Abstract

Modern LaTeX programming: an example based LaTeX3 tutorial

Ziyue Xiang

Contents

List of Examples

1 Introduction

There is no doubt that LaTeX is viewed as a typesetting language by most of the users. The programming aspect of LaTeX is often overlooked by many. In practice, many large and structured documents can benefit from the programming capabilities provided by LaTeX. Even understanding the most basic programming principles in LaTeX can greatly facilitate the efficiency of generating figures or tables that are made up of similar and repeated sub-structures. The infrastructure provided by LaTeX 2_ε [2] is already Turing complete, which means its programming capabilities are identical to those of Python [13] and C [10]. However, the syntax and conventions of LaTeX 2_ε are nonstandardized and obsolete compared to the mainstream programming languages now. This makes learning LaTeX 2_ε programming more difficult for today's LaTeX users.

In order to modernize the programming interfaces in LaTeX, the LaTeX3 programming interfaces are introduced [6]. Unfortunately, the learning materials for this tool chain is scarce. One of the few resources available for new learners is The LaTeX3 Interfaces [12], which is an API documentation that is difficult to for one to start with. Therefore, in this material we intend to provide an example based LaTeX3 tutorial for LaTeX3 learners with sufficient background in computer programming. That is, the reader is expected to understand basic structures (e.g., loops, conditional branches) as well as data types (e.g., integers, floating point numbers, strings) in programs. It would be helpful if the reader also understands the basic principles of the C programming language [10].

Since the LaTeX3 project has accumulated a huge code base over the years, it is infeasible to cover all of its functionalities in one tutorial. In this tutorial, we focus on the most frequently used components in LaTeX3. The complete API documentation can be found in [12]. In the upcoming section titles, if parentheses are present, then the section number in [12] corresponding to the section in this tutorial is shown in the parentheses.

1.1 Motivations of LaTeX3

As mentioned above, LaTeX 2_ε is already Turing complete and serves as the building blocks of many existing packages. In this section, we describe the problems of traditional LaTeX programming, which justifies the reason why LaTeX3 is developed despite already having the powerful LaTeX 2_ε and many other existing packages.

Nonuniform interface Outside LaTeX3, the interfaces provided by traditional LaTeX are not standardized. They suffer from the following disadvantages.

Firstly, the mechanism of LaTeX can affect the readability of traditional LaTeX code. In LaTeX, functions and variables are all declared as control sequences (see Chapter 3 of [5]). When a function is invoked, it is expected to execute a series of predefined procedures, whereas variables are used to store values only. In LaTeX, we can declare functions that absorbs one or multiple arguments. The homogeneous nature of functions and variables can make reading traditional LaTeX source code difficult. In the example below, we define 6 control sequences, where `\ta` and `\td` are functions, and the rest are variables.

```
1 \newcommand{\ta}[2]{[arg1={#1}, arg2={#2}]}
2 \newcommand{\tb}{$\alpha$}
3 \newcommand{\tc}{$\beta$}
4 \newcommand{\td}[1]{[arg3={#1}]}
5 \newcommand{\te}{$\gamma$}
6 \newcommand{\tf}{$\delta$}
7 \ta\tb\tc\td\te\tf
```

[arg1=α, arg2=β][arg3=γ]δ 1

On line 1:7[1], we call functions `\ta` and `\td` with their respective arguments (stored in variables), as well as output the value of `\tf`. In appearance, line 7 is six control sequences placed next to each other. It is very difficult to understand the code unless the programmer finds out which control sequences are functions and how many arguments each function

uses. This makes the source code of some LaTeX 2_ε packages challenging to read.

Secondly, in traditional LaTeX the implementation of many fundamental programming capabilities are provided by external packages. As a result, the functionalities of multiple packages may overlap. For example, to compare the equality of two strings, we can use `\ifthenelse` and `\equal` from `ifthen` package [3]; we can use `\pdfstrcmp` from `pdftexcmds` package [4]; we can also use `\IfStrEq` from `xstring` package [11]. The use of multiple similar packages is likely to cause redundancy and compatibility issues. The lack of a centralized documentation and comparison for these similar packages also increases the learning cost of traditional LaTeX programming.

Expansion control Unlike generic programming languages, LaTeX does not have support for types. Programming components such as variables, functions and function arguments are all treated in the same way as macros. As a result, LaTeX programmers usually need to more precisely control how variables are defined and how functions are called. These techniques are known as expansion control.

