Using TEX's Language within a Course about Functional Programming

Abstract
We are in charge of a teaching unit, entitled *Advanced Functional Programming*, for 4th-year university students in Computer Science. This unit is optional within the curriculum, so students attending it are especially interested in programming. The main language studied in this unit is Scheme, but an important part is devoted to general features, e.g., lexical vs dynamic scoping, limited vs unlimited extent, call by value vs call by name or need, etc. As an alternative to other programming languages, TEX allows us to show a language where dynamic and lexical scoping—`\def` vs `\edef`—coexist. In addition, we can show how dynamic scoping allows users to customise TEX's behaviour. Other commands related to strategies are shown, too, e.g., `\expandafter`, `\noexpand`. More generally, TEX commands are related to macros in more classical programming languages, and we can both emphasise difficulty related to macros and shown non-artificial examples. So TEX is not our unit's main part, but provides significant help to illustrate some difficult notions.

Keywords
Functional programming, TEX programming, lexical vs dynamic scope, macros, evaluation strategies.

Introduction

If we consider *programming* in TEX [17], we have to admit that this language is old-fashioned, and programs are often viewed as rebuses, as shown in the *Pearls of TEX programming* demonstrated at BachoTEX conferences[1]. Some interesting applications exemplifying this language can be found in [19, 30], but as noticed in [5], 'some of these programming tricks are ingenious and even elegant. However [...] it is time for a change'.

So, at first glance, it may be strange to use some examples of TEX programming within a present-day course devoted to *Functional programming*. Let us recall that this programming paradigm treats computation as the evaluation of mathematical functions, avoiding state and mutable data as far as possible. Functional programming emphasises functions' application, whereas *imperative programming*—the paradigm implemented within more 'traditional' languages, such as C [16]—emphasises changes in state. Many universities include courses about functional programming, examples being reported in [35]. Besides, such languages are sometimes taught as *first* programming languages, according to an approach comparable to [1, 8, 32] in the case of Scheme.

Let us remark some tools developed as part of TEX's galaxy have already met functional programming: cl-bibtex [18], an extension of BIBTEX—the bibliography processor [26] usually associated with the LaTEX word processor [20]—is written using ANSI[2] COMMON LISP [7]; xindy, a multilingual index processor for documents written using LaTEX [24, § 11.3] is based on COMMON LISP, too; BIBTEX2HTML [4], a converter from .bib format—used by the bibliography database files of BIBTEX—to HTML[3], is written in CAML[4] [21]; MlBIBTEX[5], a re-implementation of BIBTEX focusing on multilingual features [11], is written in Scheme [15]; as another example, Haskell[6] [28] has been used in [38]; last but not at least, there were proposals for developing NTS[7]—a re-implementation of TEX—using CLOS[8], an object-oriented system based on COMMON LISP [39].

The unit we mentioned above is entitled *Advanced Functional Programming*[9], it is an optional unit for 4th-year university students in Computer Science, part of the curriculum proposed at the University of Franche-Comté, at the Faculty of Science and Technics, located at Besançon, in the East of France. Most of these students already know a functional programming language: Scheme, because they attended a unit introducing this language in the 2nd academic year in Computer Science[10]. Other students, who attended the first two university years at Belfort, know CAML. So this unit is not an initiation course, we actually go thoroughly into functional programming.

In next section, we expose the 'philosophy' of our unit. Then we summarise the features of TEX that

```
(define (factorial x)
  ;; Returns x! if x is a natural number, the 'false'
  ;; value otherwise.
  (and (integer? x) (not (negative? x))
       (let tr-fact ((counter x)
                     (acc 1))
         ;; Returns acc * counter!.
         (if (zero? counter)
             acc
             (tr-fact (- counter 1)
                      (* acc counter))))))
```

Figure 1. The factorial function, written using Scheme.

```
(defun factorial (x)
  "Behaves like the namesake function in Scheme
   (cf. Fig. 1)."
  (and
   (integerp x) (not (minusp x))
   (labels ((tr-fact (counter acc)
              ;; The labels special form of
              ;; COMMON LISP introduces local
              ;; recursive functions [33, § 7.5].
              (if (zerop counter)
                  acc
                  (tr-fact (- counter 1)
                           (* acc counter)))))
     (tr-fact x 1))))
```

Figure 2. The factorial function in COMMON LISP.

are useful within this unit and discuss our choice about TEX. Reading this article only requires basic knowledge of programming, readers who would like to go thoroughly into Scheme constructs we have used throughout our examples can refer to [32], very didactic. Of course, the indisputable reference about TEX commands is [17].

