TEX's Language within the History of Programming Languages

Jean-Michel HUFFLEN
LIFC (FRE CNRS 2661)
University of Franche-Comté
16, route de Gray
25030 BESANÇON CEDEX
FRANCE
hufflen@lifc.univ-fcomte.fr
http://lifc.univ-fcomte.fr/~hufflen

Abstract

We connect some representative statements of TEX's language to some analogous features belonging to programming languages, from a point of view related to history. Some features that look strange now are explained easily if we consider the time when TEX came out. By comparing programming in TEX with other paradigms, we also show what TEX can do easily and what is tedious for it.
Keywords programming in TEX, history of programming languages, programming paradigms.

Streszczenie

Łączymy kilka typowych elementów języka TEX z analogicznymi elementami właściwymi dla języków programowania z historycznego punktu widzenia. Niektóre z nich wyglądają dziwnie, ale łatwo je wyjaśnić, biorąc pod uwagę czas ukazania się TEX-a. Porównując programowanie w TEX-u z innymi paradygmatami pokazujemy z czym TEX radzi sobie z łatwością, a co sprawia mu kłopot.
Słowa kluczowe programowanie w TEX-u, historia języków programowania, paradygmaty programowania.

Introduction

TEX is widely known as a wonderful typeset engine. Its typeset process is controlled by means of *commands* — end-users can add their own commands — written using a language with expressive power comparable to programming's. This language allows computation, as well as alternative and loop statements ruled by performing tests that are interpreted as *true* or *false*. This language, fully used in the *plain TEX* format[1], is described in [30]. Aspects in connection with programming are emphasised in some other books introducing users to this format: e.g., [9, 13], the most representative reference from this point of view being [32], as far as we know.

Writing applications using TEX's language proceeds from a particular programming paradigm, as illustrated by the examples of 'pure' programming given in [30, Exercise 20.20] (giving the first prime numbers, counting the non-blank characters of a string). Some features are fascinating, some make difficult the programming of some operations. We could give rough or exhaustive comparison between the statments of TEX and those used within other programming languages. That probably would lead us to a simple conclusion: TEX is very suitable for programming the layout's part of a word processor, tedious elsewhere. We could think so, after reading [40], which develops the example of giving the first prime numbers sketched in [30, Exercise 20.20]. In fact, this style can be quite familiar to people who get used to programming with *macros*, as in languages such that C [29], C++ [46] and some Lisp[2] dialects [45, § 3.3].

Our purpose is different: TEX's first version came out in 1978, some of its features typically belong to the languages of the seventies. We show

[1] To be fit for use, the definitions provided by TEX's core need to be organised in a *format*. The first format, developed by Donald E. Knuth, is *plain TEX* [30]; other formats are LATEX [33], defined by Leslie Lamport, AMS-TEX — the American Mathematical Society's format — defined by Michael D. Spivak citespivak1986, and ConTEXt, defined by Hans Hagen [16].

[2] **LIS**t **P**rocessing. These dialects are the successors of the Lisp language created by John McCarthy [35].

how the difficulties caused by such choices have been managed within TeX, and compare the process of elaborating TeX's language with the evolution of programming languages. In the first section, we take a glance at the programming languages contemporary with TeX's first version. Then we examine some particular points, in particular how TeX's main statements have been built according to a coherent way, even if the approach for designing them was not the same in other programming languages.

We assume that readers are quite familiar with the basic constructs in TeX. However, being a TeX guru is not required. Some examples using other languages are given, but we think that readers should understand them, at least roughly.

A glance at 'old' programming

When TeX was designed, some important programming languages had already appeared, but were not really devoted to dealing with strings, in the sense that an advanced word processor has to do such task. In fact, Donald E. Knuth wanted to write just a typesetting language, and not a programming language as well; he said 'if there were a universal simple interpretive language, naturally I would have latched onto that right away' [31, pp. 648–649].

