LaTeX3 programming: External perspectives

Joseph Wright

Abstract

The current experimental LaTeX3 packages provide a new, documented programming interface for TeX. The key ideas implemented in this new interface are highlighted in this article.

1 Introduction

Modifying the behaviour of LaTeX 2ε often requires a combination of user macros, internal LaTeX macro and TeX primitives. This makes even trivial modifications of document layout potentially difficult, even for the experienced LaTeX user. The differing syntax used by TeX primitives and the LaTeX kernel only add to the confusion here.

The first step to develop a new LaTeX kernel is therefore to address how the underlying system is programmed. Rather than the current mix of LaTeX and TeX macros, the experimental LaTeX3 system provides its own consistent interface to all of the functions needed to control TeX. A key part of this work is to ensure that everything is documented, so that LaTeX users can work efficiently without needing to be familiar with the internal nature of the kernel or with plain TeX.

The current kernel also suffers from the mixing of design commands with structural code. Thus changing a layout element often requires modifying a kernel code block (or loading a package which provides an interface to achieve this). The second challenge for LaTeX3 is therefore separation of the basic tools of the kernel from the design of documents.

This short overview article highlights the key developments to date in LaTeX3. It is based on my own experience working with the new tools for writing packages, and a talk given recently to the UK TeX Users Group.

2 The components of LaTeX3

Currently, the experimental LaTeX3 packages are designed to be used "on top of" LaTeX 2ε. This avoids needing to wait for the entire kernel to be finished before testing what is written.

The most developed part of the code is the expl3 ("experimental LaTeX3") bundle, the core of the new kernel providing the new programming interface. The new language is fully documented in the file `source3.pdf`, which contains some notes for the experienced (La)TeX programmer.

Built on top of expl3 is the xparse package. This is meant to be a "bridge" between the internal and user parts of the new kernel. The xparse package is used to create new user macros, in a much more controlled way than is possible using `\newcommand`.

More experimental than xparse are various other "xpackages". These are designed to explore new approaches to layout and document design for LaTeX3.

The most complete part of LaTeX3 is the expl3 bundle. The rest of this article is focussed mainly on the new internal syntax introduced in expl3.

3 A new internal syntax

LaTeX3 does not use `@` as a "letter" for defining internal macros. Instead, the symbols `_` and `:` are used in internal macro names to provide structure. In contrast to the plain TeX format and the LaTeX 2ε kernel, these extra letters are used only between parts of a macro name (no strange vowel replacement).

LaTeX3 separates macros which do something (functions) from ones which only store data. The general form of an internal function in LaTeX3 is \⟨*module*⟩_⟨*function*⟩:⟨*arg-spec*⟩.

- The ⟨*module*⟩ prefix is applied to almost all macros. For a package, it will typically be the package name; the kernel is split into a number of modules, each with its own name.
- The name of the ⟨*function*⟩ should give a good description of what it does: this may contain one or more `_` characters to divide the name into logical units.
- The concept of the ⟨*arg-spec*⟩ is potentially confusing to existing (La)TeX programmers. This *argument specifier* describes the arguments expected by the function. In most cases, each argument is represented by a single letter. The letter, including its case, conveys information about the type of argument required. The use of the ⟨*arg-spec*⟩ is illustrated later in this article.

3.1 Primitives renamed

All of the TeX primitives are given new names by expl3, although many are not intended to be used outside of the LaTeX3 kernel. Instead, a number of LaTeX wrappers for primitives are provided, so that the argument syntax is consistent.

At the most basic level, the `\fi` primitive becomes `\fi:`, indicating no arguments are required.

A more complex example is `\ifdefined` (an ε-TeX primitive), which becomes `\if_cs_exist:N`.

```
\if_cs_exist:N \Macro_One
  % Do Stuff
\fi:
```

Here, the ⟨*arg-spec*⟩ contains one letter, showing that only one argument is required. This argument is of

type N, meaning that it should be a single token *not* surrounded by braces.

3.2 Example kernel functions

Renaming primitives helps to keep the new syntax consistent, but does not show why the argument specifier is useful. This is perhaps best seen by looking at some of the functions provided by expl3.

By using the argument specifier, the new kernel provides families of related functions which avoid the need for complex \expandafter runs. For example, the TEX primitive \let can only be used with a macro name and a single token; no braces. In LATEX3, the family of \let-like macros contains:

```
\cs_set_eq:NN \Macro_One \Macro_Two
\cs_set_eq:Nc \Macro_One {Macro_Two}
\cs_set_eq:cN {Macro_One} \Macro_Two
\cs_set_eq:cc {Macro_One} {Macro_Two}
```

where an argument specified as c is to be given in braces and should expand to a csname. This is much clearer than the equivalent plain TEX constructions; taking \cs_set_eq:Nc as an example:

```
\expandafter\let\expandafter\Macro_One
  \csname Macro_Two\endcsname
```

The specifiers n (no expansion), o (expand once) and x (\edef-like full expansion) allow large families of related functions to be created easily, so that using the results is simplified. Thus we can create a macro \Macro_One:nn, then create \Macro_One:no, \Macro_One:xn and so on very rapidly. Later, we will see how the v and V argument specifiers add even more power to this concept.

The argument specifier concept also makes testing much easier. As an example, the new kernel provides three tests related to the \@ifundefined macro:

```
\cs_if_exist:cT  {csname} {true}
\cs_if_exist:cF  {csname} {false}
\cs_if_exist:cTF {csname} {true} {false}
```

In all three cases, the first argument will be converted to a csname (the c specifier). The first two functions then require one more argument, either T or F. As might be expected, these are executed if the test is true or false, respectively. The third function (ending :cTF) has both a true and false branch. By providing tests with the choice of T, F and TF arguments, empty groups in code can be avoided and meaning is much more obvious.

