Which Success for TEX as an Old Program?

Jean-Michel Hufflen

Abstract

We propose a personal analysis of TEX's strong and weak points. In particular, we show that this program's history can explain some conventions viewed as strange nowadays. From a point of view related to teaching LATEX to Computer Science students, going thoroughly into these strong and weak points allows us to show how this field has evolved over decades.

Sommario

In questo articolo è proposta un'analisi dei punti di forza e di debolezza di TEX. In particolare, si mostra come la storia di questo programma possa spiegare alcune convenzioni, che al giorno d'oggi possono sembrare poco familiari. Dal punto di vista di chi insegna LATEX a studenti che seguono corsi di informatica, esaminare scrupolosamente tali punti di forza e di debolezza consente di mostrare l'evoluzione, nel corso dei decenni, di questo campo di studi.

1 Introduction

The story has begun in 1978. Donald Knuth provided a powerful system, as a *kernel* and a *format*, *Plain TEX*, able to typeset texts. The adaptation of the notion of *document type*—originating from Scribe (Reid, 1984)—by Leslie Lamport resulted in a new format, LATEX, built out of TEX. LATEX is the most well-known format nowadays, especially within scientific publication, but also used in other topics, including history, humanities and social sciences. As another illustration of this success, the curricula of many universities and high schools include an introduction to LATEX. A good example of support material for such an introduction is given as part of last GuIT[1] meeting (2019). Of course, such an introduction is designed for PhD students, but sometimes also for graduate or undergraduate ones. This way, they can learn how to manage large-sized documents. Beyond LATEX's basic commands and environments, they can get aware of the separation of *data* and presentation, and more generally, the separation of *form* and *substance*.

TEX's kernel is a very old—and old-fashioned—program: as mentioned above, its first version came out in 1978. Current coding practices in programming have very little to do with the look of the languages of that time. Nevertheless, the typesetting systems built out of TEX are still widespread. Moreover, TEX's end has been announced many times... but the word processors based on TEX—LATEX, ConTEXt (Hagen, 2001), more recent—remain unrivalled for getting high-quality output prints. Only a few programs got a lifetime comparable to TEX's. But, due to some design choices, this program shows its age, even it is still at the forefront.

Hereafter we propose a personal analysis of strong and weak points of LATEX. We aim to emphasise that most of the weak points that perpetuate in recent versions result from design choices performed since the first versions. Teaching TEX & Co. to new end-users may be complicated because of these weak points, because they may be difficult to justify. But we think that Computer Science students—especially undergraduate ones—can get some experience from going thoroughly into this program. They can see how some initial design choices may become recurrent from a version to another. They can also discover how programming and software engineering have evolved since TEX's first version. In Section 2, we briefly recall the advantages of these typesetting systems. Section 3 is devoted to the some 'historical' conventions that may seem to be strange to new users. After discussing our way to introduce LATEX in Section 4, we conclude about our approach. Reading this farticle only requires basic knowledge about (LA)TEX, more precise details can be found in (Knuth, 1986a,b; Mittelbach *et al.*, 2004). Some technical points more related to Computer Science technique are explained in appendices: we tried to be precise for readers unfamiliar with Computer Science, without being too technical.

2 LATEX's advantages

As mentioned above, the typesetting systems built out of TEX are unrivalled for getting high-quality output prints. They are not interactive, they work like a programming language's compiler. This point may be viewed as a drawback in comparison with interactive WYSIWYG[2] systems. In fact, people writing very short reports or formatting documents that will be not reworked later may prefer interactive and graphical menus of systems such as Microsoft Word or InDesign. On the contrary, LATEX is often preferred by users writing large-sized doc-

1. ***G**ruppo **U**tilizzatori **I**taliani di **T**EX.*

2. '**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et', in contrast to WYSIWYM (**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **M**ean) systems.

uments, or articles existing in different versions: a simple example can be given by an article included into a conference's proceedings, and later republished in a journal's issue, the layouts of these two versions may differ. For example, the former may be based on one-column layout, the latter on two-column layout.