Our unit's purpose

Functional programming languages have a common root as the λ-calculus, a formal system developed in the 1930's by Alonzo Church to investigate function definition, function application, and recursion [3]. However, these programming languages are very diverse, some—e.g., the Lisp dialects[11]—are dynamically typed[12], some—e.g., Standard ML[13] [27], CAML, Haskell—are strongly typed[14] and include a *type inference mechanism*: end-users do not have to make precise the types of the variables they use, they are inferred by the type-checker: in practice, end-users have to conceive a program using a strongly typed approach because if the type-checker does not succeed in associating a type with an expression, this expression is proclaimed incorrect. As examples, Fig. 1 (resp. 2) show how to program the factorial function in Scheme (resp. COMMON LISP). In both cases, the factorial function we give can be applied to any value, but returns the factorial of this value only if it is a non-negative integer, otherwise, the result is the 'false' value. Fig. 3 gives the same function in Standard ML: it can only be applied to an integer, as reported by the type-checker (see the line beginning with '>').

A course explaining the general principles of functional programming and an overview of some existing functional programming languages would be indigestible for most students, since they could difficultly get familiar with several languages, due to durations allowed to each unit. In addition, theoretical notions without practice would not be very useful. So, our unit's first part is devoted to the λ-calculus' bases [10]. Then, all the practical exercises are performed with only one language, Scheme, most students already know. Besides, this unit ends with some advanced features of this language: delayed evaluation, continuations, hygienic macros [9]. In addition, this choice allows us to perform a demonstration of DSSSL[15] [13], initially designed as the stylesheet language for SGML[16] texts. These students attended a unit about XML and XSLT[17] [36] the year before, and DSSSL—which may be viewed as XSLT's ancestor—is based on a subset of Scheme, enriched by specialised libraries.

When we begin to program, the language we are learning is always shown as *finite product*. It has precise rules, precise semantics, and is *consistent*. According to the language used, some applications may be easy or difficult to implement. When you put down a statement, running it often results in something predictible. That hides an important point: a language results from some important choices: does it use lexical or dynamic scoping, or both? To illustrate this notion with some examples in TEX, that is the difference between the commands `\firstquestion` and `\secondquestion` in Fig. 4. The former can be related to *lexical* scoping, because it uses the value associated with the `\state` command at definition-time and produces:

You're happy, aint'U?

whereas the latter can be related to *dynamic* scoping, because it uses the value of the `\state` command at run-time and yields:

You're afraid, aint'U?

Students difficultly perceive this notion: some know that they can redefine a variable by means of

```
fun factorial x =
  (* If x is a negative integer, the predefined
     exception Domain is raised [27, §§ 4.5–4.7]. The
     internal function tr_fact is defined by means of
     pattern matching [27, § 4.4].
  *)
  if x < 0 then raise Domain
  else let fun tr_fact 0 acc = acc |
               tr_fact counter acc =
                 tr_fact (counter - 1)
                         acc * counter
       in tr_fact x 1
       end ;
> val factorial = fn : int -> int
```

Figure 3. The factorial function in Standard ML.

a `let` form in Emacs Lisp [22], but they do not realise that this would be impossible within lexically-scoped languages such as C or Scheme. In other words, they do not have *transversal* culture concerning programming languages, they see each of them as an independent cell, a kind of *black box*.

The central part of our unit aims to emphasise these choices: what are the consequences of a lexical (resp. dynamic) scope? If the language is lexical (resp. dynamic), what kinds of applications are easier to be implemented? Likewise, what are the advantages and drawbacks of the call-by-value[18] strategy vs call-by-name one? In the language you are using, what is variables' extent[19]? Of course, all the answers depend on the programming languages considered. But our point of view is that a course based on Scheme and using other examples in TEX may be of interest.

TEX's features showed

As mentioned above, `\def` and `\edef` allow us to illustrate the difference between lexical and dynamic scope. Most of the present-day programming languages are lexical, but we can observe that the dynamic scope allows most of TEX commands to be redefined by end-users. The dynamic scope is known to cause *variable captures*[20], but TEX is protected against undesirable redefinitions by its internal commands, whose names contains the '@' character. Of course, forcing these internal commands' redefinition is allowed by the `\makeatletter` command, and restoring TEX's original behaviour is done by the `\makeatother` command.