The oldest programming language, FORTRAN[3] [21] was used for scientific applications using numerical computation. Strings were mainly used to display messages and were called *Hollerith constants* w.r.t. FORTRAN's terminology. The use of strings was improved in FORTRAN IV, issued in 1966 [23], and FORTRAN 77 [2], but was still tedious. In addition, let us remark that at this time, the FORTRAN language did not allow programming without 'goto' statements: FORTRAN 77 did not provide a 'while' statement, that is, a loop statement with an unspecified number of iterations.

COBOL[4] [11] is the second-oldest language and the most used at this time. It is a wordy language — programs using it are verbose — oriented to data processing, especially file processing in batch mode, but had good data typing for the time. Nevertheless, it does not have a rich library of procedures dealing with strings. However, let us remark that some arithmetic operations of this language have syntax close to TeX's:

(COBOL)	(TeX)
`ADD 1 TO X`	`\advance\x by 1`
`MULTIPLY 2 BY Y`	`\multiply\y by 2`
`DIVIDE 3 INTO Z`	`\divide\z by 3`

[3] **FOR**mula **TRAN**slating system.
[4] **CO**mmon **B**usiness **O**riented **L**anguage.

Algol[5] is the first programming language whose syntax looks 'modern' and has been defined rigorously: since this time, the BNF[6] notation [4] has been used to describe grammars of programming languages in a formal way. However, the first version, Algol 60 [37] does not have a string type. This point had been slightly improved in Algol 68 [49], but with a small library of functions dealing with strings. More generally, the successors of Algol, including general languages such as PL/1[7] [22] and Pascal [50] implemented strings by arrays of characters. In addition to the concatenation operator '`+`' provided by Algol 68[8], PL/1 included `INDEX` and `SUBSTR` functions for searching a string and extracting a substring. *Packed arrays* of characters, included in Pascal, allow the values being this type to be written in a compact form. But in fact, the operations programmers can apply to strings are the operations they can apply to (packed) arrays of characters.

We can observe the same point — lack of rich libraries for strings — about most of specialised languages of this time: JOVIAL[9] [42], which is an extension of a preliminary version of Algol 60 including real-time aspects, the first versions of Lisp — the first functional programming language[10] — [45], especially used within Artificial Intelligence area, and another functional programming language designed and implemented early: APL[11] [24], used for numerical computation and well-known because of its very compact syntax.

As far as we know, the only 'early' language aimed at processing strings nicely was SNOBOL[12] [15], in the sense that this language allowed high-level operations such as pattern-matching. As an example, Figure 1 gives a program in SNOBOL 4 displaying the number of words within an input text. As it can be seen, syntax looks old, but strings are viewed as a whole, without reference to a composite structure like an array. Pattern-matching has

[5] **ALGO**rithmic **L**anguage.
[6] **B**ackus-**N**aur **F**orm.
[7] **P**rogramming **L**anguage Number one.
[8] ... denoted by '`||`' in PL/1.
[9] **J**ules' **O**wn **V**ersion of the **I**nternational **A**lgorithmic **L**anguage. 'IAL' was the name of the preliminary version of Algol 60.
[10] Functional programming is a style of programming that emphasises the evaluation of expressions, rather than execution of statements in imperative style, as in languages like Pascal or C. Expressions in a functional programming language — which supports and encourages this style — are formed by using functions to combine basic values. In addition, functions may be used as arguments or results of other functions.
[11] **A** **P**rogramming **L**anguage.
[12] **S**tri**N**g-**O**riented sym**BO**lic **L**anguage.

```
* Such a line is a comment. A word is defined to be a contiguous run of letters, digits, apostrophe or
* hyphen. '=' is for the assignment operator and strings are concatenated simply by writing one after the
* other, as we do for the values of WORD and WORDPATTERN. First, our pattern — the value of
* WORDPATTERN — matches up to but not including the elements of WORD, and then, it matches one or
* more characters in WORD. If &TRIM is set to 1, trailing blanks are removed when a line is input. &UCASE
* (resp. &LCASE) stands for upper-case (resp. lower-case) letters.

          &TRIM          = 1
          WORD           = "'-" '0123456789' &UCASE &LCASE
          WORDPATTERN    = BREAK(WORD) SPAN(WORD)

* Read a line, and go to the statement referenced by DONE if this operation fails (':F'). Then perform the
* pattern-matching operation: if it fails, read the following line. Pattern-matching is expressed by
* 'SUBJECT PATTERN', left to the '=' sign. Replacing the matched string can be expressed, right to the '='
* sign. ':(NEXTLINE)' means for going unconditionally to NEXTLINE.

NEXTLINE  LINE           = INPUT              :F(DONE)
NEXTWORD  LINE WORDPATTERN =                  :F(NEXTLINE)
          N              = N + 1              :(NEXTLINE)

* We rely upon the fact that the system initialises N to an empty string. It is converted to zero when N is
* incremented — see above — or by means of '+N' in next statement:

DONE      OUTPUT         = +N ' words'
END
```