TEX's kernel includes a kind of programming language, allowing users to develop new functions or customise existing ones. Roughly speaking, this language, mainly based on *macros*[3], is old-fashioned, and putting ambitious applications using it may be tedious, but many forums of LATEX users, organised by local user groups—e.g., GUIT—can help new programmers fix many mistakes. Anyway, such synergy among end-users is an important feature of *free software*. The same holds true for ConTEXt. LATEX's current version—LATEX 2ε—came out in 1994 (Lamport, 1994). As a suitable illustration of such synergy, this version provides many *packages* (Mittelbach *et al.*, 2004), developed by many users and whose fields are very diverse: some can be used for literary purposes, some are devoted to applications in Physics and Chemistry, etc. Many graphical tools usable around LATEX have been developed, too, as described in (Goossens *et al.*, 2009).

L. Lamport (1994) introduces LATEX 2ε as a higher-level system, in comparison with *Plain TEX*. This effort to move towards a very high-level programming language to express tasks LATEX can perform is in progress with the LATEX 3 project (Mittelbach and Schöpf, 1991). The result should be a language whose syntax is very different than LATEX 2ε's, this new language using precise notations for *types*, *variables*, *functions* and *evaluation strategies*. As shown in (LATEX3 project, 2020), the expl3 package allows end-users to experiment this new syntax. A recent survey is given in (Mittelbach and the LATEX3 project team, 2020).

Although TEX's end, as a legacy program's, was announced many times, new versions progressively incorporated modern requirements such as internationalisation or new schemes for font management. Moreover, the expressive power of engines based on TEX's kernel has been greatly increased by coupling it with a modern programming language. The most known example is LuaTEX (Hagen, 2006), where the engine can call procedures written using the Lua language (Ierusalimschy, 2006), other experiment connect TEX with Python (Lutz, 1996). Interesting applications based on such a *modus operandi* can be found in (Menke, 2019; Ziegenhagen, 2019). Developing such channels of communication between TEX and modern languages have succeeded, but let us mention that putting such extensions into action by using TEX's implementation language would have been perilous: on the one hand, this language is no longer practised, on the another hand, no one but D. Knuth is sufficiently familiar with TEX's sources in order to change them.

3. See Appendix A for more details about this notion.

3 LATEX's strange behaviour

If you practise LATEX for many years, you probably got used to these points for a long time. But if you have now to design an introduction to new end-users, some conventions will immediately appear as strange. Syntactic rules for mark-up are more homogeneous in LATEX than in *Plain TEX*[4]: a good example is given by the use of square brackets for commands' optional arguments. The recommended use of:

```
\begin{name} ... \end{name}
```

—where ***name*** is a command name dealing with sizes, such as `small` or `large`—rather than:

```
{ \name ... }
```

makes easier the use of such commands[5]. But let us remark that braces are often used to delimit arguments, and sometimes to limit local behaviour, what may be ambiguous. On the contrary, the duplicate commands for font style changes—e.g.:

```
\textbf{...} vs {\bfseries ...}
```

is related to the difference between *short* and *long* commands (Mittelbach *et al.*, 2004, Table 7.2): the latter can include an end-of-paragraph mark—that is the `\par` command—not the former. But this difference is less relevant than thirty years ago: at that time, running LATEX on a document that was about one hundred pages long took a while[6]. When LATEX processed such an end-of-paragraph mark, checking that there was no short command unclosed before continuing seemed more efficient than taking a while in order to reach the document's end, performing such check, and discovering that many commands were unclosed. But due to progress about computers' efficiency, this point is less justified nowadays.

TEX's language allows Computer Science students to deal with a *dynamic* language, whereas the programming languages used nowadays are *lexical*, in general (cf. Appendix B). When modern programming languages are used, compiling source files is based on both *lexical* and *syntactic* analyses. On the contrary, TEX's analyser of input streams has only one analyser, performing the whole of the process. As an illustration of old-fashioned

4. ... but such rules are more systematic in ConTEXt than in LATEX.

5. The same remark applies to LATEX's `sloppypar` environment, in comparison with the `\sloppy` command (Mittelbach *et al.*, 2004, p. 103).

6. We personally remembered that time, as reported in (Hufflen, 2003) in French.

syntactic conventions, a command's name must be followed by a separator, and a space character used as such is not put in the ouput built by LaTeX:

`\LaTeX is beautiful` (1)

(it is well-known that a possible workaround is to replace '`\LaTeX `' by '`\LaTeX\ `'). Likewise, some strange behaviour result from the dynamic processing of conditional expressions: more examples are given in Appendix C.

