When Typography Meets Programming*

Jean-Michel HUFFLEN
LIFC (EA CNRS 4157)
University of Franche-Comté
16, route de Gray
25030 BESANÇON CEDEX
FRANCE
jmhufflen@lifc.univ-fcomte.fr
http://lifc.univ-fcomte.fr/home/~jmhufflen

Abstract

TEX's language is a wonderful tool to customise (LA)TEX's behaviour. However, some applications related to 'pure' programming are difficult to put into action, as shown by LuaTEX, where some tasks can be delegated to Lua procedures. We show some examples directly related to typesetting, and demonstrate some solutions using 'more classical' programming languages.
Keywords (LA)TEX, Scheme, programming workarounds.

Streszczenie

Makrojęzyk TEXa jest cudownym narzędziem do modyfikowania zachowania (LA)TEXa. Jednak niektóre zastosowania związane z „czystym" programowaniem są w nim trudne do zrealizowania, co demonstruje LuaTEX, w którym niektóre zadania mogą być delegowane do procedur Lua. Pokażemy kilka przykładów bezpośrednio związanych z typografią i zaprezentujemy ich rozwiązania przy użyciu „bardziej klasycznych" języków programowania.
Słowa kluczowe (LA)TEX, Scheme, rozwiązanie zastępcze.

0 Introduction

TEX's language is known as a wonderful tool to typeset texts nicely. However there are some examples related to typography, where using a 'more classical' programming language can be judicious. The purpose of this talk is to show some examples of that. Hereafter, a particular case is given *in extenso*, other examples will be demonstrated at the BachoTEX conference. Reading this article only requires basic knowledge of TEX, the commands we mention are documented in [14].

1 Dealing with 'false' small capitals

Let us assume that you want to typeset 'Bachotek' using small capitals, that is, 'BACHOTEK'. Unluckily you are using an 'exotic' font that does not provide small capitals directly[1],so a workaround is to use smaller-sized capital letters. That is — even if the result may seem to be ugly — you can replace '`\textsc{Bachotek}`' by '`B\fake{ACHOTEK}`', using the `\fake` command defined in Fig. 1. Let us notice that putting:

```
\fake{BACHOTEK}
```

— or '`\fake{\uppercase{Bachotek}}`' — is incorrect, because the initial 'B' must be an 'actual' capital letter, in normal size. In addition, a command such as `\TeX` would remain untouched in:

```
\textsc{Bacho\TeX}
```

('BACHOTEX'), so our workaround should apply to this last example as '`B\fake{ACHO}\TeX`'.

Some solutions to this problem have already been proposed — e.g., in [13, § 103] — but as far as we know, all these solutions are not robust: they yield incorrect results if they are applied to texts containing uppercase letters or nested commands, that is, cases such as:

```
\textsc{\textbf{Bacho\TeX}}
```
[2]

If we would like to define an alternative command for `\textsc`, the solution is to examine *each* letter of

* Title in Polish: *Gdy typografia spotyka się z programowaniem*.

[1] ... or your publisher forces you to use such a font.

[2] In this last case, a font substitution may also be ordered by means of the `\DeclareFontShape` command [16]. We do not go thoroughly into this way.

```
\def\fake#1{{\small#1}}

\def\alttextsc#1{%
 \def\iter{%
  \afterassignment\testcase\let\next= %
 }%
 \def\testcase{%
  \expandafter\ifx\next\enditer\else%
   \ifcat\noexpand\next\relax\noexpand\next%
   \else%
    \if\next a\fake{A}\else%
     \if\next b\fake{B}\else%
      ...
     \fi%
    \fi%
   \fi\let\next=\relax%
  \fi\next\iter}%
 \def\enditer##1{\relax}%
 \iter#1\enditer%
}
```

Figure 1: Using 'false' small capitals.

the argument of this command, so the result would look like:

```
\textbf{%
 B\fake{A}\fake{C}\fake{H}\fake{O}\TeX%
}
```

for our last example. It is not difficult for a TEXpert to write a loop iterating on a string, but quite tedious. We need a marker for the end of this string, and a macro using the `\afterassignment` command puts this *modus operandi* into action[3], as shown in Fig. 1 by our `\alttextsc` command.

