorio
Dipartimento di Informatica,
Università di Verona
enrico dot gregorio (at) univr
dot it

[3] https://tex.stackexchange.com/a/355576/