In other situations, LaTeX aims to be ready for the next run. When we teach cross-references—by means of the commands `\label` and `\ref`—to students, they ask if it would possible to iterate LaTeX's running until solving cross-references reaches a stable state[7]. Here also, the answer is given by what happened many years ago: when a text was not finished, end-users may run LaTeX just in order to ensure that there was no syntactic mistake, but these end-users were not interested in checking an unfinished text's layout, they only aimed to know if they could go on later with their text. Let us recall that at that time, running LaTeX sometimes took a while…

According to a close point of view, LaTeX sometimes aims to be ready for the next run, provided that end-users help it. When LaTeX warns an end-user about an *overfull hbox*, often it allows this user to see where a word should be hyphenated. Even if a good technique is to fix that only on a text's final version, such convention yields good result, provided that end-users understand that LaTeX does not always build the best and final version, but allows users to approach it at next run. This point also holds on about marginal notes misplaced at a page's beginning. About vertical alignment of successive paragraphs on the same page, it has been said that TeX was either perfect, or deficient. In fact, if you aim to reach higher-level quality, these problems—orphans and widows—could be solved by using some commands such as `\looseness` (Knuth, 1986a, pp. 103–104). More recently, some advanced features of the microtype package (Schlicht, 2010)—*protrusion* and *font expansion*—based on the PDF[8] format—allow final texts to be reworked very precisely. This package also allows *tracking*[9]—that is, working on space between letters—by means of the commands `\textls` and `\SetTracking`, but this *modus operandi* should be very marginal. Anyway it should be reserved for very difficult cases, because it is not recommended within good typography, it should be applied to fragments using only capitals or only small capitals.

7. Let us remark that such iteration aiming to solve cross-references is done by ConTeXt's `texexec` command.

8. **P**ortable **D**ocument **F**ormat, Adobe's format.

9. This feature is also provided by the companion package letterspace, and by the soul package (Mittelbach *et al.*, 2004, § 3.1.7).

4 Discussion

As mentioned above, TeX's syntax is complicated, in comparison with LaTeX's. For example, LaTeX recommends an `\input` command's argument to be surrounded by braces—as any command's argument—whereas braces are not needed in *Plain TeX*. As any end-user may see, `$\frac12$` yields '$\frac{1}{2}$', that is '`1`' and '`2`' are obviously distinct tokens. That is not true for '`16`' in the following statement:

```
\write16{Do you enjoy \LaTeX?}
```

displaying the second argument at a terminal (Knuth, 1986a, pp. 226–228). In addition, braces are not allowed to surround an output stream number, the first argument of the `\write` command. LaTeX tries to deal with more systematic notations, but in general, what is usable in *Plain TeX* works within LaTeX. Some notations originating from *Plain TeX* are still used, because of bad habits. For example, '`$$...$$`' for a centered mathematical formula, although this convention is deprecated in LaTeX and should be replaced by '`\[...\]`' (Downes, 2017, § 2.1). *Plain TeX*'s conditional statements (cf. Appendix C) are old-fashioned and in LaTeX, we should use only the `\newif` command for simple cases (Knuth, 1986a, p. 211)) and the ifthen package (Mittelbach *et al.*, 2004, § A.3.2) or even better the etoolbox package (Lehman and Wright, 2020) for the others. But many source texts of packages use these old conditional statements. User-defined commands should be able to end with a space character if need be when the xspace pakage (Mittelbach *et al.*, 2004, § 3.1.2) is used. So, the behaviour pointed at (1) can be avoided. But this package sometimes makes a wrong decision, in which case some settings belonging to this package can be needed (Carlisle and Høgholm, 2014), putting '`{}`' at the end of such a command being another immediate solution.

We agree that some technical points may be avoided for non-Computer Science students. As an Assistant Professor of Computer Science, we think that Computer Science is… a *science*, and we have to teach this point to our students, who aim to be specialised in this topic. This science encompasses *techniques* and *methods*, but also has a *History*. Some programs got a very long lifetime, because they have been able to incorporate some new requirements, and a program such as TeX can illustrate that. Some programs are *monolithic* and it seems too difficult to rework the kernel: TeX belongs to this kind of programs, too. Some techniques have been experienced in Computer Science and have led to some behaviour difficult to understand and reproduce, or to some inconsistency: the problems raised by the way TeX parses its programs can show that. Some students have never read any program using '`goto`' statements: some examples chosen within (Knuth, 1986b) can illustrate this technique... and its drawbacks. This choice is relevant since TEX's implementation language, `WEB`[10], is quite structured, that is, the use of '`goto`' statements is marginal, but instructive from a teaching point of view. Last, but not least, many students program as if they take no care of efficiency[11]. Here also, we can speak about NTS[12] (Taylor *et al.*, 2000). This rewriting of TEX in Java, developed at the 20th century's end, focused mainly on an object-oriented approach and neglected efficiency questions. As a result, this program worked much slower than the original TEX program and was unable to replace it.