Expansion control is mostly required in two scenarios.

Fundamentally, TeX works by doing: commands are substituted by their definition, which is subsequently replaced by definition's definition, until something irreplaceable is reached (e.g. text or TeX primitives). This process is called *expansion*. The mechanism of expansion may sound simple and straightforward. However, it usually requires a lot of manual fine-tuning in practice.

Consider the example below. We know that the `\uppercase` macro capitalize English letters, which renders the first output line in all caps. But if we store some text in `\myname` and then apply `\uppercase` to the command, we can see that the output is *not* turned into uppercase letters.

```
1 \par\uppercase{Alan Xiang}
2 \newcommand*{\myname}{Alan Xiang}
3 \par\uppercase{\myname}
```

```
ALAN XIANG
Alan Xiang
                                                    2
```

Why would this happen? Let us dig into how `\uppercase` works. The `\uppercase` macro scans each token[2]inside its argument group one by one. If an English letter is encountered, its uppercase form is left in the output stream. If a command is encountered, it will not try to apply `\uppercase` to the content of the command. Instead, the command itself will be placed into the output stream. In this case, `\myname` will be left untouched in the output, which is subsequently expanded to its original definition.

(More details about tokens can be found in and Chapter 7 of [5])

What if we also want to capitalize the content of `\myname` as well? To achieve this, we need to fine-tune the expansion process by changing the *order* of expansion. That is, to expand `\myname` before `\uppercase`. In this way, the `\uppercase` command will receive the content of `\myname` in the form of English letters, which allows capitalization to function correctly.

In LaTeX, the classic way of controlling the order of expansion is via the `\expandafter` macro, which it is notoriously difficult to use. According to *A Tutorial on* `\expandafter` [1], to reverse the expansion of a series of n tokens, the ith token has to be preceded by $2^{n-i}-1$ `\expandafter`s. The exponential growth of the number of `\expandafter`s greatly reduces the readability of source code and increases the chances of mistakes. For example, in Joseph Wright's answer to an expansion-related question on TeX StackExchange [9], a total of 26 `\expandafter`s are used to reorder the expansion of merely 4 arguments. To avoid this annoyance, one of the key features of LaTeX3 is to provide simple and reliable expansion control.

Modernized experience TeX was first designed in the late 1970s, when computer hardware and programming languages were prototypes compared to their contemporary counterparts. As a result, TeX and LaTeX contain quirky usages that may seem odd for programmers today. For example, to multiply a counter variable by 3, one writes `\multiply\counter by 3`; to invoke the `date` command via the terminal, one writes `\immediate\write18{date}`. It can be seen that these syntaxes are either outdated or perplexing. In a fairly popular language nowadays (e.g. Python), these two tasks can be done by `counter*=3` and `os.system('date')`, whose code possesses superior simplicity and interpretability. LaTeX3 attempts to modernize the LaTeX language by adapting to modern-language-like syntaxes and introducing a naming system that makes LaTeX code more readable.

[1] Every listing has a unique index, which is shown at the bottom right corner. A line in the listing is referenced by <listing index>:<line number>.

[2] Tokens are smallest units that TeX compilers work with. For now, we can consider a token to be either a character or command. For more about TeX tokens, see [8].

1.2 Compiling Examples

This tutorial is based on examples. To compile the examples, the minimum preamble required is:

```
1 \documentclass{article}
2 \usepackage{tikz} % load TikZ for some TikZ examples
3 \usepackage{expl3} % load latex3 packages
```

The example code should be placed between `\begin{document}` and `\end{document}`. All examples are tested with TeXLive 2020 on Ubuntu 20.04. For newer versions of LaTeX compilers, there is no need to load the `expl3` package explicitly (i.e., line 3:3 is optional).

The source code of this tutorial can be obtained from https://github.com/xziyue/latex3-tutorial-latex-source.

2 LaTeX3 Naming Conventions (I-1)

In Python or C++, if we see `a(b);`, we can tell `a` is a function and `b` is its argument. However, in LaTeX, if we see `\a\b`, there are be two possibilities:

- `\a` is a function and `\b` is its argument
- Both `\a` and `\b` are variables

The syntactic design of LaTeX makes it difficult to distinguish between functions and variables, for each control sequence can either be a function that receives arguments or a variable that absorbs nothing. It can lead to confusion when one is trying to understand others' source code. Therefore, LaTeX3 introduces a set of naming rules that encode important information into the name of control sequences as a way to improve readability.

