Extracting information from (LA)TEX source files

Jean-Michel HUFFLEN

Abstract

We present some tools that allow us to parse all or part of (LA)TEX source files and process suitable information. For example, we can use them to extract some metadata of a document. These tools have been developed in the Scheme functional programming language. Using them requires only basic knowledge of functional programming and Scheme. Besides, these tools could be easily implemented using a strongly typed functional programming language, such as Standard ML or Haskell.

0 Introduction

In many places, it has been told or written that TEX is a wonderful tool for typesetting texts. But it deals only with its own formats: that is well-known, too. However, the information contained in source file texts processed by TEX — or any format or engine built out of it — may be of interest for purposes other than typesetting, e.g., enriching the metadata usable by Web search engines.

Doing such jobs by means of (LA)TEX commands arranged into an option of a class or a package is possible, but we think that this is *misusing* TEX. From our point of view, this tool does not aim to be a universal multi-task program, able not only to typeset texts, but also to generate Web pages or fulfill any other purpose we can imagine. From a point of view related to theoretical computer science, TEX's language has the same expressive power as a Turing machine, so any function can be programmed using TEX's primitives,[1] but as with any specialised language, using it for a purpose other than its intended one is tedious.[2] In addition, this language's syntax is old, its parsing uses old-fashioned conventions, it does not provide advanced data structures, as we can find in many more recent programming languages.

Hereafter we describe a way to connect Scheme functions to TEX commands when a (LA)TEX source file is parsed and these commands recognised. Our basic idea is that often only a little information is relevant, e.g., the metadata of a document. Extracting them from (LA)TEX source files allows us to avoid *information redundancy*. Section 2 explains the origins and reasons for our choices, discussed further in Section 3. Reading this article requires only basic knowledge about TEX and LATEX commands [14, 18] and the division of a LATEX source file into a *preamble* and *body*. Some basic notions of programming in Scheme are needed, too, as can be found in any good introductory book to this functional programming language, e.g., [23].

[1] Interested readers can consult [2, 19] about this subject.

[2] As another accurate example, any programmer knows that using Prolog [4] outside logic programming is quite painful.

1 Our Scheme library

1.1 Why Scheme?

As mentioned above, we aim to extract accurate information from (LA)TEX source files; we are not interested in processing the whole of such a file; we do not want to put a 'new TEX program' into action.

Now let us recall that in functional programming, functions are first-class objects, just like other data. So functions can be arguments or results of a computation. This feature allows us to write *generators* of functions. Our tool is a wonderful example of such a generator. You choose which information you would like to retain and how you plan to process it. This step is done by a computation which returns a function. This second function's argument is the input filename to be parsed.

In Section 2, we will see that some parts have already been written using Scheme [22] for several years. Let us recall that within this functional programming language — as within any Lisp dialect — data and programs have the same format. Hereafter, the description of our library's main features emphasises that functions and other data are mixed by means of a unique format.

1.2 How to use our library

Building a function parsing a (LA)TEX source file is done by the construct:[3]

```
(g-mk-tex-parsing-f directive ...)
```

with any number of *directives*.[4] There are *two* kinds of directives:

```
(g-retain-command command-name arg-nb
                  optional-arg? top-level?
                  recursive? preamble?
                  occ-nb-info function)
(g-retain-match command-name s top-level?
                recursive? preamble?
                occ-nb-info function)
```

[3] Let us recall that Scheme systematically uses *prefixed* syntax. All the definitions introduced by our library are prefixed by 'g-'.

[4] This is the terminology used within our source files. You can use `g-mk-tex-parsing-f` without arguments — that is, *no directive* — in which case the result will just move along the file's preamble without performing any other operation.

doi.org/10.47397/tb/43-2/tb134hufflen-extract

where:

command-name is the name of the command to be caught, without the initial '\' character;

arg-nb is the argument number for this command;

optional-arg? is true[5] if the first argument is optional, surrounded by square brackets,[6] false otherwise;

top-level? is true if we have to look for this command only at the top level, false otherwise;

recursive? is used when `\input` commands are encountered: if it is true, corresponding files are searched recursively, otherwise such an `\input` command is just skipped;

preamble? stops searching after a preamble if it is bound to true; otherwise, search goes on;

occ-nb-info may be bound to:

- 0 or the false value: we check that this command does not appear within files;
- a positive integer n: the first n occurrences of this command are processed, and following ones are ignored;
- the true value: all the occurrences of this command are processed;

function the Scheme function to call; it *must* accept the same number of arguments than the *command-name* command. All the arguments of such a function are supposed to be *strings*.