To sum up, we think that the students who aim to become computer scientists can take advantage to the study of some obscure points of TEX. This program is still used: they can learn how it has been designed and how it works. This is an occasion of dealing with old techniques, not artificially. In fact, most of them enjoy that[13].

5 Conclusion

People who have to process short texts, or texts that will be not reworked, may be not interested in learning LATEX. The advantages of its approach are relevant for big-sized documents, possibly written by several authors. On another point, the curriculum of our students in Computer Science includes some *projects*, by groups. Every project ends with a report and an oral defence. We have noticed that many groups use LATEX for these two tasks, although they do not have to. Anyway, we can deduce that they enjoy using this typesetting program. The final touch of our lecture consists of a short introduction to Overleaf (2020), which is a collaborative cloud-based editor of LATEX documents and templates. Even if the LATEX version used may be not updated, even if no program can impose consistency about the style of several authors, students dealt with these aspects, and they noticed that LATEX was able to be adapted to such modern use. We began to experience this way last year, just before the Covid health crisis (!). Of course, we do not know how long it will last, but since teleworking is now encouraged, we think that our students follow the right path. To sum up, we are a happy teacher of LATEX, especially to Computer Science students. We hope that these students are happy, too, even if they go behind the scenes.

10. A good introduction to this language is (Knuth, 1992).

11. For example, ask them for a program checking if a string s is a palindrom. Many will answer $s = \mathit{reverse}(s)$, where *reverse* returns the reversed form of a string. Of course, such a solution would be acceptable for a specification, but not for a real program, that would perform more comparisons than needed.

12. **N**ew **T**ypesetting **S**ystem.

13. But we recognise that these obscure points of TEX have never been used within any examination!

A Functions vs macros

Most often, *functions* belonging to modern programming languages use *calls by value*, that is, an argument is evaluated before applying this function. Let us consider the following *function*, written using Python, returning a list, its two elements being equal to the function's argument:

```
def twicef(x):
  return [x,x]
```

If you try to evaluate `twicef(2019 + 1)`, first the argument is evaluated to `2020`, and then this value `2020` is bound to the local argument `x`. Of course, the result is the list `[2020,2020]`. Different techniques may be applied in other languages, including more modern techniques that optimise the evaluation of an argument—interested readers can find a survey in (Aho *et al.*, 2006)—but in general, an argument of such a subprogram is not evaluated several times[14]. Now let us consider this definition of a *macro* of the Scheme functional programming language[15]:

```
(define-syntax twice-m
  (syntax-rules ()
    ((twice-m x) ; Such a call is expanded to:
     (list (quote x) (quote x)))))
```

Let us recall that Scheme systematically uses prefixed operators, as shown by this evaluation:

```
(+ 2019 1) ⟹ 2020
```

but when a *macro* is called, every argument is processed *as it is*. Let us notice that the `quote` special form[16]—used within the expanded form of the `twice-m` macro—inhibits the evaluation of its argument, so:

```
(twice-m (+ 2019 1)) ⟹
        ((+ 2019 1) (+ 2019 1))
```

TEX's commands are closer to macros than functions[17]:

```
\def\twicem#1{[#1,#1]}
```

14. ... except for some early programming languages... That was a long time ago...

15. A good introduction to this functional programming language, including its modern *hygienic* macro system, is (Dybvig, 1996).

16. Of course, `quote` is not a function.

17. By the way, let us mention that LATEX 3's terminology uses the *function* word (LATEX3 project, 2020), but according to a different sense in comparison with functional programming languages, even if programming *functions* according to this 'classical' sense is possible with expl3 (Gregorio, 2020). In fact, commands are divided into *variables* and *functions*, precise conventions rule available strategies for parameter passing.