However, we can observe the huge number of '`\if...\fi`' constructs. Unfortunately, TEX does not provide any construct to check if a letter is upper-case (or down-case). First, the tables used by the commands `\uppercase` and `\downcase` are not directly readable; second, changing these tables by means of the commands `\lccode` or `\uccode` may affect other commands' behaviour. The second remark holds on for the table controlling character category (changeable by means of the `\catcode` command). Infact,inthislastcase,TEX provides the construct '`\ifcat...\fi`' in order to check a character's category... but uppercase and downcase letters default to the same category. In addition, let us notice that even if the best solution induced changes in a table (by `\catcode`, `\lccode`, or `\uccode`) for 26 letters — uppercase or downcase letters — such a way would result in a lengthy source text for our new `\alttextsc` command.

If we decide to use a 'more classical' programming language in order to implement the replacement of the string '`\textsc{Bachotek}`' by a quite equivalent string using our `\fake` command — that is, doing 'macro'-replacement before applying TEX's macros — the advantage is that most modern languages provide efficient constructs to iterate a function along a string. More precisely, they allow a function to be iterated on each character of a string, and then all these subresults can be collected into a structure, most often into another string[4]. Unfortunately, processing each character separately in the case of the `\textsc` command's argument fails for '`\textsc{Bacho\TeX}`'. Another solution using a 'mini' TEX parser is given in Fig. 2, using the Scheme programming language[5] [11]. The comments we put in this figure should make clear the broad outlines of our method. In addition to some predefined functions and special forms of Scheme, our `alttextsc` function uses some additional definitions provided by SRFIs[6]: the `let*-values` macro is defined in [3], the `cons*` function in [17], the `string-concatenate-reverse/shared` function in [18]. This `alttextsc` function also uses definitions belonging to MlBibTEX[7] source files [4]:

- `(mk-r-string-thunk s)` may be viewed as a character-generator: this expression returns a *thunk*[8]; successive calls of this thunk return the successive characters of the *`s`* string, in turn; when the end of *`s`* is reached, all the subsequent calls return the false value '`#f`';
- `(t-next-token thunk char x)` where *`thunk`* is a character-generator, *`char`* the last character read within the string or the false value '`#f`' if the end has been reached, and *`x`* an escape function to be called in case of an error, returns two values: the next token parsed according to TEX's conventions, and the character after; the resulted token may be a single character, or a command's name given as a string if this token

[3] We gave a similar example in [5]: a macro computing the numbers of characters of a string.

[4] Good examples are Scheme functions `string-map` and `string-for-each` [18]. So are the *iterators* of the STL (**S**tandard **T**emplate **L**ibrary), part of the C++ language [10].

[5] Readers interested in an introductory book to this functional programming language can refer to [19].

[6] Scheme **R**equests **F**or **I**mplementation. That is an effort to coordinate libraries and other additions to the Scheme language between implementations.

[7] **M**ulti**L**ingual BibTEX. The functions we mention hereafter are used by MlBibTEX to parse a LATEX document's preambule, in order to know which multilingual packages are used [7].

[8] That is, a zero-argument function, w.r.t. Scheme's terminology.

```
(define alttextsc
  (let ((backslash-as-string "\\") ; This escape character must be escaped itself.
        (fake-open-string "\\fake{")
        (fake-close-string "}"))
    (lambda (string-0)
      (let ((r-thunk (mk-r-string-thunk string-0)) ; Builds a character-generator along string-0.
            (x (lambda (y) #f)))                   ; This function will be called by t-next-token in case
                                                   ; of an error. y is supposed to be an error label.
        (let loop ((char (r-thunk)) ; Launches the parsing of the string.
                   (acc '()))
          (let*-values (((ttoken char-0) (t-next-token r-thunk char x))) ; 2 values returned: the next
                        ; token — #f, a character, or a string — and the character after.
            (if ttoken     ; W.r.t. MlBibTeX's terminology, 'ttoken' is for 'TeX token.
                (loop char-0
                      (if (char? ttoken)
                          (if (char-lower-case+? ttoken)
                              (cons* fake-close-string (string (char-upcase+ ttoken))
                                     fake-open-string acc)
                              (cons (string ttoken) acc))
                          ;; Otherwise, ttoken is a command's name, given as a string. We restore the '\'
                          ;; character before this name.
                          (cons* ttoken backslash-as-string acc)))
                ;; If ttoken is the 'false' value #f, this means that we have reached the string's end. To get the
                ;; final result, all the strings of the accumulator acc are concatenated by using reverse order.
                (string-concatenate-reverse/shared acc))))))))
```

