Implementing a Bibliography Processor in Scheme

Jean-Michel Hufflen

LIFC (FRE CNRS 2661)
University of Franche-Comté
16, route de Gray
25030 BESANÇON CEDEX
FRANCE
hufflen@lifc.univ-fcomte.fr

Abstract

We report an experience of implementing the MlBibTeX bibliography processor, a re-implementation of BibTeX with particular focus on multilingual features. First we describe the behaviour of this software and explain why we chose Scheme to implement the first public version. Then we give the broad outlines of our implementation and show how we took as much advantage as possible of the main features of Scheme. We also explain what we really missed and suggest some ways to improve these points.

Keywords MlBibTeX, bibliography processor, medium-sized programming in Scheme.

1. Introduction

This article reports an experience of implementing medium-sized software in Scheme. The 'philosophy' related to the definition of this programming language is that 'a very small number of [basic] rules [...] suffice to form a practical and efficient programming language,' as mentioned at the introduction of the current revision of Scheme [24]. So this article is an attempt to show how software can be developed using 'a very small number of basic rules.' Anyway, we do not regret to have developed our software in Scheme, but have some criticisms: we think they are constructive.

The program we have written using Scheme is a *bibliography processor*. More precisely, this is a re-implementation of BibTeX [36], the bibliography processor associated with the LaTeX word processor [29]. Reading this paper does not require precise knowledge of LaTeX and BibTeX, but we provide a brief introduction in order to make our purpose more precise. LaTeX is not an interactive word processor: first, users type a *source file*, then LaTeX processess this source file and produces an output file that can be displayed on a screen or printed on a laser printer. LaTeX, which uses TeX as typeset engine[1] [28], produces high-quality print outputs. In particular, it is able to hyphenate words correctly [28, App. H].

Items of the bibliographical information cited throughout an article typeset with LaTeX can be denoted by an identifier, e.g.:

```
\cite{ziemianski2002a}
```

[33, § 12.2.1]—from a syntactic point of view, LaTeX commands begin with '\' and braces are used to surround arguments—and when BibTeX is used to build the 'References' section of the article, it searches *bibliography files* containing *entries*, e.g.:

```
@INPROCEEDINGS{ziemianski2002a,
        ...}
```

[33, § 13.2] and generates a file containing bibliographical *references* cited throughout the article. When LaTeX runs again, such references are typeset and appear as part of this article.

LaTeX's recent versions have greatly improved multilingual capabilities. For example, the command for specifying the beginning of a chapter and its title is:

```
\chapter{...}
```

[33, § 2.2] and the keyword put by LaTeX defaults to the English one, i.e., 'Chapter,' but can be '*Chapitre*' (resp. '*Kapitel*, ...) for a book written in French (resp. German, ...) In addition, LaTeX is able to switch to accurate patterns for hyphenating non-English words. The babel package[2] [33, Ch. 9] provides multilungal operations for LaTeX. Other packages do that, too, but the most multilingual one is babel, in the sense that this package processes all the natural languages it knows, without giving any privilege to a particular one. Therefore this package is especially suitable for mixing several languages within the same document. For example, if we want to write an article in Polish with some fragments in German, we declare:

```
\usepackage[german,polish]{babel}
```

the last option—here, 'polish'—gives the default language of the document. If we want to write 'Bus to Poznán,' the words 'bus to' being translated in German, we can use the `\foreignlanguage` command of the babel package:

```
\foreignlanguage{german}{Autobus nach} Pozna\'{n}
```

Besides, LaTeX is able to deal with 'foreign' characters, that is, accented letters (e.g., 'ń') and other diacritics, but in this paper, we do not go thoroughly into that, we just use LaTeX commands to produce such characters.

[1] TeX, defined by Donald E. Knuth [28], provides a general framework to format texts. To be fit for use, the definitions of this framework need to be organised in a *format*. There are several formats, the most well-known being LaTeX.

[2] W.r.t. LaTeX's terminology, a **package** is a collection of commands. Something belonging to the Lisp world and close to this notion is the system of *modules* in Common Lisp, controlled by the `*modules*` variable and the functions `provide` and `require` [47, § 11.8].

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

However, BibTeX's present version does not provide as many multilingual features as LaTeX's, even if the insertion of some multilingual aspects has been put into action [33, pp. 733–734 & § 13.5.2]. In fact, the commands of the babel package can be used within the values of BibTeX fields—as we showed in the previous example—and BibTeX will copy these values onto the files it generates. However, this method seems to us to be bad, because users of such bibliographical entries have to load the babel package with all the accurate options in any document. This package operates statically, that is, all the languages possibly used throughout a document must be declared as options of the `usepackage` command, located at the beginning of the document. In other words, using languages that are not selected when the babel package is loaded causes errors. That may be the case if files of bibliographical references use such language identifiers, put down in the bibliography data bases. In addition, using such LaTeX command in bibliography data bases is just a hack and obviously obstructs the generation of bibliographies for output formats other than LaTeX. Given these considerations, we started a new implementation, called MlBibTeX (for '**M**ulti**L**ingual BibTeX'). We roughly describe the behaviour of this program in Section 2 and explain why we have developed it in Scheme. Section 3 presents MlBibTeX's architecture and gives the guidelines of its development. Section 4 reviews what we enjoyed in Scheme and what we missed.

2. MlBibTeX

2.1 Purpose

We sketched BibTeX's behaviour in the introduction. Now we explain why MlBibTeX can be viewed as a 'better BibTeX,' especially for multilingual features.

Let us consider the entry given in Figure 1, concerning a novella included in an anthology. This novella, written in Polish by a Polish writer, is entitled '*Autobus nach Poznán*,' let us remark that two words of this title belong to the German language. If this novella is cited in an article written in Polish, the corresponding reference should look like:

> [1] Andrzej Ziemiański, *Autobus nach Poznań*. [W:] *Zajdel 2002*. Fabryka słów; Lublin 2002; strony 165–238.

as an item belonging to a `thebibliography` environment [33, § 12.1.2]. A *plain* bibliography style is used above, that is, items are labelled by numbers, and first names are not abbreviated. (Other styles exist—for example, within *alpha* styles, the label of an item is formed from the author's name and the year of publication—various examples are given in [33, Table 13.4].) Besides, let us recall that this reference is supposed to be put at the end of a document written in Polish. Let us have a look at the same reference, but within the bibliography of a document in English and showing the items of this bibliography according to English-speaking conventions as far as possible:[3]

> [1] Andrzej Ziemiański. Autobus nach Poznań. In *Zajdel 2002*, pp. 165–238, Lublin, 2002. Fabryka słów. No English translation.