If we are interested in implementation considerations, the commands within an `\edef`'s body are expanded, so this body is evaluated as far as possible[21]. To show this point, we can get dynamic

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

Figure 4. Lexical and dynamic scope within TEX.

scope with an `\edef` command by preventing command expansion by means of `\noexpand`:

```
\edef\thirdquestion{%
 You're \noexpand\state, ain't U?\par}
```

and this command `\thirdquestion` behaves exactly like `\secondquestion` (cf. Fig. 4).

A second construct, useful for a point of view related to conception, is `\global`, shown in Fig. 5, because it allows 'global' commands to be defined within local environments. There is an equivalent method in Scheme, but not naturally: see Appendix. Let us go on with this figure, any TEXnician knows that `\thisyear` no longer works if '`\edef`' is replaced by '`\def`'. This illustrates that TEX commands have limited extent.

Nuances related to notion of equality exist in TEX: let `\a` be a command already defined:

```
\let\b\a
\def\c{\a}
```

the former expresses that `\a` and `\b` are 'physically' equal, it allows us to retain `\a`'s definition, even it is changed afterwards, the latter expresses an equality at run-time, it ensures that the commands `\c` and `\a` are identical, even if `\a` changes[22].

Scheme's standard does not allow end-users to know whether or not a variable `x` is bound[23]. A TEXnician would use:

```
\expandafter\ifx\csname x\endcsname\relax...%
 \else...%
\fi
```

For beginners in programming with TEX, this is a quite complicated statement causing commands `\relax` and `\ifx` to be introduced. However, that leads us to introduce not only the construct `\csname...\endcsname`, but also `\expandafter`, which may be viewed as kind of call by value. A simpler example of using this strategy is given by:

```
\uppercase\expandafter{\romannumeral 2009}
```

—which yields 'MMIX'—since this predefined command `\uppercase` is given its only argument as it is; so putting the `\expandafter` command causes this argument to be expanded, whereas removing it would produce 'mmix', because `\uppercase` would leave the group `{\romannumeral 2009}` untouched, then `\romannumeral` would just be ap-

```
{\def\firsttwodigits{20}
 \def\lasttwodigits{09}
 \global\edef\thisyear{%
  \firsttwodigits\lasttwodigits}}
```

Figure 5. Using TeX's `\global` command.

plied to 2009. That is, TEX commands are *macros*[24] in the sense of 'more classical' programming languages.

The last feature we show concerns is related to *mixfixed* terms, related to parsing problems and priorities. TEX can put mixfixed terms into action by means of *delimiters* in a command's argument, like in:

```
\def\put(#1,#2)#3{...}
```

Discuss

As shown in last section, we use TEX as 'cultural complement' for alternative constructs and implementations. Sometimes, we explain some differences by historical considerations: for example, the difference between `\def` and `\long\def`—that is, the difference in LATEX between `\textbf{...}` and `\begin{bfseries}...\end{bfseries}`—comes of performance considerations, since at the time TEX came out, computers were not as efficient as today. Nevertheless, are there other languages that could be successfully used as support of our unit? Yes and no.

An interesting example could be COMMON LISP. Nevertheless, this language is less used now than some years ago, and it is complexified by the use of several *namespaces*[25]. Besides, this language's initial library is as big as possible, it uses old constructs[26]. That is why we give some examples in COMMON LISP, but prefer for our course to be based on Scheme, which is 'the' modern Lisp dialect, from our point of view.

Concerning the coexistence between lexical and dynamic variables, the Perl[27] language [37] provides it. In addition, it has been successfully used to develop large software, so examples could be credible. However, it seems to us that dynamic variables in Perl are rarely used in practice. In fact, the two dynamic languages mainly used are Emacs Lisp and TEX, in the sense that end-users may perceive this point. From our point of view, using examples in Emacs Lisp requires good knowledge about the emacs[28] editor, whereas we can isolate, among TEX's features, the parts that suit us, omitting additional details about TEX's tasks. Likewise, such an approach would be more difficult with Perl.

Conclusion

Our unit is viewed as theoretical, whereas other optional units are more practical, so only a few students attend ours. But in general, students who chose it do not regret that and enjoy it. They say that they have clear ideas about programming after attending it. Some students view our examples in TEX as historical curiosity since this language is quite old and originates from the 1980's, but they are surprised by its expressive power. Some, that are interested in using TEX more intensively, can connect programming in TEX to some concepts present in more modern languages.