We can see that the directives introduced by the `g-retain-command` function are suitable for most LaTeX commands, possibly with a leading optional argument. More difficult cases are handled by the `g-retain-match` function: its second argument is the command's *pattern*, given as a string, according to TeX's conventions used by the `\def` primitive, the command's name being omitted. Here are two examples:

`\csname` ← `"#1\endcsname"`
`\ifx` ← `"#1#2#3\else#4\fi"`

All the other arguments of this `g-retain-match` function have the same meaning as the namesake arguments of `g-retain-command`. Let us notice that `g-retain-match` and `g-retain-command` are functions, whereas `g-mk-tex-parsing-f` is a *macro*.[7]

[5] Let us recall that the boolean values *true* and *false* are expressed in Scheme by the expressions `#t` and `#f` respectively.

[6] That is, according to LaTeX's conventions [15].

[7] Let us recall that Scheme uses a *call-by-value* strategy for functions: arguments are evaluated before applying the function. Defining `g-mk-tex-parsing-f` as a macro allows us to install the structures we need, before applying the directives to populate these structures, and finally building the parsing function. The process put into action by that macro may be viewed as a kind of *compiling*.

The result of `g-mk-tex-parsing-f` is a function that applies to a filename. It parses this file by performing *one* pass and returns:

`false` if something went wrong, or a forbidden command is included into the file;

`true` in all other cases.

You have to use Scheme functions interfaced with TeX constructs to update your own structures when a file is parsed. Beware that if an error occurs, these structures may be in an inconsistent state.

1.3 Other functions

Scheme's initial library and our basic functions include a rich set of functions dealing with strings. For example, *s* being a string:

`(normalize-space` *s*`)` whitespace-normalises the *s* string, that is, leading and trailing spaces are stripped, multiple occurrences of whitespace are replaced by a single space character; the result is a newly allocated string.

The next two functions can be useful to destructure an argument of a TeX command; the successive characters of the s_0 string are supposed to be a comma-separated list, s_1 is any string:

`(g-parse-to-list` s_0`)` returns its elements within a linear list, e.g.:

```
(g-parse-to-list "New-York, New-York")
    ⟹ ("New-York" "New-York")
```

`(g-parse-to-alist` s_0 s_1`)` returns the successive pairs *key=value* of s_0 within an *association list*; if a key is given without a value, this missing value is replaced by s_1, e.g.:

```
(g-parse-to-alist "town=LA,state" "CA")
    ⟹ (("town" . "LA")
        ("state" . "CA"))
```

In both cases, the original order is preserved.

1.4 A simple example

As a simple example, let us consider a source text for LaTeX. We would like to know:

- its title,
- the options given to the babel package[8] [18, Ch. 9] if it is loaded,
- the number of occurrences of the `\emph` command.

The function we build and run is given in Fig. 1. Some remarks:[9]

[8] We do not consider the '`main=...`' construct.

[9] You may notice that we specify the commands of interest in alphabetical order. This is just a personal habit; the order of directives inside the `g-mk-tex-parsing-f` macro is irrelevant.

```
(define tug-2022-example-emph-occ-nb '0)                     ; Initialisations needed.
(define tug-2022-example-language-name-list '*dummy-value*)  ; (...)
(define tug-2022-example-title '*dummy-value*)

(define tug-2022-example-function
  (g-mk-tex-parsing-f (g-retain-command "emph" 1 ; One argument.
                                        #f       ; No optional argument.
                                        #f       ; May be located at any level.
                                        #f       ; Look for it recursively.
                                        #f       ; Search the preamble and body.
                                        #t       ; Process all the occurrences of this command.
                                        (lambda (ignored-s) ; The argument is ignored.
                                          (set! tug-2022-example-emph-occ-nb
                                                (+ tug-2022-example-emph-occ-nb 1))))
                      (g-retain-command "title" 1 #f ; One non-optional argument.
                                        #t           ; Top level only.
                                        #f           ; Recursively search.
                                        #f           ; Search the preamble and body.
                                        1            ; Process only the first occurrence.
                                        (lambda (title-s)
                                          (set! tug-2022-example-title (normalize-space title-s))))
                      (g-retain-command "usepackage" 2 #t ; Two arguments, including an optional one.
                                        #t                ; Search only at the top level.
                                        #t                ; Recursive search.
                                        #t                ; Search only the preamble.
                                        #t                ; Process all the occurrences.
                                        (lambda (option-s package-names-s)
                                          (when (member "babel" (g-parse-to-list package-names-s)
                                                        string=?)
                                            (set! tug-2022-example-language-name-list
                                                  (g-parse-to-list option-s)))))))

(tug-2022-example-function "⟨this article's source file⟩") ⟹ #t ; Parsed successfully!

tug-2022-example-emph-occ-nb         ⟹ 39                    ; Variables updated.
tug-2022-example-language-name-list  ⟹ ("french" "english")  ; (...)
tug-2022-example-title               ⟹ "Extracting Information from \AllTeX\ Source Files"
```