That is, '*[W:]*' is replaced by 'In,' '*strony*' by 'pp.' for 'pages.' It is easy to see that such simple cases can be processed by means of *substitutions*, that is, by means of LaTeX commands for generating keywords—`\bblin`, `\bblpp`, ...—whose effect is language-dependent—'*in*' and '*pp.*' in English. Other cases are subtler. First, the order is not the same: for example, the address of the publisher

[3] W.r.t. MlBibTeX's terminology, such a convention is called *document-dependent approach* [18, § 4].

```
@INPROCEEDINGS{ziemianski2002a,
        AUTHOR = {Andrzej Ziemiański},
        TITLE = {[Autobahn nach] : german {Poznań}},
        BOOKTITLE = {Zajdel 2002},
        EDITION = 1,
        PAGES = {165--238},
        PUBLISHER = {Fabryka Słów},
        ADDRESS = {Lublin},
        NOTE = {[No English translation] ! english},
        YEAR = 2002,
        LANGUAGE = polish}
```

Figure 1. Example of MlBibTeX entry.

is given before its name in an English-speaking bibliography, after it in a Polish-speaking one. Second, the value associated with the `NOTE` field (see Figure 1), the '`[...] ! english`' notation means that the string surrounded by square brackets is put only if the language of the reference is '`english`.' Users could build an entry for a document, usable when the reference is to be put within an English-speaking bibliography, another entry for the same document, but usable within a French-speaking bibliography, and so on. As a consequence, the information common to these entries would be duplicated. The '`[...] ! ...`' construct avoids such a behaviour; more technical details about such switches are given in [18, § 2.3].

To show some difficulty related to the generation of multilingual bibliographies, let us go back to the Polish version of our reference and recall that the title of the `zemianski2002a` entry uses some German words, which are expressed by the '`[...] : german`' notation. To ensure that these words will be properly hyphenated if need be, we can generate the following text:[4]

```
\bibitem{zemianski2002}Andrzej Ziemia\'{n}ski,
\newblock \emph{\foreignlanguage{german}{Autobus
nach} {Pozna\'{n}}}. \bblin\ \emph{Zajdel 2002}.
\newblock Fabryka s{\l}\'{o}w; Lublin 2002;
\bblpp\ 165--238.
```

provided that the document uses the babel package, with at least the `german` option. This document's author may think that he does not need to write in German even if a work using German words in its title is cited. In such a case—the `german` option has not been selected—MlBibTeX does not put the `\foreignlanguage` command:

```
\bibitem{zemianski2002}Andrzej Ziemia\'{n}ski,
\newblock Autobus nach {Pozna\'{n}} ...
```

but some words might be hyphenated incorrectly. Besides, the babel package is not the only way to write texts in Polish: there exists a `polski` package [4, § F.7], in which case other commands should be used. More precisely, here is the text that will cause the 'foreign' (non-Polish) words to be hyphenated correctly when this package is loaded:

```
\bibitem{zemianski2002}Andrzej Ziemia\'{n}ski,
\newblock \emph{{\selecthyphenation{german}Autobus
nach} {Pozna\'{n}}} ...
```

These examples aim to give some idea about the complexity of MlBibTeX's task. The management of the language specifications (`LANGUAGE` field, '`[...] ...`') and multilingual packages is explained in more detail in [21]. Of course, such problems are

[4] Notice the use of the commands `\bblin` and `\bblpp` for the keywords used in bibliographies. We show how to manage them in [20]. The `\emph` command is for texts to be emphasised, the `\newblock` command is used by some document styles [33, Table 7.2, § 12.2.1].

unknown for 'old' BibTeX. As shown in Figure 1, MlBibTeX's syntax extends BibTeX's. Square brackets are syntactic markers in MlBibTeX, 'normal' characters in 'classical' BibTeX. Likewise, the `LANGUAGE` field, specifying the language of an entry for MlBibTeX, is ignored by BibTeX since unused fields are ignored.

2.2 Requirements

When LaTeX processes a document, it produces an output file and puts the information about bibliographical citations in an *auxiliary* file. For example, processing the '`\cite{zemianski2002a}`' citation will cause the '`\citation{zemianski2002a}`' string to be put into an auxiliary file. This file should contain the specification of the bibliography style to be used: e.g., '`\bibstyle{plain}`' for the 'plain' bibliography style. In fact, these auxiliary files are not TeX source files in the sense that they do not contain texts to be typeset, but the tokens these files use are the same from a syntactic point of view (cf. [33, § 12.1.3] for more details). Here is what is to be done by MlBibTeX:

(i) look into an auxiliary file for the keys cited throughout a document and the bibliography style to be used for this document;

(ii) search bibliography files for corresponding entries;

(iii) look into the beginning of the source file in order to get information about the multilingual packages used and try to determine the document's language;[5]

(iv) sort them (the sort used depends on the bibliography style,[6], it may also depends on the document's language);

(v) arrange them according to the bibliography style chosen.

Tasks (i) and (iii) require a TeX parser, whereas Task (ii) requires a parser of bibliography files. (Because of the compatibility mode for 'old' bibliography styles of BibTeX [16], another parser is required for such files, written using the `bst` language [35].) The directives for Tasks (iv) and (v) are put in bibliography style files. We see that such a bibliography processor has to manage several formalisms.

2.3 A bit of story

When we designed and implemented MlBibTeX's first version [13], we decided to develop it in C [25], for sake of efficiency and portability. In fact, we confess that we were surprised when TeX had been reimplemented as a new system $\mathcal{NTS}$[7] [43], programmed in `Java` [23]: it resulted in a program over 100 times slower than TeX [48]. We also were trying to propose an alternative for a program with good reputation of efficiency. We put into action a precise modular decomposition and a precise terminology to name our functions and variables for this first version [14], so using C to develop a program supported by precise methodology seemed to us to be good compromise between efficiency and maintainability.

This first version—we reported the experience we got in [14]—was able to deal with substitutions, that is, commands such as `\bblin` and `\bblpp` (cf. § 2.1). It was also able to process constructs such as '`[...] : ...`' and '`[...] ! ...`.' But when it was ready for use and when we were arranging the interface files—the different values to give to the '`\bbl...`' commands—we became aware that this prototype was not multilingual enough. As an example, putting the different field values concerning the `zemianski2002a` entry in the right order for documents in English and Polish would have led to complicated bibliography styles, very hard to maintain. As we explained in [17], we decided to get rid of the language used by BibTeX for bibliography styles [35]: that is an old-fashioned language, non-modular and only based on handling a stack, as it can be seen in [33, § 13.6]. As abovementioned, a compatibility mode exists [16], but the best way for developing bibliography styles is given by a new language, called `nbst`, for '**n**ew **b**ibliography **st**yles', close to XSLT [52], the language of transformations for XML[8] documents.