Acknowledgements

When I decided to use TEX to demonstrate 'alternative' implementations of some features related to programming, I was quite doubtful about the result, even if I knew that some were interested. But feedback was positive, some students were encouraged to go thoroughly into implementing new TEX commands for their reports and asked me for some questions about that. Thanks to them, they encouraged me to go on with this way in turn.

Appendix: `\global` in Scheme

In this appendix, we show that a construct like `\global` in TEX may be needed. Then we will explain why it cannot be implemented in Scheme using a 'natural' way.

Let us consider that we are handling *dimensions*, that is, a number and a measurement unit—given as a symbol—like in TEX or DSSSL. A robust solution consists of using a list prefixed by a *marker*—e.g., `((*dimension*) 1609344 mm)`—such that all the lists representing dimensions—of type *dimension*—share the same head element, defined once. To put this marker only when the components—a number and a unit[29]—are well-formed, it is better for the access to this marker to be restricted to the functions interfacing this structure. So here is a proposal for an implementation of functions dealing with dimensions:

```
(let ((*marker* '(*dimension*)))
  (define (mk-dimension r unit)
    ;; Performs some checks and creates an
    ;; object of type dimension, whose
    ;; components are r and unit.
    ...)
  (define (dimension? x)
    ;; Returns #t if x is of type dimension, #f
```

```
(define mk-dimension)
(define dimension?)
(define dimension->mm)

(let ((*marker* '(*dimension*)) ; Only the cell's address is relevant.
      (allowed-unit-alist
       '((cm . ,(lambda (r) (* 10 r))) ; Each recognised unit is associated with a function giving the
         (mm . ,values))))             ; corresponding length in millimeters.
  (set! mk-dimension
        (let ((allowed-units (map car allowed-unit-alist)))
          (lambda (r unit)
            (and (real? r) (>= r 0) (memq unit allowed-units) (list *marker* r unit)))))
  (set! dimension? (lambda (x) (and (pair? x) (eq? (car x) *marker*))))
  (set! dimension->mm
        (lambda (dimension-0) ; dimension-0 is supposed to be of type dimension.
          ((cdr (assq (caddr dimension-0) allowed-unit-alist)) (cadr dimension-0)))))
```

Figure 6. Global definitions sharing a common lexical environment in Scheme.

```
    ;; otherwise.
    ...)
  (define (dimension->mm dimension-0)
    ;; Returns the value of dimension-0,
    ;; expressed in millimiters.
    ...))
```

Unfortunately, this does not work, because `define` special forms inside the scope of a `let` special form are viewed as *local* definitions, like `\def` inside a group in TEX. So, `mk-dimension`, `dimension?`, and `dimension->mm` become unaccessible as soon as this `let` form is processed. The solution is to define these three variables globally, and modify them inside a local environment, as shown in Fig. 6.

This *modus operandi* is quite artificial, because it uses side effects, whereas functional programming aims to avoid such as far as possible. But in reality, from a point of view related to conception, there is no 'actual' side effect, in the sense that variables like `mk-dimension`, `dimension?`, and `dimension->mm` would have been first given values, and then modified. The first bindings may be viewed as *preliminary declarations*[30]; however, using 'global' declarations for variables introduced within a local environment would be clearer, as in TEX. To sum up, such an example illustrates that some use of assignment forms are not related to actual side effects, and TEX's `\global` command allows us to explain how this example could appear using a 'more functional' form, without any side effect[31].

References

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

[2] Neil BRADLEY: *The Concise* SGML *Companion*. Addison-Wesley. 1997.

[3] Alonzo CHURCH: *The Calculi of Lambda-Conversion*. Princeton University Press. 1941.

[4] Jean-Christophe FILLIÂTRE and Claude MARCHÉ: *The BIBTEX2HTML Home Page*. June 2006. `http://www.lri.fr/~filliatr/bibtex2html/`.

[5] Jonathan FINE: "TEX as a Callable Function". In: *EuroTEX 2002*, pp. 26–30. Bachotek, Poland. April 2002.

[6] Michael J. GORDON, Arthur J. MILNER and Christopher P. WADSWORTH: *Edinburgh* LCF. No. 78 in LNCS. Springer-Verlag. 1979.