Figure 1: Counting words within a string: program in SNOBOL 4.

nice expressive power, including pattern-matching with replacement: in fact, SNOBOL already developed ideas later put into action in some modern script languages, such as awk[13] [1] and Perl[14] [48]. Last, let us notice how alternatives are programmed in this language: each statement either succeeds or fails. Transfers of control ('goto's') are specified at the end of a statement: it may be unconditional — ':(...)' — or may occur only if the statement has succeeded — ':S(...)' — or failed — ':F(...)'. See some examples in Figure 1.

In comparison to this program in SNOBOL, we put down the same function written in the Scheme programming language [27] — the modern Lisp dialect — in Figure 2. The behaviour is more 'classical': an iteration along the string, by means of the internal function `thru`[15]. Let us notice that the whitespace characters located at the beginning and the end of the string do not count, as we did in the program given in Figure 1.

We do not pretend to be exhaustive about this overview; for example, we did not mention some didactic languages, like BASIC[16] [28], intended to introduce people to programming. As a matter of fact, we do not want to write the history of early programming languages[17]. Our purpose is just to emphasise the fact that often, early programming was not really applied to strings. Besides, the first word processors[18] did not output texts with layout comparable to TEX's. In addition, let us recall that at this time, 'non-conventional' languages — for example, functional languages — were supposed to be inefficient. On another hand, no 'universal' language has really succeeded. Ada® will attempt to become such, but in 1978, when TEX's first version came out, only the final requirements were finalized by the American Department of Defense, and the first version of this language will be issued only in 1983 [3]. All these points probably explain that Donald E. Knuth wanted to design his own language for ruling the operations of typesetting texts.

A small example

Let us begin our analysis of TEX's statements by programming in TEX the example already shown in

[13] **A**ho, **W**einberger, **K**ernighan, the three authors of this language.

[14] **P**ractical **E**xtraction and **R**eport **L**anguage.

[15] Readers interested in an introductory book to Scheme can refer to [43].

[16] **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode.

[17] Readers interested in this topic can refer to [41].

[18] For example, troff, the word processor included in the first distribution of the UNIX® system [8].

```
(define (count-words string-0)
  ;; Here a word is defined as a sequence of non-whitespace characters.
  (let ((string-length-0 (string-length string-0)))
    (let thru ((index 0)
               (skipping-whitespace? #t) ; Are we skipping whitespace characters or
               (nb-of-words 0))          ; non-whitespace ones? '#t' — resp. '#f' — stands for
                                         ; 'true' — resp. 'false'.
      (if (= index string-length-0) ; We have just run out of the string...
          nb-of-words               ; ... and we return how many words have been counted.
          (let ((next-index (+ index 1)))
            ;; nb-of-words is given another value only when we are skipping consecutive whitespace
            ;; characters — except those possibly beginning string-0 — and encounter a
            ;; non-whitespace one.
            (cond ((char-whitespace? (string-ref string-0 index))
                   (thru next-index #t nb-of-words))
                  (skipping-whitespace? (thru next-index #f (+ nb-of-words 1)))
                  (else (thru next-index #f nb-of-words))))))))
```

Figure 2: Counting words within a string: program in Scheme.