Figure 1: Example of using our Scheme functions.

- the `\title` command may or may not be given in the preamble, but is unique;
- if babel package is loaded, it can only be located in the preamble; but there may be *several* `\usepackage` commands, possibly for other packages;
- the innermost occurrences of the `\emph` command are processed first: some additional details about this point are given in App. A.

The evaluation given in Fig. 1 applies to the source of the present text. The three Scheme variables used are initialised at Fig. 1's top.

1.5 Types used

Scheme is a *dynamically typed* language. This property allows variables to be bound to a value being any type, *a priori*. Scheme is not *strongly* typed, since variables are not given types, as in the C programming language [13]. This feature may be viewed as an advantage or drawback, depending on programmers' feelings. However we mention that our tool could be implemented using a strongly-typed functional programming language, such as Standard ML [20] or Haskell [21]. Let us recall that programmers of these languages do not have to put down the types associated with variables, but a type-checking mechanism is in charge of determining such types. If this operation fails, your program is rejected. So in practice, programmers of these languages pay great attention to types used.

When arguments of our directives are strings or booleans — true or false values — there is no problem. The information about the number of occurrences to be processed can be viewed as the *union* of natural numbers and boolean values. Since these two sets are disjoint, modern strongly-typed functional programming languages can implement such a construct by means of a *disjoint union*:[10]

$$\textit{Occ-nb-info-type} \stackrel{\text{def}}{=} \textit{Boolean} \uplus \textit{Natural}$$

The type of the functions connected to TEX commands can be specified by a direct sum, too, due to a limitation of TEX. Let us consider that all the possible results of such a function are encompassed into a type called '*Result*'. Let n be a natural number, the type of a function associated with a n-argument command is $\textit{String}^n \rightarrow \textit{Result}$, where '*String*' is the type of strings.[11] Since the greatest argument number for a TEX command is '#9' [14],[12] the complete functions are finally of the type:

$$\textit{Function-for-TEX} \stackrel{\text{def}}{=} \biguplus_{0 \leq i < 10} (\textit{String}^i \rightarrow \textit{Result})$$

2 History

2.1 *Genesis*

Let us recall that we implemented MlBibTEX[13], a possible successor of BibTEX, the bibliography processor that was commonly associated with LATEX for a long time. In particular, MlBibTEX has aimed to ease the production of *multilingual* bibliographies.

When we put MlBibTEX's first public version into action [9], we realised that we needed to parse the beginning of source .tex files, in order to get the way to process the languages used throughout a document; this information was not given in .aux files.[14] There was at most one occurrence of loading the babel package or an *ad hoc* package such as french or polski.[15] Such a load order could be located in a subfile grouping the packages for the set up of a document. On another point, we did not have to parse the whole of a LATEX document: we stopped either after encountering such a load order, or encountering '\begin{document}', that is, at the end of the document's preamble. When we designed the second version [10], we needed to get the encoding used through a document. To do that we proceeded in an analogous way. In other words, we had already created a kind of *mini-TEX* parser, possibly recursive.

2.2 *Apotheosis*

In December 2020, we became the new editor of the *Cahiers GUTenberg*, the journal of the French-speaking TEX user group.[16] For many reasons, we decided to revise the class used for this journal and discovered that the previous version was used to build other files, such as metadata for Web search engines. On another point, we also decided to automate as many tasks as possible. For example, we plan to extract the information about the title, author(s), and pages from each article's source file, in order to build the table of contents of an issue. In addition, we wished to check the succession of page numbers for successive articles.

We did not implement the production of metadata from issues of *Cahiers GUTenberg*. But we adapted our mini-parser into a library customisable as shown in §1.2 and we succeeded in generating automatically the table of contents of [1], although several engines were used for separate articles.