[7] Paul GRAHAM: ANSI COMMON LISP. Series in Artificial Intelligence. Prentice Hall, Englewood Cliffs, New Jersey. 1996.

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

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

[10] Jean-Michel HUFFLEN : *Introduction au λ-calcul (version révisée et étendue)*. Polycopié. Besançon. Février 1998.

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

[12] Jean-Michel HUFFLEN: "Managing Languages within MlBIBTEX". TUG*boat*, Vol. 30,

no. 1, pp. 49–57. July 2009.

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

[14] *Java Technology*. March 2008. `http://java.sun.com`.

[15] 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.

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

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

[18] Matthias KÖPPE: *A BIBTEX System in Common Lisp*. January 2003. `http://www.nongnu.org/cl-bibtex`.

[19] Thomas LACHAND-ROBERT : *La maîtrise de TEX et LATEX*. Masson. 1995.

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

[21] Xavier LEROY, Damien DOLIGEZ, Jacques GARRIGUE, Didier RÉMY and Jéróme VOUILLON: *The Objective Caml System. Release 0.9. Documentation and User's Manual*. 2004. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

[22] 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/software/emacs/elisp-manual/`.

[23] John MCCARTHY: "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". *Communications of the ACM*, Vol. 3, no. 4, pp. 184–195. April 1960.

[24] 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.

[25] Chuck MUSCIANO and Bill KENNEDY: HTML & XHTML*: The Definitive Guide*. 5th edition. O'Reilly & Associates, Inc. August 2002.

[26] Oren PATASHNIK: *BIBTEXing*. February 1988. Part of the BIBTEX distribution.

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

[28] Simon PEYTON JONES, ed.: *Haskell 98 Language and Libraries. The Revised Report*. Cambridge University Press. April 2003.

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

[30] Denis B. ROEGEL : « Anatomie d'une macro ». *Cahiers GUTenberg*, Vol. 31, p. 19–27. Décembre 1998.

[31] Michael SPERBER, William CLINGER, R. Kent DYBVIG, Matthew FLATT, Anton VAN STRAATEN, Richard KELSEY, Jonathan REES, Robert Bruce FINDLER and Jacob MATTHEWS: *Revised^6 Report on the Algorithmic Language Scheme*. September 2007. `hhtp://www.r6rs.org`.

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

[33] Guy Lewis STEELE, JR., with Scott E. FAHLMAN, Richard P. GABRIEL, David A. MOON, Daniel L. WEINREB, Daniel Gureasko BOBROW, Linda G. DEMICHIEL, Sonya E. KEENE, Gregor KICZALES, Crispin PERDUE, Kent M. PITMAN, Richard WATERS and Jon L WHITE: COMMON LISP. *The Language. Second Edition*. Digital Press. 1990.

[34] "TEX Beauties and Oddities. A Permanent Call for TEX Pearls". In *TEX: at a turning point, or at the crossroads? BachoTEX 2009*, pp. 59–65. April 2009.

[35] Simon THOMPSON and Steve HILL: "Functional Programming through the Curriculum". In: *FPLE '95*, pp. 85–102. Nijmegen, The Netherlands. December 1995.

[36] W3C: XSL *Transformations (*XSLT*). Version 2.0*. W3C Recommendation. Edited by Michael H. Kay. January 2007. `http://www.w3.org/TR/2007/WD-xslt20-20070123`.

[37] Larry WALL, Tom CHRISTIANSEN and Jon ORWANT: *Programming Perl*. 3rd edition. O'Reilly & Associates, Inc. July 2000.

[38] Halina WĄTRÓBSKA i Ryszard KUBIAK: „Od

XML-a do TEX-a, używając Emacsa i Haskella". *Biuletyn* GUST, tom 23, strony 35–39. In *BachoTEX 2006 conference*. kweicień 2006.

[39] Jiři ZLATUŠKA: "ℵĵ§: Programming Languages and Paradigms". In: *EuroTEX 1999*, pp. 241–245. Heidelberg (Germany). September 1999.