That command builds its argument twice since it is put twice within the result. A simple test is given by an expression incrementing a counter and returning it[18]:

```
\newcounter{c}
\twicem{\stepcounter{c}\thec}
```

which yields '`[1,2]`'. If you would like this argument to be build only once, just put a *box* (Knuth, 1986a, pp. 120–122) as a container:

```
\newbox\tmpbox
\def\twicemversiontwo#1{%
 \setbox\tmpbox\hbox{#1}%
 [\unhcopy\tmpbox,\unhbox\tmpbox]}
```

and the expression:

```
 \twicemversiontwo{\stepcounter{c}\thec}
```

now yields '`[3,3]`'. Simpler cases can be solved by using local definitions or the `\expandafter` macro (Knuth, 1986a, p. 213).

B Lexically or dynamically

The difference between *lexical* and *dynamic* programming languages is related to *variables' scope.* Let us consider a subprogram that uses a notation not included into its arguments. When this subprogram is applied, a lexical (resp. dynamic) scope is put into action if we consider the value associated with this notation at definition-time (resp. runtime). Most programming languages used nowadays are lexical. About TEX, let us consider the following example, already given in (Hufflen, 2009):

```
\def\state{happy}
\edef\firstquestion{%
 You're \state, ain't U?\par}
\def\secondquestion{%
 You're \state, ain't U?\par}
\def\state{afraid}
```

In *Plain TEX*, many commands are introduced using the `\def` command (Knuth, 1986a, Ch. 20): such commands are dynamic, that is, other commands used inside bodies are expanded at runtime, so processing `\secondquestion` causes the paragraph:

You're afraid, ain't U?

to be typeset. By default, TEX is dynamic, but some lexical behaviour can be expressed using TEX's `\edef` command, because the whole body of such a command is fully expanded as soon as this command is defined. So, processing the `\firstquestion` command causes the paragraph:

You're happy, ain't U?

18. The `c` counter, defined hereafter, will be reused later.

```
def max3(x,y,z):
  if x >= y :
     if z >= x : return z
     else : return x
  elif z >= y : return z
  else : return y
```

Figure 1: Python program using conditional statements.

to be processed, even if the `\state` command has been redefined. LATEX's kernel provides the `\protected@edef` command (Mittelbach *et al.*, 2004, p. 892), similar to `\edef`. However, LATEX can be viewed as dynamic, since the preferred ways to define new commands—the `\newcommand` command and similar constructs mentioned in (Mittelbach *et al.*, 2004, § A.1.2)—introduce dynamic commands.

We have to pay particular attention when `\edef` is used for commands with arguments, since such a definition's body is expanded as far as possible *at definition-time*—some pitfalls are described in (Knuth, 1986a, pp. 215–216)—in particular, it is unsuitable to avoid an argument's multiple evaluation in the commands given in § A:

```
\def\twicemversionthree#1{%
 \edef\tmp{\noexpand#1}%
 [\tmp,\tmp]}
```

If the `\noexpand` command (Knuth, 1986a, p. 213) is removed within the previous definition, it crashes because the argument is unavailable at definition-time. Processing:

```
\twicemversionthree{\stepcounter{c}\thec}
```

yields '`[3,3]`', that is, the `c` counter has not been incremented, the argument is not evaluated.

C Processing conditional expressions

As an example of using conditional statements, Fig. 1 gives a Python function returning the greatest value among three numbers. In Python and most of modern programming languages, some constructs are specified by means of *reserved words*, e.g., '`if`', '`else`', '`elif`' (for 'else if'). Such a reserved word cannot be used as a variable and conventions based on *delimiters* allow us to use them as *literals*: '`'if'`', '`"else"`', ...

When such a source text is processed by an *interpreter* or *compiler*, two analyses—*lexical* and *syntactic*—are performed. The former aims to divide a source text into a sequence of *tokens*: e.g., '`if`', `x`', '`>=`', '`y`', ... The latter checks if the successive tokens are assembled in conformity with the programming language's grammar. The result of these two analyses may be viewed as a *tree*, Fig. 2

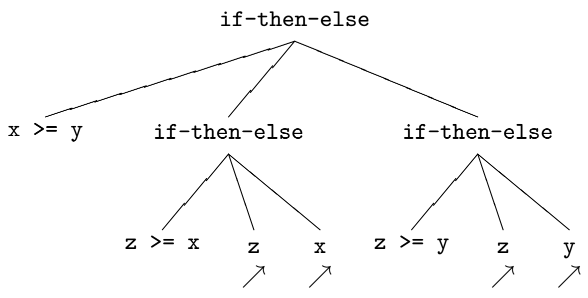

FIGURE 2: Abstract syntax tree for Fig. 1's program.

giving a sketch[19] of a tree modelling Fig. 1's program. So the *complete* structure of a source text is made explicit before interpreting this structure, or generating an executable program from it.