3 Discussion

Coupling engines based on TEX's kernel with a more modern programming language has shown increased interest for more than a decade. The best-known example is LuaTEX [7], where the engine can call procedures written using the Lua language [12], other experiments connect TEX with Python [16]; applications based on such a *modus operandi* can be found in [17, 24].

Using functions written using the Lua programming language — as allowed by LuaLATEX — for the tasks described in §2.2 was impossible: some articles of [1] needed pdfTEX or XƎLATEX, and compiling them with LuaLATEX crashed. Besides, we confess that we were not disappointed. Extracting metadata from a source text is not tightly tied to typesetting texts — so it should work regardless of the engine used — and should be performed by a separate program.

An alternative could be given by the use of *regular expressions*[17] for most cases. However, let

[10] Let S_0 and S_1 be two sets, the **disjoint union** [8] of S_0 and S_1 is defined by:

$$S_0 \uplus S_1 \stackrel{\text{def}}{=} (\{0\} \times S_0) \cup (\{1\} \times S_1)$$

If we connect this formula with an abstract data type definition, '0' and '1' may be viewed as the *constructors* of this data type.

[11] This definition includes zero-argument commands, since a zero-argument function $f_0 : \rightarrow \textit{Result}$ may be viewed as $f_0 : \{\varnothing\} \rightarrow \textit{Result}$, as mentioned by [6]. In programming languages such as Standard ML or Haskell, the $\{\varnothing\}$ set is implemented by the *unit* type, containing only the () value.

[12] There are workarounds if more arguments are needed, as explained in [14]. This point is obviously out of this article's scope.

[13] **M**ulti**L**ingual BibTEX.

[14] Incidentally, BibTEX only parses .aux files and *never* reads .tex files.

[15] At this time, the polyglossia package [3] had not yet come out, and babel did not yet support the Unicode TEX engines.

[16] *GUTenberg : **G**roupe francophone des **U**tilisateurs de TEX.*

[17] Interested readers can consult [5] for a good introduction to this field.

us notice that TeX's conditional and iterative expressions are not balanced as in modern programming languages, as we showed in [11]. So we are not sure that difficult matching cases can be reasonably handled by regular expressions, which are 'naturally' static. In addition, let us recall that our functions resulting from constructs performed by the `g-mk-tex-parsing-f` macro work in one pass, which seems to us to be more efficient than using several regular expressions.

In practice, we have applied such Scheme functions to examples in LaTeX, or close to this format, that is, XeLaTeX or LuaLaTeX. We think we could build functions able to parse plain TeX or ConTeXt documents and extract suitable information from them, in which case the `g-retain-match` function will be used more intensively.

4 Conclusion

Our contribution consists in a bridge between TeX and more 'classical' programming. More experience will be needed in order to evaluate the relevance of our method. We can be told that using our tool requires mastering Scheme. But there is a price to pay for interesting applications outside typesetting texts. In other words, this program is not intended for end-users who just typeset texts. But we think that our tool may be enjoyed by LaTeX users who can program. Finally, we can observe that simple requirements can be put into action easily, as shown for getting an article's title.

Acknowledgments

I thank Denis Bitouzé for his impressions about a first version of this article. I am also grateful to the very efficient proofreaders of *TUGboat*: Karl Berry and Barbara Beeton.

A How (La)TeX files are parsed

You can discover the behaviour of the Scheme functions generated by the `g-mk-tex-parsing-f` macro by choosing some commands judiciously and associating them with functions that *trace* their arguments. Hereafter we give broad outlines of the complete process. Let us recall that a token recognised by TeX may be a command name, a begin or end of a group, or a single character. Some groups of characters can be processed globally, e.g., two or more consecutive occurrences of end-of-line characters, equivalent to the `\par` command.

Our parser processes such tokens in turn. If a command is associated with a Scheme function, its arguments are parsed recursively, either by using the information about the argument number provided by the `g-retain-command` function, or by processing the pattern introduced by the `g-retain-match` function. As soon as these arguments are built, the associated Scheme function is applied to these corresponding arguments. Getting such arguments causes tokens to be processed, so commands located with these arguments will be processed according to a kind of call by value.[18] So if we consider the following example:

```
An \emph❶{emph'd \emph❷{internal} text}.
```

if all the occurrences of the `\emph` command are to be processed, the '. . . ❷' occurrence will be processed first, then the '. . . ❶' occurrence will be processed, according to a leftmost-innermost strategy. Of course, as soon as a Scheme function associated with a command is executed and returns its result, the process of exploring successive tokens in turn is resumed.