Notes

1. The most recent pearls can be found in [34].
2. **A**merican **N**ational **S**tandards **I**nstitute.
3. **H**yper**T**ext **M**arkup **L**anguage, the language of Web pages. [25] is a good introduction to it.
4. **C**ategorical **A**bstract **M**achine **L**anguage.
5. **M**ulti**L**ingual BIBTEX.
6. This language has been named after logician Haskell Brooks Curry (1900–1982).
7. **N**ew **T**ypesetting **S**ystem. It was finally developed using Java [14].
8. **C**OMMON **L**ISP **O**bject **S**ystem.
9. '*Programmation fonctionnelle avancée*', in French. In 2009, it has been renamed into '*Outils pour le développement*' (*Tools for Development*), but changes about the contents are slight.
10. The program of this 2nd academic year unit can be found in French in [8].
11. 'Lisp' stands for '**LIS**t **P**rocessing, because Lisp dialects' major structures are linked lists. Their syntax is common and based on fully-parenthesised prefixed expressions. Lisp's first version, designed by John McCarthy, came out in 1958 [23]. This language has many descendants, the most used nowadays being COMMON LISP and Scheme.
12. 'Dynamically typed' means that we can know the type of an object at run-time. Examples are given in Figs. 1 & 2.
13. 'ML' stands for '**M**eta**L**anguage' and has been initially developed within the formal proof system LCF (**L**ogic for **C**omputable **F**unctions) [6]. Later on, it appears as an actual programming language, usable outside this system, and its standardisation resulted in the Standard ML language.
14. There are several definitions of *strong typing*. The most used within Functional programming is that the variables are typed at compile-time. Some courses being the same level are based on a strongly typed functional programming language, examples are CAML or Haskell. Is that a choice better than Scheme? This is a busy debate... but it is sure that these courses do not emphasise the same notions than a course based on a Lisp dialect.
15. **D**ocument **S**tyle **S**emantics **S**pecification **L**anguage.
16. **S**tandard **G**eneralised **M**arkup **L**anguage. Ancestor of XML (e**X**tensible **M**arkup **L**anguage), it is only of a historical interest now. Readers interested in SGML (resp. XML) can refer to [2] (resp. [29]).
17. e**X**tensible **S**tylesheet **L**anguage **T**ransformations.
18. Nowadays, the call by value is the most commonly used strategy—in particular, in C and in Scheme—the argument expression(s) of a function are evaluated before applying this function. For example, the evaluation of the expression `(factorial (+ 1 9))`—see Fig. 1—begins with evaluating `(+ 9 1)` into `10`, and then `factorial` is applied to `10`. In other strategies, such as *call by name* or *call by need*, argument expressions are evaluated whilst the function is applied.
19. The *extent* of a variable may be viewed as its lifetime: if it is *limited*, the variable disappears as soon as the execution of the block establishing it terminates; if it is *unlimited*, the variable exists as long as reference's possibility remains. In Scheme, variables have unlimited extent.
20. A *variable capture* occurs when another binding than expected is used.
21. On the contrary, Scheme interpreters do not evaluate a `lambda` expression's body. They use a technique—so-called *lexical closure*—allowing the function to retrieve its definition environment.
22. There is another distinction in Scheme, between 'physical' equality (function `eq?`) and 'visual' one (function `equal?`) [31, § 11.5].
23. COMMON LISP allows that about variables and functions, by means of the functions `boundp` and `fboundp` [33, 7.1.1].
24. Macros exist in Scheme: the best way to implement them is the use of *hygienic* macros, working by pattern-matching [31, §§ 11.2.2 & 11.19].
25. As an example of handling several namespaces in COMMON LISP, `let` is used for local variables, whereas local recursive functions are introduced by `labels`, as shown in Fig. 2.
26. For example, there is no hygienic macro in COMMON LISP.
27. **P**ractical **E**xtraction **R**eport **L**anguage.
28. **E**diting **MAC**ros.
29. ... although we consider only centimeters and millimeters in the example given in Fig. 6, for sake of simplicity.
30. In Scheme's last version, a variable can be defined without associated value [31, § 11.2.1]. That was not the case in the version before [15, § 5.2], so such a variable declaration was given a *dummy* value, which enforced the use of side effects.
31. Many data structures, comparable with our type *dimension*, are used within MlBIBTEX's implementation, as briefly sketched in [12]. Another technique, based on *message-passing*, allows us to avoid side effects. Only one function would be defined to manage dimensions, and the three functionalities implemented by `mk-dimension`, `dimension?`, and `dimension->mm` would be implemented by messages sent to the general function, the result being itself a function.

Jean-Michel Hufflen
LIFC (EA CNRS 4157),
University of Franche-Comté, 16, route de Gray,
25030 Besançon Cedex, France