2.4 The nbst language

Within this new framework, parsing a bibliographical entry from a `.bib` file results in an XML tree, that is, the `zemianski2002a` entry given in Figure 1 can be viewed as the XML tree:

```
<inproceedings id="zemianski2002a"
               language="polish">
  <author>...</author>
  <title>...</title>
  ...
</inproceedings>
```

Processing such a tree can be done this way by using a *template* of the `nbst` language:

```
<nbst:template match="inproceedings">
  <!-- Putting the reference's label, text omitted. -->
  <nbst:apply-templates select="author"/>
  <nbst:apply-templates select="title"/>
  ...
</nbst:template>
```

Such a template is similar to those used in XSLT, it is invoked when the entry we are processing is rooted by the `inproceedings` element. The two `nbst:apply-templates` elements we mentioned in that sketch aim to look for templates matching the `author` and `title` elements respectively: if such templates exist, they are invoked. The main difference between XSLT and `nbst`: in the latter, a template can be refined for a particular language:

```
<nbst:template match="inproceedings"
               language="polish">
  ...
  <nbst:apply-templates select="title"/>
  ...
</nbst:template>
```

a template with the `language` attribute having higher priority than a template without. So this template is invoked when we are processing an `inproceedings` element for a Polish-speaking bibliography, whereas the template without `language` attribute is invoked in order to format `inproceedings` elements for bibliographies written in languages other than Polish (more precisely, in languages other than those put in all the `language` attributes of the templates matching `inproceedings` elements). This kind of inheritance is applied whenever we are looking for a template. For example, let us consider the following statement:

```
<nbst:apply-templates select="title"/>
```

When it is run, we are looking for a template whose `match` attribute is `title`. First, we are looking for a template whose `language` attribute is associated with the current language. In particular, if this `apply-templates` statement is run from the template with the `language` attribute associated with `polish`, we are looking for a template with `language="polish"`. If such a template exists, it is invoked. If not, a template matching `title` elements without

[5] BibTeX does not need this step, but MlBibTeX does.

[6] For example, the `unsrt` bibliography style of BibTeX leaves the entries unsorted: they are put according to the order of appearance within the document.

[7] **N**ew **T**ypesetting **S**ystem.

[8] Reading this article does not require advanced knowledge about XML. Readers interested in this metalanguage can refer to [38].

language attribute—that is, a default template—is invoked. Such organisation allows us to build several variants, for English- and Polish-speaking bibliographies, as shown in § 2.1. More technical details are given in [18].

Like in XSLT, the values of match attributes belong to the XPath language, used to address parts of an XML document [51]. In fact, our expressions selecting parts of a bibliographical item are very close to XPath's expressions, but we added some functions for operations difficult to perform with the functions provided by the first version of XPath (1.0, the normative document being [51]). For example, using the functions provided by standard XPath to capitalise some words in a title is tedious. In addition, multilingual features require some information included in TEX source files (cf. § 2.2), and are implemented by means of calling external functions: we go thoroughly into this choice in [21]. Obviously, it is preferable for such external functions to be written in a high-level programming language, more precisely, in a language that should ease operations on strings. Such a criterium puts C at a disadvantage: plenty of successful text-processing packages have been written in C, but the memory management is explicit, such operations like concatenation require functions whose use is far from obvious. We cannot require a bibliography style designer to be an experienced programmer in C. So, as we report in [19], we decided to develop MlBIBTEX's first public version (1.3) using Scheme. In particular, this choice allowed us to use the representation of SXML [27] as a Scheme implementation of our XML trees. So the bibliographical entry given in Figure 1 is represented as:

```
(inproceedings (@ (id "zemianski2002a")
                  (language "polish"))
               (author ...) (title ...) ...)
```

In addition, let us recall that nbst programs are XML texts. To parse them, we use SSAX [26], its outputs being SXML expressions. Among other tools related to SXML, we have also gained experience by studying the functions implementing SXPath [27], but have given our own implementation, in order to ease the calls of external functions. Likewise, we wholly put into action the implementation of nbst, as a 'super-XSLT' processor with a kind of inheritance about the language attribute.

2.5 A language accessible by end-users

The choice of a language with XML-like syntax for bibliography styles opens a window towards XML's world and some applications become easier: for example, using nbst to build a HTML file [53] from a bibliography file in order to display its entries on the Web. Or generating bibliographies for documents in DocBook, an XML-based system for writing structured documents [54]. But another problem occurs: it is well-known that many end-users puts LATEX commands inside values of BIBTEX fields, because 'old' BIBTEX itself does not have enough expressive power. We already mentioned this fact in the introduction about commands from the babel package. In fact, it does not matter if LATEX documents are generated—although it can be told that such behaviour makes difficult the sharing of bibliography files among several users because users have to load the same packages as abovementioned—but may cause errors on other cases. For example:

```
TITLE = {{\textsc{la}} Confidential}
```

In such a case—some letters to be typeset using small capitals—our parser of bibliography files can easily process this title by using an element with accurate attributes:[9]

```
<title>
  <asitis>
    <emph emf="no" scf="yes">la</emph>
  </asitis>
  Confidential
</title>
```

since the \textsc command is predefined in LATEX. The problem is more complicated if end-users put commands they have defined themselves, e.g.:

```
TITLE = {{\logo{la}} Confidential}
```

where \logo is a user-defined command meaning that its its argument is an acronym. Such a command may be defined as follows [33, § A.1.2]:

```
\newcommand{\logo}[1]{\textsc{#1}}
```

that is, the \logo command has the same effect than the \textsc command, but it is more readable about its meaning and can be redefined if users wish to change the display of acronyms.

This example shows that if end-users have put some LATEX commands inside values of BIBTEX fields and wish to use MlBIBTEX to output files according to other formats than LATEX, they should be able to specify how their commands have to be processed when bibliography files are parsed and transformed into XML trees. They can do that by means of the define-pattern function of MlBIBTEX, some examples being given in Figure 2. The first example shows how the previous \logo command can be processed: in this case, it is processed like the \textsc command (see the title and emph elements above).

Hereafter we sketch the effect of the define-pattern function, in order to show that end-users can easily customise the transformation of bibliography files into SXML trees. In particular, such a customisation is easy since Scheme allows powerful operations on strings nicely. If a language like C was still used for MlBIBTEX's implementation, this kind of specification would be tedious, or we would have to define a mini-language to do that.

The define-pattern function has two arguments. The first is a string viewed as a **pattern**, following the conventions of TEX for defining commands, that is, the arguments of a command are denoted by '#1,' '#2,' ... (cf. [28, Ch. 20]). If the second argument is a string, it specifies a replacement, the arguments of the corresponding command being processed recursively. The result—that is, the second argument—could be given as an SXML expression, but we wish a particular representation not to occur inside the Scheme code introduced by the define-pattern function: that is why we give it as a string whose content is expressed by means of 'usual' XML syntax.

This simple form can deal with many cases, but not all. If we look at the second example, we see how the \textbf command of LATEX is replaced by an emph element with accurate attributes: using bold face and non-italicised characters. That may be wrong, because \textit{\textbf{...}} produces both bold face and italicised characters[10] in LATEX. More expressive power is needed to deal with such cases. In the developed form of the define-pattern function, the second argument is a zero-argument function that results in a string, which is the replacement of the pattern. When this form is used, all the operations must be explicit within the body of this zero-argument function. In fact, the form:

[9] Hereafter the asitis element means that its contents should not be capitalised or uncapitalised, even if the bibliography style requires that. The emph element and its attributes specifies typographic effects, e.g., using small capitals in this example. Readers interested in a description of elements and attributes used within the XML versions of bibliography files can refer to [15]: that is an earlier version, but changes are slight.