Figures 1 and 2: counting words within a string. For sake of simplicity, we assume that this string does not contain active characters or command names. The complete text of our command `\countwords` is given in Figure 3 and if we trace the commands defined over there on a small example, we get:

```
\countwords{TeX is great}                  ⟹
\reachnospace TeX is great\enditer         ⟹
\onnospace eX is great\enditer             ⟹
\next\iteralongword eX is great\enditer    ⟹
\skipword X is great\enditer               ⟹
\next\iteralongword X is great             ⟹
\skipword\ is great\enditer                ⟹
\next\iteralongword\ is great\enditer      ⟹
\skipword is great\enditer                 ⟹
\next\reachnospace is great\enditer        ⟹
\onnospace s great\enditer                 ⟹
...                                        ⟹
\skipword t\enditer                        ⟹
\next\iteralongword t\enditer              ⟹
\skipword\enditer                          ⟹
\next\iteralongword\enditer                ⟹
\enditer\iteralongword                     ⟹
\number\nbofwords                          ⟹
3
```

In other words, whilst we are going through the string, the two commands `\reachnospace` and `\iteralongword` put the first character in the value of the `\next` command. The `\onnospace` command checks the category code of this first character, and as soon as a non-whitespace character is encountered, the `\nbofwords` counter is incremented and we are eating the other letters of the word by means of the commands `\iteralongword` and `\skipword`. If a whitespace character is encountered whilst the `\skipword` command is running, we are calling the `\reachnospace` command, that skips the consecutive whitespace characters possibly put after this whitespace character — that is done by:

`{\afterassignment...\let\next=␣}`

— then we are calling the `\onnospace` command. When we are reaching the end of the string, the process ends up by processing:

- either `\enditer\reachnospace`,
- or `\enditer\iteralongword`,

depending on the character — whitespace or not — terminating the string. This style of programming may look strange, but is very common in TeX for going along a string: similar examples — and more explanations to understand them — are given in [30, Exercise 20.20] and [32, ¶ 426].

As shown by this example, there is no distinction between scanner and parser in TeX, that is, no distinction between lexical and syntactic analysis. There is only one analyser, which returns either a whitespace character, or another character, different from '\', or the complete name of a command. Let us remark that we can explain such a convention easily because of the history: this notion was not obvious for the first programming languages. This distinction does not exist in FORTRAN, either[19]. As

[19] That indirectly caused a spatial engine to be lost: readers interested in this pitfall can refer to [12] for a survey.

```
\newcount\nbofwords
\def\countwords#1{%
 \nbofwords=0
 \def\reachnospace{%
  \afterassignment\onnospace\let\next=}%
 \def\onnospace{\ifcat\noexpand\next A%
   \advance\nbofwords by 1%
   \let\next=\relax%
  \fi%
  \next\iteralongword}%
 \def\iteralongword{%
  \afterassignment\skipword\let\next= }%
 \def\skipword{%
  \ifcat\noexpand\next\space%
   \let\iter=\reachnospace\else%
   \let\iter=\iteralongword%
  \fi%
  \ifcat\noexpand\next\relax\else%
   \let\next=\relax%
  \fi%
  \next\iter}%
 \def\enditer##1{\number\nbofwords}%
 \reachnospace#1\enditer}
```

Figure 3: Counting words within a string: program using *plain TEX*'s macros.

another example, there is no syntactic analysis in PL/1 in the sense of a parser of modern language, because there is no reserved keywords: the compiler must do a first pass only to determine where the 'actual' keywords are, the other occurrences of these words being 'simple' identifiers.

Mixfixed terms in TEX

TEX allows a kind of pattern-matching for the arguments of its commands. For example, the command producing the square root symbol in plain TEX could be defined as:

`\def\root#1of#2{...}`

('`$\root 3 of 2$`' produces '$\sqrt[3]{2}$'). As far as we know, TEX is the first language providing this feature[20]. It is not very well-known among LATEX users: Leslie Lamport recommends them to systematically surround commands' arguments by braces and get rid of the TEX command `\def`. According to his view, it should be replaced by `\newcommand` and `\renewcommand`, provided by the LATEX format [33, § 6.1.4].