References

[1] Association GUTenberg : *Ils sont de retour !*, Vol. 58 de *Cahiers GUTenberg*. Septembre 2021. `https://www.gutenberg-asso.fr/-Cahiers-GUTenberg-`

[2] Pieter Belmans: *TeX is Turing-Complete*. December 2010. Universiteit Antwerpen, `https://pbelmans.files.wordpress.com/2010/12/textalk.pdf`

[3] François Charette, Arthur Reutenauer, Bastien Boucariès and Jürgen Spitzmüller: *polyglossia: Modern Multilingual Typesetting With XeLaTeX and LuaLaTeX*. 18 July 2022. `https://ctan.org/pkg/polyglossia`

[4] William F. Clocksin and Christopher S. Mellish: *Programming in Prolog*. 5th edition. Springer-Verlag. 2003.

[5] Jeffrey E. F. Frield: *Mastering Regular Expressions*. 3rd edition. O'Reilly. August 2006.

[6] George Grätzer: *Universal Algebra*. 2nd edition. Springer-Verlag. 1979.

[7] Hans Hagen: "LuaTeX: Howling to the Moon". *Biuletyn Polskiej Grupy Użytkowników Systemu TeX*, vol. 23, pp. 63–68. April 2006. Also published in *TUGboat* vol. 26, no. 2, pp. 152–157. `https://tug.org/TUGboat/tb26-2/hagen.pdf`

[8] Paul Richard Halmos: *Naive Set Theory*. Undergraduate Texts in Mathematics. Springer-Verlag. 1987.

[9] Jean-Michel Hufflen: "MlBibTeX's Version 1.3". *TUGboat*, vol. 24, no. 2, pp. 249–262. July 2003. `https://tug.org/TUGboat/tb24-2/tb77hufflen.pdf`

[18] If such commands are to be processed. Let us recall that we can restrict our process to work at the top level for a precise number of occurrences.

[10] Jean-Michel Hufflen: "MlBibTEX Now Deals with Unicode". In: Tomasz Przechlewski, Karl Berry and Jerzy B. Ludwichowski, eds., *Premises, Predilections, Predictions. Proc. TUG@BachoTEX 2017*, pp. 39–41. April 2017. Also published in *TUGboat* vol. 38, no. 2, pp. 245–248. `https://tug.org/TUGboat/tb38-2/tb119hufflen-mlbibtex.pdf`

[11] Jean-Michel Hufflen: "Which Success for TEX as an Old Program?". *ArsTEXnica*, vol. 30, pp. 24–30. In Proc. GUIT 2020 meeting. October 2020.

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

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

[14] Donald Ervin Knuth: *Computers & Typesetting. Vol. A: The TEXbook*. Addison-Wesley Publishing Company, Reading, Massachusetts. 1986.

[15] Leslie Lamport: *LATEX: A Document Preparation System. User's Guide and Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts. 1994.

[16] Mark Lutz: *Programming Python*. O'Reilly & Associates. October 1996.

[17] Henri Menke: "Parsing Complex Data Formats in LuaTEX with LPEG". *TUGboat*, vol. 40, no. 2, pp. 129–135. In Proc. TUG. 2019. `https://tug.org/TUGboat/tb40-2/tb125menke-lpeg.pdf`

[18] 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.

[19] Walter Moreira: *A Turing Machine in TEX*. April 2004. Montevideo, Uruguay. `http://www.cmat.edu.uy/%7Ewalterm/turing/turing.html#download`

[20] Lawrence C. Paulson: *ML for the Working Programmer*. 2nd edition. Cambridge University Press. 1996.

[21] Simon Peyton Jones, ed.: *Haskell 98 Language and Libraries. The Revised Report*. Cambridge University Press. April 2003.

[22] Alex Shinn, John Cowan, and Arthur A. Gleckler, with Steven Ganz, Aaron W. Hsu, Bradley Lucier, Emmanuel Medernach, Alexey Radul, Jeffrey T. Read, David Rush, Benjamin L. Russel, Olin Shivers, Alaric Snell-Pym and Gerald Jay Sussman: *Revised^7 Report on the Algorithmic Language Scheme*. 6 July 2013. `https://small.r7rs.org/attachment/r7rs.pdf`

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

[24] Uwe Ziegenhagen: "Combining LATEX with Python". *TUGboat*, vol. 40, no. 2, pp. 126–128. In Proc. TUG 2019. `https://tug.org/TUGboat/tb40-2/tb125ziegenhagen-python.pdf`

◇ 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`