Figure 2: Generating a source text typeset using 'false' small capitals.

begins with the '\' character; if the end of the input or string port has been reached, the two values are equal to `#f`;

- the `char-lower-case+?` (resp. `char-upcase+`) function extends the `char-lower-case?` (resp. `char-upcase`) function — dealing to the 'basic' ASCII[9] encoding — to the Latin 1 encoding[10]; they should be replaced by Unicode-compliant functions in Scheme's future versions.

This implementation of the `alttextsc` command — as *pre-processing* in Scheme, the result being processed by LaTeX — could be improved by putting as few occurrences of the `\fake` as possible. Of course, it requires some knowledge of Scheme, but it seems to us to be not as heavy as TeX's. Moreover, we think this point holds on with most modern programming languages, provided that they can use a TeX-like parser.

2 Requirements for an interface

Roughly speaking, if we are interested in doing some pre-processing about a fragment of a TeX document, we do not have to pay attention to some subtleties related to the application of macros and commands by TeX, we do not have to be careful with commands related to *control features* like `\expandafter` [14]. We do not have to be careful with the space characters ending a command's name and gobbled by this command, either[11]. But let us recall that such a 'mini' TeX parser must rule out comments. More precisely, it must skip the characters located between a '`%`' sign and the end of the current line, these two delimiter characters belonging to the sequence to be skipped. Space characters at the beginning of a line must be skipped, too.

Values returned by a TeX-like tokeniser may be three types: the character type[12] for 'simple' characters — in particular, contents of paragraphs to be typese — the type implementing commands' names, and a one-valued type for the parsing process' end. To do that, our 'mini' TeX parser respectively uses characters in Scheme, strings in Scheme, and the false value '`#f`'. As an alternative implementation,

[9] **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange.

[10] In [6], we explain why this encoding is internally used by MlBibTeX's present version.

[11] So does the 'mini' TeX parser developed as part of MlBibTeX (cf. Footnote 7, p. 1002). On the contrary, the parser of bibliography data (.bib) files, built by means of analogous methods, *expands* some (LA)TeX commands. Examples are given by accent commands whose result belongs to the Latin 1 encoding: '`\'{e}`' is expanded into the '`é`' character within internal forms used by MlBibTeX.

[12] Possibly Latin 1, Latin 2, ... or Unicode-compliant, according to what you would like to parse.

let us mention that we have written another TEX-like tokeniser, only experimental, using the object-oriented Ruby programming language [15].

3 Conclusion

In the past, some programming languages were designed to be universal, that is, to serve all purposes. All of these languages PL/1 [8], Ada [1] have failed to be accepted as filling this role. Nowadays, only the C programming language [12] is still used for a very range of applications. But this language can be viewed as a high-level assembly language. In particular, handling strings in C may be quite tedious. On the contrary, TEX's language, like most specialised languages, is very efficient inside its domain, quite unsuitable outside. It offers efficient primitives for advanced typesetting (`\ifcat`, `\ifmmode`, ...) [14] but programming a sort procedure is nightmare.

The cooperation among procedures written using different programming languages is not new: for example, Ada's second version [20] provides interfaces with other programming languages (e.g., C). The new typesetting engine LuaTEX [2] belongs to this framework. Work related to typesetting is done by TEX, whereas other tasks may be *delegated* to procedures written using Lua [9]. But this cooperation is unidirectional: LuaTEX can run procedures in Lua, but these procedures cannot call TEX's commands. TEX's major drawback is that it recognises only its own formats. We hope that the mini-parsers we wrote will indirectly make TEX's features open to other environments.