[20] There is something equivalent in some Lisp dialects, but rarely used.

```
(let ((n 2005))
  (let ((f (lambda (x) (+ x n))))
    (let ((n 0))
      ;; Hereafter, we use funcall because the
      ;; call of f is computed: f is a variable.
      (funcall f 0)))) ⟹lexically     2005
                       ⟹dynamically   0
```

Figure 4: Lexical and dynamic scope in Lisp.

Such *mixfixed* terms have been used in algebraic specification languages[21]: in OBJ [14], in the PLUSS[22] language, part of the Asspegique[23] toolbox [6]. The description of the CIGALE parser used within this toolbox is given in [47]. This parser for mixfixed terms was adapted for the GLIDER[24] language [20]: see [25] for more details. The CASL[25] language [7] allows mixfixed terms, too:

$$\textit{root} \_\_ \textit{of} \_\_ : \ldots$$

('$\_\_$' is for a placeholder).

Command management in TEX

When identifiers are given values, in any language, an important question is '*how* are these identifiers bound?' By *reference* or by *value*? That is, a semantics based on *share* or *copy*? Both are allowed within TEX, respectively by means of the commands `\def` and `\let`. If we consider:

`\def\newcmd{\oldcmd}`

unless the `\newcmd` command is redefined, it will always behave like `\oldcmd`, even if `\oldcmd` is redefined. On the contrary:

`\let\newcmd=\oldcmd`

reads that the `\newcmd` command is bound to the present value of `\oldcmd`. If it is changed, `\newcmd` will be still bound to the same value. In fact, when a command is to be redefined and the 'new command' depends on the 'old command', the method is:

`\let\cmdsaved=\cmd`

[21] This kind of languages aim to describe the behaviour of software by means of mathematical models, namely *algebras*. Interesting properties — completeness, coherence,... — can be studied formally within this framework.

[22] **P**roposition of a **L**anguage **U**seable for **S**tructured **S**pecifications.

[23] In French, ***AS**sistance à la **SPÉ**cification al**G**ébr**IQUE***, that is, assistance in algebraic specification.

[24] **G**eneral **L**anguage for the **I**ncremental **D**efinition and **E**laboration of **R**equirements.

[25] **C**ommon **A**lgebraic **S**pecification **L**anguage. This language has been designed by an international workgroup aiming to join efforts from different teams, in order to produce a unified algebraic specification language.

```
(defun map-1 (f l)
  (if (null l)
      ()
      (cons (funcall f (car l))
            (map-1 f (cdr l)))))
```

(map-1 (lambda (x) 0) '(1)) $\Longrightarrow_{\text{dynamically}}$ (0)

(map-1 (let ((l '(0)))
(lambda (x) (car l)))
'(1)) $\Longrightarrow_{\text{dynamically}}$ (1)

Figure 5: Variable capture within a dynamically scoped Lisp interpreter.

```
\def\cmd{...\cmdsaved...}
```

that is, the 'old command' is previously saved by means of the `\let` macro.

We think that the `\edef` command is an intermediate form, comparable with a lexical closure within a Lisp dialect. To explain this notion, let us consider the Lisp expression given in Figure 4 and remark that the body of the local function `f` contains the `n` variable. Let us apply this `f` function, if we consider the value of `n`:

- at run-time, the language is *dynamically scoped*;
- at the time of the definition of `f`, the language is *lexically* scoped.

Historically, the first Lisp dialects chose the dynamic scope, because that is easy to put it into action efficiently. But it is incorrect from a mathematical point of view, because it causes *variable captures*. Consider the example given in Figure 5 within a dynamically scoped Lisp interpreter[26]: `map-1` is a function that applies an `f` function to each element of a list, and returns the list of the results of `f`. A simple example is given with a constant function returning zero. But if we reformulate this function by using an `l` variable, this variable is captured by the `l` variable belonging to `map-1`'s definition. To avoid such an error, we could rename the first or second variable `l`, but such an evaluation should not rely upon variables' names.