The *modus operandi* is very different within TEX. Only one analyser encompasses both lexical and syntactic analyses. More precisely, a command extracts its arguments as soon as it is recognised. Then TEX runs the command's body, and the process goes on. As a consequence, when TEX begins to process an '`\if...`' command, it does not know where this command will end. That is why we personally think that such process is fully *dynamic*, it may behave strangely if you get used to modern languages. Let us recall that all these '`\if...`' commands are documented in (KNUTH, 1986a, pp. 209–210): they end with a '`\fi`' marker. There may be an '`\else`' optional part, but we do not use this feature in the examples given below. They aim to emphasise the difference between recognising the beginning of an '`\if...`' command and popping a token, even if it could begin an '`\if...`' command.

(i) `\iffalse no\iftrue yes\fi more\fi`

In such a case, the `\iffalse` command causes the following tokens to be skipped until the `\fi` marker: as explained in (KNUTH, 1986b, §§ 500ff), skipping `\iftrue` adds a level and the two occurrences of `\fi` are correctly associated with `\iftrue` and `\iffalse`, respectively;

(ii) `\if\iftrue a\fi a ok\fi`

because of the `\if` command, the first token is expanded, the second, too, and 'ok' is typeset;

(iii) `\ifx\iftrue a ko\fi`

here, the `\ifx` command takes the first two tokens without expanding them, so the comparison just involves `\iftrue` and `a`, the `\fi` marker is associated to the `\ifx` command, which is quite counter-intuitive at first glance;

(iv) `\iftrue\verb+\iffalse+something\fi`

because of the `\iftrue` command, the following tokens will be processed until the `\fi` marker: `\iffalse` is processed as the argument of the `\verb` command and the `\fi` marker is associated with `\iftrue`;

(v) `\iffalse\verb+\iffalse+no\fi no\fi`

like in (i), the tokens following the first occurrence of `\iffalse` are skipped, but the second occurrence of `\iffalse` adds a level when it is skipped and the `\fi` marker's two occurrences are needed in order for this input line to be processed without crash, even if this behaviour is counter-intuitive[20].

Finally, let us remark that an '`\if...`' command marker can appear inside a body's command without being associated with a `\fi` marker and *vice-versa*[21]. A very good example is the '`\loop ... \repeat`' macro, provided by *Plain TEX* for expressing iterative operations (KNUTH, 1986a, pp. 217–218 & 352), where `\repeat` is defined this way:

```
\let\repeat=\fi
```

Acknowledgements

This work has been supported by the EIPHI Graduate school. Many thanks to Francesco Biccari for his patience and Massimiliano Dominici, who has translated the abstract in Italian. I am also very grateful to the reviewers, who addressed me constructive criticisms and suggested me some improvement points.

References

AHO, Alfred V., Monica S. LAM, Ravi SETHI and Jeffrey D.ULLMAN (2006). *Compilers, Principles, Techniques and Tools.* Pearson Internationel Edition, 2ª edizione.

CARLISLE, David and Moten HØGHOLM (2014). *The xpace package.* http://ctan.org/pkg/xspace.

DOWNES, Michael J. with Barbara BEETON (2017). *Short Math Guide for LATEX.* http://tug.ctan.org/info/short-math-guide.pdf.

DYBVIG, R. Kent (1996). *The Scheme Programming Language. ANSI Scheme.* Prentice-Hall, 2ª edizione.

19. An 'actual' abstract syntax tree would be more expanded: an expression such as '`x >= y`' would be replaced by a sub-tree, the same for '...↗', picturing a value to be returned.

20. Roughly speaking, the `\verb` command belongs to LATEX, not to TEX's kernel, so it may be told that mixing LATEX and *Plain TEX* is not recommended and may cause strange behaviour. This remark does not apply to Example (iii), which is counter-intuitive, too.