[10] Readers interested in the font management in LATEX can refer to [33, Ch. 7].

```
(define-pattern "\\logo{#1}" "<emph emf='no' scf='yes'>#1</emph>")      ; 'scf' is a flag for 'small capitals.'
(define-pattern "\\textbf{#1}" "<emph emf='no' bff='yes'>#1</emph>")    ; 'bff' is for 'boldface flag.'
(define-pattern "\\textit{#1}" (lambda ()
                                 ;; Notice that the emf attribute of the emph element—a switch between roman and italicised characters—
                                 ;; defaults to yes, the other attributes default to no.
                                 (define-pattern "\\textbf{#2}" "<emph bff='yes'>#2</emph>")    ; Local pattern.
                                 (string-append "<emph>" (pattern-process "#1") "</emph>")))
```

Figure 2. Patterns for LaTeX commands in Scheme.

`(define-pattern` p s`)`

—where p and s are strings—is equivalent to:

```
(define-pattern p
  (lambda () (pattern-process s))
```

the `pattern-process` function belonging to MlBibTeX's program. The body of the function that is the second argument of `define-pattern` may include the specification of *local patterns*, as shown in the third example given in Figure 2. Let us consider the last two patterns shown in this figure: when an occurrence of a `\textbf` command is encountered, the local pattern of the third example is applied inside the argument of a `\textit` command, the 'global' pattern of the second example being applied anywhere else.

3. The program

3.1 MlBibTeX's architecture

In the previous section, we introduced to the main modules of MlBibTeX; now we show how they are put together. Figure 3 pictures MlBibTeX's architecture. This figure emphasises the data flow: given some citation keys extracted from an auxiliary (.aux) file, some bibliography (.bib) files are searched and the result is a list of bibliographical entries, given as SXML data. To do that, MlBibTeX's parser is enriched with a module for dealing with patterns. As shown in Figure 3, some patterns are predefined, some—like the pattern matching the `\logo` command in Figure 2—can be user-defined. The analysis of the .aux file also allows us to get information about a bibliography style. If we do not consider the compatibility mode for old .bst files, bibliography styles are written using the nbst language. These files are parsed using SSAX, grouped and 'semi-compiled,' in the sense that templates are rearranged in order to ease the determination of the template to be invoked when we are moving to a particular element. Each template results in a Scheme function after this pre-processing, and the bibliography processor applies such functions.

Like XSLT [52, § 16], nbst supports '`text`,' '`xml`' and '`html`' output modes.[11] There is also a `LaTeX` mode, taking into account some particular points of LaTeX's syntax. So, only the strings to be output are concerned by the differences between `text` and `LaTeX` modes. As examples—`nbst:text` is used to put a string *verbatim*, like the `xsl:text` element in XSLT—:

- `<nbst:text>%</nbst/text>` yields '`%`' in `text` mode, '`\%`' in `LaTeX` mode (in LaTeX, '`%`' introduces a comment [29, § 2.2.1], so it must be escaped to loose this property),
- `<nbst:text>&#163</nbst:text>`—the character numbered 163—yields this character ('£') in `text` mode and the command to produce it ('`\pounds`') [29, § 3.2.2] in `LaTeX` mode, this command being suitable whatever the encoding used by the word processor is.

As mentioned in § 2.4, the programs in .nbst files can use calls to external functions written in Scheme. That is not heretic: this feature—using external procedures—exists in XSLT. We use such external functions in Scheme to implement operations on strings, to program lexical ordering that depend on natural languages, and to search .tex files for information about the multilingual capabilities allowed by the user of the source files, as shown in Figure 3.

We can be asked for a question: 'why two languages: nbst and Scheme? why have we not used Scheme for the whole of a bibliography style?' Such a conventions would have made MlBibTeX close to the stylesheets written in DSSSL[12] [22], associated with SGML texts. But it was told that programming with DSSSL was difficult for style designers that are not experienced programmers. In fact, DSSSL is not declarative enough, if we compare it to XSLT or nbst. Besides, nbst allows refinements to be put into action without modifying an existing style directly. For example, if a Polish style designer finds out that the default version for a style does not fit the Polish requirements for the layout of a reference for an `inproceedings` entry, such requirements can be implemented by developing additional templates whose the `language` attribute is associated with `polish`. External functions written in Scheme should be used for low-level computation, for examples, for operations dealing with the different characters of a string. In fact, we think that style designers will not have to develop such functions, but they can do that if they wish.

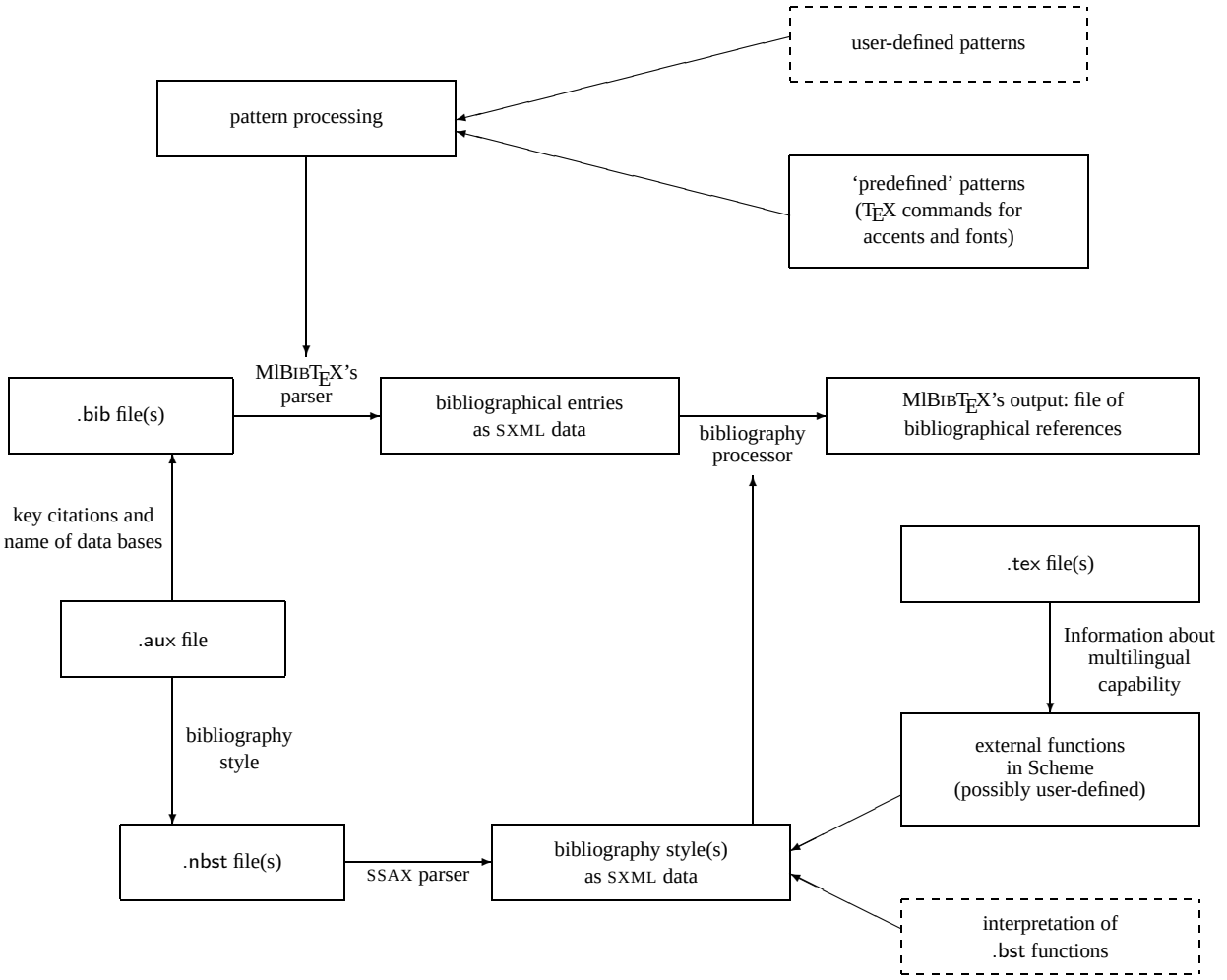
Last but not least, Figure 3 makes precise the parts that are finished presently: all, except that those pictured within a dashed box, they are planned for the next version.