Dynamic scope is of interest in when some variables can be redefined. For example:

```
(let ((*print-base* 8))
  (write 2005))
```

displays the 2005 number in the octal number system, that is, `3725`. Most often, this `write` function

[26] For example, Emacs Lisp [34], the language of the emacs editor, is dynamically scoped.

(let ((n 2005))
(let ((f #'(lambda (x) (+ x n))))
(let ((n 0))
(funcall f 0)))) $\Longrightarrow$ 2005

Figure 6: Lexical closure within a Lisp dialect.

is used when the `*print-base*` variable defaults to `10`, and we can redefine it for a particular use.

Redefining commands is allowed by TeX: for example, users can redefine the layout of a paragraph by changing its left and right additional margins. This is done by redefining the two commands `\leftskip` and `\rightskip`. TeX is dynamically scoped. But how to avoid variable captures? As far as possible, implementors of Lisp dialects attempt not to evaluate the body of a function unless this function is applied. To avoid variable captures, they put into action *lexical closures*: a function encloses the environment of its definition. Generally[27], a lexical closure is expressed by means of the form '`(function ...)`', abbreviated in '`#'...`', as shown in Figure 6. Naturally, this function form is useless for lexically scoped Lisp dialect, that is why it does not exist in Scheme, lexical closures are implicit for any function expression[28].

TeX's behaviour is to expand commands until there is only 'pure' text. It seems that Donald E. Knuth preferred 'immediate' expansion to lexical closure, so there is a way — the `\edef` macro — to expand the body of a command as soon as this command is defined, as shown in Figure 7. In addition, when TeX came out, lexical closures were heavy to be put into action. However, the `\edef` macro does not replace a lexical closure completely. First, all the commands are expanded, as far as possible, so Donald E. Knuth introduced a `\noexpand` command, in order to delay such an operation if need be[29]. Second, the predefined commands might be redefined inadvertently[30]. TeX is protected against such errors by using the '`@`' character as a constituent of

[27] Most of dynamically scoped Lisp interpreters provide lexical closures, but not all: Emacs Lisp does not. The `function` form exists within this language, but for another purpose.

[28] Common Lisp [44], one of modern Lisp dialects, is lexically scoped, too, and using the `function` form is needed whenever a value is viewed as a function.

[29] This `\noexpand` command is used in Figure refhb-figure-cw-tex to prevent the expansion of the `\next` command. If its value is a macro, it is retained instead of being run.

[30] For example, when we apply our `map-1` function, defined in Figure 5, we can rename our local variable `l` within the expression '`(let ((l ...)) ...)`'... provided that we know

```
\def\kind{lexical}
\edef\firstversion{%
 I'm using \kind\ scope.\par}
\def\secondversion{%
 I'm using \kind\ scope.\par}
\def\kind{dynamic}
```

Figure 7: Lexical and dynamic definitions in TEX.

the names of basic commands, end-users cannot redefine.

Evaluation

As mentioned in the introduction, TEX's commands can be viewed as macros and in such a case, their arguments are not evaluated before applying it, they are just bound to `#1`, `#2`, ... Such a behaviour, close to a *call by name* in 'traditional' programming languages, had been expressed by the `nlambda` constructor within Lisp's first versions:

$$\texttt{((lambda (x) x) (+ 1 2))} \Longrightarrow \texttt{3}$$
$$\texttt{((nlambda (x) x) (+ 1 2))} \Longrightarrow \texttt{(+ 1 2)}$$

From a theoretical point of view, the call by name ensures that the result of an evaluation — the *normal form* of a term, w.r.t. lambda-calculus' terminology [5] — is reached if it exists. But it is inefficient because an argument is evaluated each time it occurs within a function's body, that is why implementors of functional languages generally prefer *calls by value*, that is, arguments are evaluated before applying a function[31].