Before discussing LaTeX3 naming conventions, let us take a diversion to look at the low-level design of LaTeX and find out how we can use non-English characters in command names.

2.1 Category Code & Command Name

When the LaTeX compiler reads a source file, it will read and process each character one by one. For each character in the file, in addition to its character code, LaTeX compiler will also assign a *category code* based on current category code table. The default LaTeX category code table is shown in Table 1.

Category Code	Description	Character(s)
0	Escape character: tells LaTeX to start looking for a command	\
1	Start of group	{
2	End of group	}
3	Toggle math mode	$
4	Alignment tab	&
5	End of line	'\r'
6	Macro parameter	#
7	Superscript	^
8	Subscript	_
9	Ignored character	'\0'
10	Spacer	'\32', '\t'
11	Letter	A–Z, a–z, ...
12	Other	0–9, +, @...
13	Active character: used for single character commands	~...
14	Comment character: ignore everything that follows until end of line	%
15	Invalid character: not allowed in `.tex` files	'\127'...

Table 1: Default LaTeX category code table [7]. Characters surround by single quotes indicate their C-style representation.

LaTeX reacts to each character according to its category code instead of character code. If we change the category code associated with a character, we can completely change the *meaning* of that character. For example, if we assign category code 7 to _ and category code 8 to ^, we can use _ to denote superscript and ^ to denote subscript.

Example 2.1

```
1  \ExplSyntaxOn
2  \tl_set:Nn \l_tmpa_tl {A}
3  \group_begin:
4  \tl_set:Nn \l_tmpa_tl {B}
5  \par value~inside~group:~\tl_use:N \l_tmpa_tl
6  \group_end:
7  \par value~outside~group:~\tl_use:N \l_tmpa_tl
8
9  \tl_set:Nn \l_tmpb_tl {A}
10 \group_begin:
11 \tl_gset:Nn \l_tmpb_tl {B}
12 \par value~inside~group:~\tl_use:N \l_tmpb_tl
13 \group_end:
14 \par value~outside~group:~\tl_use:N \l_tmpb_tl
15 \ExplSyntaxOff
```

```
value inside group: B
value outside group: A
value inside group: B
value outside group: B
```

4

References

[1] S. V. Bechtolsheim. A tutorial on `\expandafter`. *TUGboat* 9(1):57–61, 1988.

[2] K. Berry, S. Gilmore, and T. Martinsen. *LaTeX 2ε: An Unofficial Reference Manual*. 12th Media Services, 2017.

[3] D. Carlisle, The LaTeX Project Team, and L. Lamport. The `ifthen` package. https://ctan.org/pkg/ifthen?lang=en

[4] D. Carlisle, The LaTeX Project Team, and L. Lamport. The `ifthen` package. https://ctan.org/pkg/ifthen?lang=en

[5] D. E. Knuth and D. Bibby. *The TeXbook*. Addison-Wesley Reading, 1984.

[6] F. Mittelbach and the LaTeX Project Team. Quo vadis LaTeX(3) team — a look back and at the upcoming years. *TUGboat* 41(2):201–207, 2020.

[7] Overleaf. Table of TeX category codes. https://www.overleaf.com/learn/latex/Table_of_TeX_category_codes

[8] Overleaf. What is a "TeX token"? https://www.overleaf.com/learn/latex/Articles/What_is_a_%22TeX_token%22%3F

[9] PLK. Expanding arguments before macro call, March 2013. https://tex.stackexchange.com/questions/104506/expanding-arguments-before-macro-call

[10] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk. *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[11] C. Tellechea. The `xstring` package. https://www.ctan.org/pkg/xstring

[12] The LaTeX Project Team. The LaTeX3 Interfaces, October 2020. http://ctan.math.washington.edu/tex-archive/macros/latex/contrib/l3kernel/interface3.pdf

[13] G. VanRossum and F. L. Drake. *The Python language reference*. Python Software Foundation Amsterdam, Netherlands, 2010.

◇ Ziyue Xiang
Purdue University
ziyue.alan.xiang (at) gmail (dot) com
https://www.alanshawn.com
ORCID 0000-0001-6054-5801