4 Acknowledgements

Many thanks to Jerzy B. Ludwichowski, who has translated the abstract and keywords in Polish.

References

[1] ANSI: *The Programming Language Ada® Reference Manual*. Technical Report ANSI/MIL-STD-1815A-1983, American National Standard Institute, Inc. LNCS No. 155, Springer-Verlag. 1983.

[2] Hans Hagen: "The Luafication of TEX and ConTEXt". In: *Proc. BachoTEX 2008 Conference*, pp. 114–123. April 2008.

[3] Lars T. Hansen: *Syntax for Receiving Multiple Values*. March 2000. `http://srfi.schemers.org/srfi-11/`.

[4] Jean-Michel Hufflen: "MlBibTEX in Scheme (*First Part*)". *Biuletyn* GUST, Vol. 22, pp. 17–22. In *BachoTEX 2005 conference*. April 2005.

[5] Jean-Michel Hufflen: "TEX's Language within the History of Programming Languages". *Biuletyn* GUST, Vol. 22, pp. 23–32. In *BachoTEX 2005 conference*. April 2005.

[6] Jean-Michel Hufflen: "Managing Order Relations in MlBibTEX". TUG*boat*, Vol. 29, no. 1, pp. 101–108. EuroBachoTEX 2007 proceedings. 2007.

[7] Jean-Michel Hufflen: "Managing Languages within MlBibTEX". TUG*boat*, Vol. 30, no. 1, pp. 49–57. July 2009.

[8] IBM System 360: *PL/1 Reference Manual*. March 1968.

[9] Roberto Ierusalimschy: *Programming in Lua*. 2nd edition. Lua.org. March 2006.

[10] ISO: *Programming Languages—C++*. Technical Report ISO/IEC 14882:2003, ISO/IEC. 2003.

[11] Richard Kelsey, William D. Clinger, Jonathan A. Rees, Harold Abelson, Norman I. Adams iv, David H. Bartley, Gary Brooks, R. Kent Dybvig, Daniel P. Friedman, Robert Halstead, Chris Hanson, Christopher T. Haynes, Eugene Edmund Kohlbecker, Jr, Donald Oxley, Kent M. Pitman, Guillermo J. Rozas, Guy Lewis Steele, Jr, Gerald Jay Sussman and Mitchell Wand: "Revised[5] Report on the Algorithmic Language Scheme". HOSC, Vol. 11, no. 1, pp. 7–105. August 1998.

[12] Brian W. Kernighan and Denis M. Ritchie: *The C Programming Language*. 2nd edition. Prentice Hall. 1988.

[13] Marie-Paule Kluth : FAQ *LATEX française pour débutants et confirmés*. Vuibert Informatique, Paris. Également disponible sur `CTAN:help/LaTeX-FAQ-francaise/`. Janvier 1999.

[14] Donald Ervin Knuth: *Computers & Typesetting. Vol. A: The TEXbook*. Addison-Wesley Publishing Company, Reading, Massachusetts. 1984.

[15] Yukihiro Matsumoto: *Ruby in a Nutshell*. O'Reilly. English translation by David L. Reynolds, Jr. November 2001.

[16] Frank Mittelbach and Michel Goossens, with Johannes Braams, David Carlisle, Chris A. Rowley, Christine Detig and Joachim Schrod: *The LATEX Companion*. 2nd edition. Addison-Wesley Publishing Company, Reading, Massachusetts. August 2004.

[17] Olin Shivers: *List Library*. October 1999. `http://srfi.schemers.org/srfi-1/`.

[18] Olin Shivers: *String Library*. December 2000. `http://srfi.schemers.org/srfi-13/`.

[19] George Springer and Daniel P. Friedman: *Scheme and the Art of Programming*. The MIT Press, McGraw-Hill Book Company. 1989.

[20] S. Tucker Taft and Robert A. Duff, eds.: *Ada 95 Reference Manual. Language and Standard Libraries*. No. 1246 in LNCS. Springer-Verlag. International Standard ISO/IEC 8652:1995(E). 1995.