3.2 Our programming

Working about natural languages is an open domain, in the sense that there is no general framework, from a theoretical point of view, that would cover all the natural languages in the world. What is suitable for a particular language may be unsuitable for another. So even if we consider a wide range of natural languages, we have to do experiments and other experiments, reprogram some parts if they have been modelled insufficiently, that is, if some particular cases made fail a general scheme. Only a high-level programming language allows such approach. Besides, the ability for end-users to enrich the program by means of patterns (cf. § 2.5) seemed to us to be a decisive point for choosing Scheme. Let us compare this feature with the Emacs editor, written in Emacs Lisp [31], and customisable by user-defined functions written in this language. Such issues seem to us to justify the choice of a LISP dialect. In addition, when we decided to do a second implementation using another language than C, we were familiar with LISP universe, we have already developed a medium-sized program in COMMON LISP: a rewrite engine for an algebraic specification language [10]. But we noticed that COMMON LISP was too big and heavy. We did not want to accept its complexity, whereas we needed only a

[11] A `html` mode is needed since HTML texts do not fit XML's syntax, stricly speaking.

[12] **D**ocument **S**tyle **S**emantics **S**pecification. This formalism is a side-effect free subset of Scheme, enriched by a library for formatting outputs.



'$A \leftarrow B$' means that A uses B. More precisely, functions or data put in A use functions of B or data from B.

Figure 3. Data flow in MlBIBTEX.

small part of it. We were interested in programming in a simpler LISP dialect, using only a few powerful constructs. In addition, we already have taught Scheme to undergraduate [11] and graduate [12] students.

Here are our rules of programming. Most of them aim to ease maintainability.

- There are precise rules for naming global variables. MlBIBTEX is organised into modules,[13] each module defining a prefix for naming variables. For example, '`pattern-`' is the prefix of the functions dealing with patterns (cf. § 2.5). Here are the exceptions:
  - some general functions and macros, grouped into one file,
  - local variables, that is, variables defined in the body of the special forms `define`, `do`, `lambda`, `let`, `let*`, and `letrec`,
  - *protected variables*, as we will see below, their names always end with '`-pv`;' when they are used in several modules, they do not have any prefix.

[13] From our point of view, these modules only exist in connection to *conception*, we do not use any syntactic feature—e.g., the `module` specification of the Scheme compiler bigloo [40, § 2.2] or PLT Scheme [6, § 5]—for them.

- Side effects are only allowed for local variables. In addition, we have carefully followed the recommendation about naming destructive functions in Scheme [24, § 1.3.5]: if a function mutates any of its arguments, its name ends with '`!`'
- Information is retained locally, by means of lexical closure and unlimited extent as far as possible. If several functionalities share the same environment, they are put into action by one function working by *message-passing*. This technique is used for *protected variables*: they are protected since they are enclosed within a lexical environment. For example, the bibliography style, as a path to a nbst program, is managed this way:

```
((bibliographystyle-pv 'see))       ; Get the value.
((bibliographystyle-pv 'set) ...) ; Update.
```

In fact, this technique can be viewed as object-oriented programming in Scheme, as shown in [1, Ch. 2] and [39]. We could have defined a global variable whose value is such a path and setting it whilst MlBIBTEX is running. We could put a syntactic sign inside its name to warn readers of our program that this variable is supposed to be modified. But we have preferred for

```
(define (parsers-make-launching filename launcher)
  ;; launcher is the function that rules the analysis of the input file. Its arguments are the function going forward through
  ;; the file and the function managing errors.
  (call-with-current-continuation (lambda (parser-exit-c)
                                    (parsers-filename-rp-loop filename launcher parser-exit-c))))

(define (parsers-filename-rp-loop filename launcher parser-exit-c)
  ;; filename being an absolute path to an existing file, opens it, runs a read-and-process loop, and closes the
  ;; corresponding port.
  (let ((input-p '*dummy-value*))
    (dynamic-wind
      ;; Even if the launcher function encounters errors, the input port is closed. The side effect on input-p is allowed
      ;; w.r.t. our conventions, because it is a local variable.
      (lambda ()
        ;; Reenter the middle thunk causes the input file to be open again:
        (set! input-p (open-input-file filename)))
      (lambda () (launcher (make-r-thunk input-p) parser-exit-c))
      (lambda () (close-input-port input-p)))))

(define (make-r-thunk input-p)
  ;; The result is a thunk—zero-argument function—that moves forward through the input file.
  (lambda () (read-char input-p)))

(define (make-x-function parser-exit-c)
  ;; The result is an escape function—'x' is for 'eXit'— that displays an error message, and stops reading through the input file.
  (lambda (msg-idf)
    (msg-manager msg-idf)
    (parser-exit-c #f)))
```

Figure 4. Basic functions to build MlBibTeX's parsers.

all the ways to get the value of such information or update it to be grouped into one function within our program.[14]

We did not use lexer and parser generator like those proposed in [34], analogous to LEX and YACC, which generates C programs [30]. In fact, we could have done that for the bst language, because lexical and syntactic analyses are clearly distinguished in this case. However, there is no distinction between scanner and parser in TeX's language,[15] also used in auxiliary files where information about bibliographies to be build are located (cf. § 2.2). For this language, there is only one analyser, which returns either a whitespace character, or another character, different from '\', or the complete name of a macro. Concerning bibliography files, we can separate lexical and syntactic analysis—we did that in the first version [13, Annex]—but that yields a two-level grammar: a first level for entries ('`@...{...}`'), a second for values associated with field names. So, we have preferred to develop *ad hoc* parsers for these languages. Last, we use the SSAX parser for nbst programs, since they are XML documents.

We have defined a common framework for the parsers we have built ourselves, the main functions are given in Figure 4. By convention, the arguments of the parser's functions include a zero-argument function to move forward through the input file and an escape function stopping reading through the input file.[16] Since this zero-argument function is our only way to get something from the current input file inside the functions of our parser, we do not 'unread' a character.[17] On the other hand, a parser is reading in advance. The solution put into action is that the functions of our parser return at least two values: the result of processing a fragment of the input file, and the first character belonging to the token after what has just been processed. A simple example is given in Figure 5. These parsers were easy to debug: we replaced the function moving forward through an input file by a function given in Figure 6 and exploring successive characters of a string.[18] as the `read-char` function would do after opening a string port in the sense of SRFI[19] Nr. 6 [3].