4 Data storage

In LATEX3, macros which carry out some process are called functions, and all contain an argument specifier. Macros used for storage are handled separately, to help to make code cleaner and easier to read. To further aid the programmer, expl3 defines several new data types:

- token lists (tl),
- comma lists (clist),
- property lists (prop),
- sequences (seq),

in addition to the existing types, which are renamed:

- boolean switches (bool),
- counters (int),
- skips (skip),

and so on.

The name "token list" may cause confusion, and so some background is useful. TEX works with tokens and lists of tokens, rather than characters. It provides two ways to store these token lists: within macros and as token registers (toks). LATEX3 retains the name "toks" for the later, and adopts the name "token lists" (tl) for macros used to store tokens. In most circumstances, the tl data type is more convenient for storing token lists.

The other new variable types are all essentially lists of items separated by a special token. The nature of the separator determines the type of variable and what functions apply. For example, a comma list is, as you might expect, a set of tokens separated by commas.

These are all created explicitly as either local or global, according to a prefix \l_ or \g_. For example, a local tl may be named:

```
\l_mymodule_myname_tl
```

while a global tl looks like this:

```
\g_mymodule_myname_tl
```

The other variable types follow the same pattern, with the appropriate type identified in the variable name.

As well as the new data types, expl3 provides a range of functions for manipulating data. Often, these had to be coded by hand when using LATEX 2_ε. For example, \tl_elt_count:N is available to count the number of elements (often characters) in a token list.

5 Expanding variables

When coding in (LA)TEX, the need to access data in variables is made more complicated by the different possibilities for recovering information later. For example, if three macros are defined as

```
\def\tempa{Some text}
\def\tempb{\tempa}
\def\tempc{\tempb}
```

then there are two likely scenarios for using the information in `\tempc`:

- Use of the value that `\tempc` contains (in this case `\tempb`);
- Exhaustive expansion of `\tempc` to use the unexpandable token list it represents (in this case "Some text").

The situation is further complicated as macros do not need an accessor function, whereas other TEX variables (toks, counts, skips) do. This leads to the need for carefully-constructed `\expandafter` runs in (LA)TEX, in order to get the content needed.

To avoid this, LATEX3 provides two argument specifiers which will always return the content of a variable. The `V` specifier requires the name of a variable, and returns the content. For example, if we define two variables, one of type `tl` and the other of type `toks`,

```
\toks_set:Nn \l_my_toks { Text \mymacro }
\tl_set:Nn  \l_my_tl  { Text \mymacro }
```

and pass them to some function `\foo_bar:V`,

```
\foo_bar:V \l_my_toks
\foo_bar:V \l_my_tl
```

both sets of input will result in "`Text \mymacro`" being passed as the argument to the "underlying" function (explained below) `\foo_bar:n`. The `V` specifier can be applied to any LATEX3 variable: this means that the programmer does not have to worry about how data is stored at a TEX level. A function using a `V` specifier will always receive the content of the variable passed.

The second "variable" specifier is `v`. This converts its argument to a csname, then recovers the content of the resulting variable and passes the content. Thus we might use a `\foo_bar:v` as:

```
\foo_bar:v { l_my_toks }
\foo_bar:v { l_my_tl }
```

with the same result as the previous example.

The two variable specifiers are very powerful. By using them, the programmer can almost entirely avoid the need to worry about the order of expansion when using stored information.

In LATEX3, functions which differ only in the argument specifier should carry out the same underlying operation: the only difference should be the processing of arguments *prior* to applying the function. Normally, the "underlying" function will act without argument expansion (taking `n` or `N` type arguments). Thus `\foo_bar:c` will normally be defined as expanding a csname and passing it to `\foo_bar:N`.

6 Other key features

The new kernel will require the ε-TEX extensions. Thus, those new primitives are always available when working with LATEX3. For example, `\unexpanded` is part of the expansion module, as `\exp_not:n`.

Boolean switches in TEX and LATEX 2ε use the `\iftrue` and `\iffalse` primitives. This can lead to problems nesting (`! Incomplete \if...`). To avoid this, LATEX3 does not create switches in the same way. This means that all of the switches use exclusively LATEX syntax, and require an "access" function.

```
\bool_if:NT  \l_example_bool { true code }
\bool_if:NF  \l_example_bool { false code }
\bool_if:NTF \l_example_bool { true code }
  { false code }
```

One of the most useful features of the new coding syntax is the treatment of white space. The literal space character ( ) is ignored inside code blocks, meaning that the text can be laid out to aid ease of reading. When a space is required in the output, a tilde (`~`) can be used. In this context, `~` is *not* a "hard" space, but a character with category code 10. The ability to finish lines without worrying about omitting or including `%` is highly welcome!

7 Conclusions

The current LATEX3 modules provide a new and powerful programming language for TEX. The full details of the language are collected in one place, and the language is much more logical than the current mix of TEX and LATEX 2ε. LATEX3 is therefore ready for serious use by (LA)TEX programmers.

At this stage, the document level of LATEX3 is much less defined. It seems likely that good separation of programming and document design will be made available. The new code syntax means that a number of ideas currently implemented as independent packages will need to be re-implemented either in the new kernel or as supported tools.

My own experience with LATEX3 convinces me that the kernel team need outsiders to use the code. The team has done a very good job so far, but everyone will bring new approaches to using the code. With the involvement of the wider TEX community, LATEX3 has the potential to be a major step forward for LATEX.

◇ Joseph Wright
2, Dowthorpe End
Earls Barton
Northampton NN6 0NH
United Kingdom
joseph dot wright (at)
morningstar2 dot co dot uk