21. That is impossible within modern languages. In Python (cf. Fig. 1), an `if` construct ends with a line beginning at the left of the indentation point.

GOOSSENS, Michel, Frank MITTELBACH, Sebastian RAHTZ, Denis B. ROEGEL and Herbert VOSS (2009). *The LATEX Graphics Companion.* Addison-Wesley Publishing Company, Reading, Massachusetts, 2ª edizione.

GREGORIO, Enrico (2020). «Funzioni e expl3». *ArsTEXnica*, **30**, pp. 36–45.

GuIT (a cura di) (2019). *Meeting, Sessione mattutina.* ArsTEXnica 28, pp. 8–134, Turin.

HAGEN, Hans (2001). *ConTEXt, the Manual.* http://www.pragma-ade.com/general/manuals/cont-enp.pdf.

— (2006). «LuaTEX: Howling to the moon». *Biuletyn Polskiej Grupy Użytkowników Systemu TEX*, **23**, pp. 63–68.

HUFFLEN, Jean-Michel (2003). «Mes diverses périodes avec LATEX». *Cahiers GUTenberg*, **42**, pp. 38–60.

— (2009). «Using TEX's language within a course about functional programming». *MAPS*, **39**, pp. 92–98. In EuroTEX 2009 conference.

IERUSALIMSCHY, Roberto (2006). *Programming in Lua.* Lua.org, 2ª edizione.

KNUTH, Donald Ervin (1986a). *Computers & Typesetting. Vol. A: The TEXbook.* Addison-Wesley Publishing Company, Reading, Massachusetts.

— (1986b). *Computers & Typesetting. Vol. B: The Program.* Addison-Wesley Publishing Company, Reading, Massachusetts.

— (1992). *How to read a WEB*, Center for the Study of Language and Information, capitolo 7, pp. 179–184. Numero 27 in Lecture Notes.

LAMPORT, Leslie (1994). *LATEX: A Document Preparation System. User's Guide and Reference Manual.* Addison-Wesley Publishing Company, Reading, Massachusetts.

LATEX3 PROJECT (2020). *The LATEX3 Interfaces.* http://ctn.org/pkg/l3kernel/interface3.pdf.

LEHMAN, Philipp and Joseph WRIGHT (2020). *The etoolbox Package. An e-TEX Toolbox for Class and Package Authors.* ctan.org/pkg/etoolbox.

LUTZ, Mark (1996). *Programming Python.* O'Reilly & Associates.

MENKE, Henri (2019). «Parsing complex data formats in LuaTEX with LPEG». *TUGBoat*, **40** (2), pp. 129–135. In Proc. TUG.

MITTELBACH, Frank and Rainer SCHÖPF (1991). «Towards LATEX 3.0». *TUGBoat*, **12** (1), pp. 74–79.

MITTELBACH, Frank and THE LATEX3 PROJECT TEAM (2020). «Quo vadis LATEX(3) team—A look back and at the coming years». *TUGBoat*, **41** (2), pp. 201–207.

MITTELBACH, Frank and Michel GOOSSENS, with Johannes BRAAMS, David CARLISLE, Chris A. ROWLEY, Christine DETIG and Joachim SCHROD (2004). *The LATEX Companion.* Addison-Wesley Publishing Company, Reading, Massachusetts, 2ª edizione.

OVERLEAF (2020). *LATEX, Evolved. The Easy to Use, Online, Collaborative Editor.* https://www.overleaf.com.

REID, Brian Keith (1984). «SCRIBE document production system user manual». Technical report, Unilogic, Ltd.

SCHLICHT, Robert (2010). *Microtype: an Interface to the Micro-Typographic Extensions of pdfTEX.* http://ctan.org/pkg/microtype.

TAYLOR, Philip, Jiří ZLATUŠKA and Karel SKOUPÝ (2000). «The NTS project: from conception to implementation». *Cahiers GUTenberg*, **35–36**, pp. 53–77.

ZIEGENHAGEN, Uwe (2019). «Combining LATEX with Python». *TUGBoat*, **40** (2), pp. 126–128. In Proc. TUG 2019.

▷ Jean-Michel Hufflen
FEMTO-ST (UMR CNRS 6174) & University of Bourgogne Franche-Comté,
16, route de Gray,
25030 Besançon CEDEX
France
`jmhuffle at femto-st dot fr`