Concerning the management of multilinguism, the information related to natural languages used throughout bibliography data bases is organised into a *trie*:[20] see [21] for more details.

4. Scheme as an implementation language

First we developed MlBibTeX's present version with MIT Scheme [2, 9]. Then we study how to put a portable implementation into action with bigloo [40] and PLT Scheme [6, 37]. We carefully grouped non-portable code in one file, so we knew which parts could be difficult to adapt.

[14] In addition, if we consider a variable defined globally and updated at run-time, it can be difficult to detect that it has not been assigned yet to its 'actual' value. We could define it by bounding it to a 'dummy value', but there is no 'universal dummy value.'

[15] That is the case for some early languages.

[16] In particular, this function is called when an error is encountered. There is no error recovery in MlBibTeX—our parsers stop as soon as an error is encountered—but there was not in 'old' BibTeX, either.

[17] In fact, we could use the `peek-char` function of Scheme [24, § 6.6.2] for this operation, but we decided to proceed only ahead, homogeneously.

[18] Besides, this function is used in 'final' MlBibTeX: when an abbreviation, defined by '`@STRING{schw = {Scheme Workshop}}`'—cf. [33, § 13.2.3]—is used, e.g., in '`BOOKTITLE = schw`', MlBibTeX's parser inserts the contents of the string associated with '`schw`' by means of the `make-r-string-thunk` function.

[19] Scheme **R**equest **F**or **I**mplementation. For more details, see the Web page `http://srfi.schemers.org`.

[20] A **trie** is a particular case of a tree for storing strings: there is only one node for every common prefix.

```
(define (s-parse-string-def r-thunk char x)
  ;; 's-' is the prefix for functions parsing bibliography files. Parses '@STRING{<token-0> = <string-value>},' '@STRING' being
  ;; recognised, char being the first character after. r-thunk is the 0-argument function that allows us to move forward through the
  ;; input file, x is the escape function that stops reading and returns #f as the global result of parsing.
  (call-with-values (lambda ()
                      (s-next-bibtex-idf r-thunk
                                         ;; Checking that the token beginning with char is '{' and returning the first character
                                         ;; after, in case of success:
                                         (s-recognise-left-brace r-thunk char x)
                                         x))
    (lambda (token-0 char-0)
      :: token-0 is the abbreviation's name, char-0 is supposed to be '='.
      (call-with-values (lambda () (s-parse-value r-thunk (s-recognise= r-thunk char-0 x) x))
        (lambda (string-value char-1)
          ((s-string-defs-pv 'add) token-0 string-value)   ; Adds the binding token-0 ↦ string-value. Let us notice that
                                                           ; s-string-defs-pv is a protected variable (cf. § 3).
          ;; First, recognising '}' and returning the first character after, then processing next entry, that is, next '@{...}' and
          ;; returning two values:
          (s-next-entry r-thunk (s-recognise-right-brace r-thunk char-1 x)))))))
```

Figure 5. How our parsers use multiple values.

Our only error related to portability was an occurrence of the *false* value inadvertently replaced by the empty list.[21] Another portability problem arose from accented letters typed by using an encoding which extends ASCII:

```
(char-alphabetic? #\é) ⟹MIT Scheme          #t
(char-alphabetic? #\é) ⟹bigloo, PLT Scheme  #f
```

In reality, such a case is unspecified by the standard Scheme since this standard does not specify whether or not a character like '#\é' is a letter and since the `char-alphabetic?` function can only be applied to letters. Anyway, porting MlBibTeX raises a very small number of problems, but difficult, because they were related to features outside the standard Scheme. In fact, most of the issues mentioned hereafter are not MlBibTeX-specific and have already been debated, but we mention them, as a short report of our experience and as additional examples of these problems.

4.1 What we have liked

A common pitfall for Scheme programmers is the order of evaluation of a function's arguments: it is left unspecified by the Scheme reports [24, § 4.1.3] and may vary from an interpreter to another in practice. To be honest, the absence of a fixed order may look strange at first glance, but we think that it is straightforward, it forces programmers to emphasise what is sequential within their programs, most often by using the special forms `let` or `let*`.

As far as possible, we use Scheme as a functional programming language, in the sense that functions can be arguments or results of other functions. Since Scheme has only one namespace, that is, functions are particular values for variables, our program looks homogeneous. In COMMON LISP or other LISP dialects where functions belongs to a particular namespace [47, § 5.2], distinct from the 'other' variables, we would have had to add many occurrences of the `function` special form and the `funcall` function, what would complicate the programming.

Advanced functions like `call/cc` and `dynamic-wind` [24, § 6.4] are used in MlBibTeX (cf. Figure 4). However, let us mention that wherever we use these functions, simplified forms, as they are provided by COMMON LISP would have been sufficient: dynamically-scoped exits, by means of the special forms `catch` and `throw` [47, § 7.11], and the special form `unwind-protect`.

[21] Let us recall that in MIT Scheme, `#f` and `()` are still the same object [9, § 1.2.5].

Dealing with multiple values is very common within the source files of MlBibTeX, an example being given in Figure 5, many other examples existing for functions dealing with multilingual information. A new special form such as `let-values`, as suggested by SRFI 11 [8], would simplify these examples.

4.2 What we have missed

The functions dealing with input files, `open-input-file` and `call-with-input-file`, signal an error if the file cannot be opened. But by using only the forms of the Scheme standard, we cannot know this information before trying this operation. The same problem arises from the functions dealing with output files, `open-output-file` and `call-with-output-file`. This can be solved by means of *conditions*—this notion exists in COMMON LISP [47, Ch. 29], but not (yet?) in the standard Scheme[22]—as suggested by SRFI 36 [44].

Some interpreters—MIT Scheme [9, § 5.7], bigloo [40, §§ 4.1.8 & 4.1.10]—allow characters to be processed using Unicode [49], but only partially. That should be added in the future standard, since more and more information will be encoded according to some extensions of the ASCII code: latin-1 (or ISO–8859–1) for West-European languages,[23] latin-2 for East-European ones, ... Unicode precisely redefines what letters, signs are. Proposals for putting these definitions in Scheme are SRFI 14 [41] and 75 [7]. As mentioned at the beginning of this section, some interpreters presently diverge about this point, which should be refined for further versions of Scheme.

We especially missed an interface with the operating system, in the sense of a function that would have launched a command of the operating system, and be able to retain its result displayed on the current output port, this result being a string usable by the functions of Scheme. From a general point of view, we think that in the standard Scheme, such a function would be more useful than special interfaces with specific programming languages like C or Java[24] [40, §§ 15 & 16]. More specifically, software belonging to TeX's world usually call functions of the kpathsea library [50], used for locating files. For example, the bibliography styles used by 'old' BibTeX can be located by means of the `kpsewhich` command:

[22] ... but some Scheme interpreters incorporate them: e.g., MIT Scheme [9, Ch. 16].

[23] Internally used in MIT Scheme [9, § 5.5].

[24] Besides, this function could be used to run the compiled form of a program written using these languages.

```
(define (make-r-string-thunk string-0)
  ;; Returns a thunk exploring each character of string-0, in
  ;; turn. When the end of this string is reached, #f is
  ;; returned.
  (let ((string-length-0 (string-length string-0))
        (index 0))
    (lambda ()
      (if (< index string-length-0)
          (let ((result (string-ref string-0 index)))
            (set! index (+ index 1))
            result)
          #f))))
```

Figure 6. Moving forward through a string.

```
kpsewhich plain.bst
.../texmf/bibtex/bst/base/plain.bst
```

In order to put a similar feature into action for MlBibTeX, a workaround was to implement a simplified version of this command in Scheme. This implementation is not wholly satisfactory from a point of view related to portabiblity because this command uses *environment variables*, inaccessible directly from standard Scheme functions: `BIBINPUTS`, `TEXBIB` for 'old' BibTeX,[25]. MlBibTeX uses first the environment variable `MLBIBINPUTS`, before considering those of BibTeX [20].

Last, Scheme could include *packages* in the sense of Common Lisp [47, § 11.2], a simpler version being sufficient. If we develop software under the predefined functions of Scheme, a good discipline for naming functions is sufficient to avoid name clashes. But packages would ease software composition. For example, there is no document explaining how functions and macros of SXML have been named. So we had to be very careful to this point when we decided to use this software for dealing with XML documents.

4.3 Proposals

In [42], Dorai Sitaram writes that 'the [IEEE] Scheme standard and the Scheme reports do not define a useful programming language for all platforms. Instead they [...] define a family of programming languages that individual implementors can instantiate to a concrete programming language for a specific platform.' That is true, but what does it mean in practice? That an ambitious program rhas to rely on a particular dialect? Such dependence seems to us to be acceptable for a program using special effects (e.g., graphical parts), but is strange for functionalities related to a simple interface with an operating system (e.g., file existence). Besides, each dialect obviously provides such a function, and most often under the same name: `file-exists?` in MIT Scheme [9, § 15.3], in bigloo [40, § 4.2.2], in PLT Scheme [6, § 11.3.3]. Naming them homogeneously should be possible. Other examples are subtler, because functions are not known under the same name: if we wish to get the values of environment variables set at the operating system level (cf. § 4.2), the function is `get-environment-variable` in MIT Scheme [2, § 2.6], `getenv` in bigloo [40, § 4.2.1] and PLT Scheme [6, § 15.4]. Analogous points can be noticed about the functions passing a command to the operating system level.

In the foreword of [45], Guy L. Steele Jr. wrote: '[...] Small is easy to understand. I like the Scheme programming language because it is small.' But Scheme can include a small interface with basic services of operating systems and be still small. Such a small interface would not give Common Lisp's complexity to Scheme. It may be difficult to decide about the names to be given to the functions of this interface, because some software already use some functions specific to particular interpreters, so it would be tedious to rename them. A workaround could be an additional predefined variable whose value would group the whole information about the present interpreter, its name and version number, the running operating system, etc. The purpose of the zero-argument function `identify-world` of MIT Scheme [2, § 2.1] is close, but such information is only displayed when the function is applied and cannot be retained in a variable since this function does not return any result. Such a variable had been defined in Common Lisp: `*features*` returns a list of *features* characterising a particular implementation [47, § 25.4.2]. Features have also been proposed in SRFI 0 [5], that seems to us to be a promised way. In particular, such ways a variable would ease the writing of a tool like SCMXLATE [42], a software for porting Scheme programs from a dialect to another.

[25] However, we had to consider these environment variables for sake of compatibility with BibTeX.

5. Conclusion

When we teach Scheme to undergraduate students, some of them asks us about using this language in 'real' situations. Our personal opinion is that Scheme is certainly less used than an imperative language like C, or a language in fashion like Java. However, some medium-sized projects have been programmed using Scheme, and often the use of this language in such cases was successful. A good illustration of that is MlBibTeX. Doing the second implementation in Scheme was faster than doing the first in C, and performances are comparable. Surely, it is well-known that the higher the programming language's level, the faster the development. And to be honest, many problems had already been specified and solved for the first version, so often adapting C structures to Scheme ones was sufficient. But on the other hand, the second implementation proposes many more functionalities.

At the time of writing, we are working on MlBibTeX's installation, in order for this program to be able to work with a great number of Scheme interpreters. We think that we could succeed by using GNU tools such as make [46] and autoconf [32].

We enjoyed programming MlBibTeX in Scheme. We hope that we could go on with our implementation. We think that we could do better for future versions, especially about processing Unicode characters, according to an interpreter-independent way. So Scheme will be a modern language, since the localisation of software, including the use of several writing systems, is current challenge. Likewise, we hope that installing software programmed using Scheme will become easier. So Scheme will be not only 'an efficient and practical programming language' [24], but it will be more portable and more suitable for the modern types of strings.

Acknowledgements

I am very grateful to the anonymous referees, who allowed me to improve the first version of this article substantially.Many thanks to Michael Sperber, too, for his patience when he was waiting for this article.

References

[1] Harold Abelson and Gerald Jay Sussman: *Structure and Interpretation of Computer Programs*. The MIT Press, McGraw-Hill Book Company. 1985.

[2] Stephen Adams, Chris Hanson and the MIT Scheme team: *MIT Scheme User's Manual*, 1st edition. June 2002.

[3] William D. Clinger: *Basic String Ports*. July 1999. `http://srfi.schemers.org/srfi-6/`.

[4] Antoni Diller: *LaTeX wiersz po wierszu*. Wydawnictwo Helio, Gliwice. Polish translation of *LaTeX Line by Line* with an additional annex by Jan Jelowicki. 2001.

[5] Marc FEELEY: *Feature-based Conditional Expansion Construct*. May 1999. http://srfi.schemers.org/srfi-0/.

[6] Matthew FLATT: PLT *MzScheme: Language Manual. Version 299.100*. March 2005. http://download.plt-scheme.org/doc/299.100/mred.pdf.

[7] Matthew FLATT and Marc FEELEY: *R6RS Unicode Data*. July 2005. http://srfi.schemers.org/srfi-75/.

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

[9] Chris HANSON, THE MIT SCHEME TEAM *et al.*: MIT *Scheme Reference Manual*, 1st edition. March 2002. Massachusetts Institute of Technology.

[10] Jean-Michel HUFFLEN : *Fonctions et généricité dans un langage de programmation parallèle*. Thèse de doctorat, Institut National Polytechnique de Grenoble. Juillet 1989.

[11] Jean-Michel HUFFLEN : *Programmation fonctionnelle en Scheme. De la conception à la mise en œuvre*. Masson. Mars 1996.

[12] Jean-Michel HUFFLEN : *Programmation fonctionnelle avancée. Notes de cours et exercices*. Polycopié. Besançon. Juillet 1997.

[13] Jean-Michel HUFFLEN: "MlBibTeX: a New Implementation of BibTeX". In: *EuroTeX 2001*, pp. 74–94. Kerkrade, The Netherlands. September 2001.

[14] Jean-Michel HUFFLEN: "Lessons from a Bibliography Program's Reimplementation". In: LDTA *2002*, Vol. 65.3 of ENTCS. Elsevier, Grenoble, France. April 2002.

[15] Jean-Michel HUFFLEN: "Multilingual Features for Bibliography Programs: From XML to MlBibTeX". In: *EuroTeX 2002*, pp. 46–59. Bachotek, Poland. April 2002.

[16] Jean-Michel HUFFLEN: "Mixing Two Bibliography Style Languages". In: LDTA *2003*, Vol. 82.3 of ENTCS. Elsevier, Warsaw, Poland. April 2003.

[17] Jean-Michel HUFFLEN: "European Bibliography Styles and MlBibTeX". TUG*boat*, Vol. 24, no. 3, pp. 489–498. EuroTeX 2003, Brest, France. June 2003.

[18] Jean-Michel HUFFLEN: "MlBibTeX's Version 1.3". TUG*boat*, Vol. 24, no. 2, pp. 249–262. July 2003.

[19] Jean-Michel HUFFLEN: "A Tour around MlBibTeX and Its Implementation(s)". *Biuletyn* GUST, Vol. 20, pp. 21–28. In *BachoTeX 2004 conference*. April 2004.

[20] Jean-Michel HUFFLEN: "Making MlBibTeX Fit for a Particular Language. Example of the Polish Language". *Biuletyn* GUST, Vol. 21, pp. 14–26. 2004.

[21] Jean-Michel HUFFLEN: *Managing Languages within MlBibTeX*. Will be presented at PracTeX conference, Chapel Hill, North Carolina. June 2005.

[22] International Standard ISO/IEC 10179:1996(E): DSSSL. 1996.

[23] *Java Technology*. June 2005. http://java.sun.com.

[24] Richard KELSEY and William D. CLINGER, eds.: "Revised^5 Report on the Algorithmic Language Scheme". HOSC, Vol. 11, no. 1, pp. 7–105. August 1998.

[25] Brian W. KERNIGHAN and Denis M. RITCHIE: *The C Programming Language*. 2nd edition. Prentice Hall. 1988.

[26] Oleg KISELYOV: "A Better XML Parser through Functional Programming". In: *4th International Symposium on Practical Aspects of Declarative Languages*, Vol. 2257 of LNCS. Springer. 2002.

[27] Oleg KISELYOV and Kirill LISOVSKY: "XML, XPath, XSLT Implementations as SXML, SXPath, and SXSLT". In: *International Lisp Conference 2002*. San Francisco, California. October 2002.

[28] Donald Ervin KNUTH: *Computers & Typesetting. Vol. A: the TeXbook*. Addison-Wesley Publishing Company, Reading, Massachusetts. 1984.

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

[30] John LEVINE, Tony MASON and Doug BROWN: *lex & yacc*. 2nd edition. O'Reilly & Associates, Inc. October 1992.

[31] Bill LEWIS, Dan LALIBERTE, Richard M. STALLMAND and THE GNU MANUAL GROUP: GNU *Emacs Lisp Reference Manual for Emacs Version 21. Revision 2.8*. January 2002. http://www.gnu.org.

[32] David MACKENZIE, Ben ELLISTON and Akim DEMAILLE: *autoconf. Creating Automatic Configuration Scripts. Version 2.59*. November 2003. http://www.gnu.org/software/autoconf/manual/.

[33] Frank MITTELBACH, Michel GOOSSENS, Joannes BRAAMS, David CARLISLE, Chris A. ROWLEY, Christine DETIG and Joachim SCHROD: *The LaTeX Companion*. 2nd edition. Addison-Wesley Publishing Company, Reading, Massachusetts. August 2004.

[34] Scott OWENS, MATTHEW FLATT, Olin SHIVERS and Benjamin MCMULLAN: "Lexer and Parser Generators in Scheme". In: *Proc.* ACM SIGPLAN *2004 Scheme Workshop*, pp. 41–52. Snowbird, Utah. September 2004.

[35] Oren PATASHNIK: *Designing BibTeX Styles*. February 1988. Part of BibTeX's distribution.

[36] Oren PATASHNIK: *BibTeXing*. February 1988. Part of BibTeX's distribution.

[37] PLT: PLT *MzLib: Libraries Manual. Version 299.100*. March 2005. http://download.plt-scheme.org/doc/299.100/mzlib.pdf.

[38] Erik T. RAY: *Learning* XML. O'Reilly & Associates, Inc. January 2001.

[39] Jonathan A. REES and Norman I. ADAMS IV: "Object-Oriented Programming in Scheme". In: *Proc. of the 1988* ACM *Conference on Lisp and Functional Programming*, pp. 277–288. Snowbird, Utah. 1988.

[40] Manuel SERRANO: *Bigloo. A Practical Scheme Compiler. User Manual for Version 2.6c*. June 2004.

[41] Olin SHIVERS: *Character-set Library*. December 2000. http://srfi.schemers.org/srfi-14/.

[42] Dorai SITARAM: "Porting Scheme Programs". In: *Proc. of the 4th Workshop on Scheme and Functional Programming, UUCS–03–023*, pp. 69–74. School of Computing, University of Utah, Boston, Massachusetts. November 2003.

[43] Karel SKOUPÝ: "The Software Quality and $\mathcal{NTS}$". *GUST*, Vol. 16, pp. 41–49. 2001.

[44] Michael SPERBER: *I/O Conditions*. June 2003. http://srfi.schemers.org/srfi-36/.

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

[46] Richard M. STALLMAN, Roland MCGRATH and Paul SMITH: GNU *make. A Program for Directing Recompilation. Version 3.80*. July 2002. http://www.gnu.org/software/make/manual/.

[47] Guy Lewis STEELE, JR.: COMMON LISP. *The Language. Second Edition.* Digital Press. 1990.

[48] Philip TAYLOR, Jiři ZLATUŠKA and Karel SKOUPÝ: "The $\mathcal{NTS}$ Project: from Conception to Implementation". *Cahiers GUTenberg*, Vol. 35–36, pp. 53–77. May 2000.

[49] THE UNICODE CONSORTIUM: *The Unicode Standard Version 4.0*. Addison-Wesley. August 2003.

[50] TUG Working Group on a TeX Directory Structure: *A Directory Structure for TeX Files. Version 0.9995*. CTAN:tex/archive/tds/standard/tds-0.9995/tds.dvi. January 1998.

[51] W3C: XML *Path Language (XPath). Version 1.0*. W3C Recommendation. Edited by James Clark and Steve DeRose. November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[52] W3C: XSL *Transformations (*XSLT*). Version 1.0*. W3C Recommendation. Edited by James Clark. November 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`.

[53] W3C: *HyperText Markup Language Home Page*. May 2005. `http://www.w3.org/MarkUp/`.

[54] Norman WALSH and Leonard MUELLNER: *DocBook: The Definitive Guide*. O'Reilly & Associates, Inc. October 1999.