The `\expandafter` command may be viewed as the way to implement calls by value in TEX, in the sense that:

$$\texttt{\textbackslash expandafter}\ T_0 T_1$$

expands T_1 before applying the T_0 command. In addition, this command may play the role of an evaluation function, like the `eval` function provided in most Lisp dialects. Let us assume that we would like to define some commands as identity functions, except for the '\' character, e.g.:

```
\def\bachoTeX{bachoTeX}
```

Such commands can be defined by a command generator like:

```
\def\cmdid#1{%
 \expandafter\def\csname#1\endcsname{#1}}
\cmdid{bachoTeX}
```

Similarly, other TEX macros can control the evaluation of the tokens put after a command: `\futurelet` (resp. `\afterassignment`, used in Figure 3) peeks (resp. reads) a token at the input flow.

Active characters

An important point within TEX is the notion of *active characters*, that can run commands as soon as they are read. There is a similar notion in most Lisp dialects: readers use a readtable, and special characters of this readtable can be defined as *macro-characters*. For example, parentheses are macro-characters — begin and end of a list — in the standard readtable of Lisp. In Figure 8, we show how to define braces as macro-characters, so the input :

`{the new list}`

is read as `'(the new list)`. We define a COMMON LISP[32] function associated with the left brace character, and the right brace character has the same meaning than a right parenthesis.

Types

From a theoretical point of view, a language is said **strongly typed** if variables are typed[33], (simply) **typed** if each expression belonging to the language is given a type[34]. TEX matches the second case, even if the type information is not given by type expressions but by macros themselves. For example, the `\number` macro, described in [26], only accepts a number as its argument. Likewise, the `\setlength` command of the LATEX format accepts a command and a dimension as its two arguments. As a third example, the commands '`if...`' — e.g., the `\ifcat` command, used in Figure 3 — define *types of conditional expressions*, depending on the way to perform tests. Some are done on-the-fly — e.g., the tests

that it conflicts with the `l` variable belonging to `map-1`'s definition. This means that we should read the source of this definition before using this function.

[31] Some evaluations may loop endlessly, e.g;:

```
((lambda (y) z)
 ((lambda (x) (x x)) (lambda (x) (x x))))
```

whereas a strategy based on calls by name terminates, but since the normal form of a term is unique, the call by value is not really a bad strategy.

[32] Readtables exist in COMMON LISP — we used them when we built the parser of the FP2 (**F**unctional **P**arallel **P**rogramming) language [17] — but they have not been included in Scheme — so MlBIBTEX's parser can use such a feature and reads characters one by one [19] — which is a 'basic' Lisp. Similarly, we can notice that Ω [39], one of TEX's successors, gave up active characters.

[33] This definition includes languages with type inference, as in the SML (**S**tandard **M**eta**L**anguage) language [38]: end-users do not have to put type expressions when a variable is introduced, but the type-checker does this job.

[34] The notion of type does not imply the presence of type errors: if we consider the PL/1 language [22], a type mismatch never raises an error, there is a conversion to a suitable type.

```
;;; In Scheme and COMMON LISP, a character is denoted by '#\⟨character⟩'.
(set-macro-character #\{ #'(lambda (input-s char)
                              (declare (ignore char))
                              ;; Because of this declaration, a compiler cannot complain if it finds
                              ;; out that the char variable does not appear within the function's
                              ;; body. Note that the read-delimited-list function, used below, is
                              ;; predefined in COMMON LISP.
                              (list 'quote (read-delimited-list #\} input-s))))
(set-macro-character #\} (get-macro-character #\)))    ; The closing brace has the same effect
                                                       ; than a closing parenthesis.
```

Figure 8: Defining the behaviour of a macro-character in COMMON LISP.

done by `\if`, `\ifcase`, `\ifcat`, `\ifdim`, `\ifeof`, `\ifhbox`, `\ifhmode`, `\ifinner`, `\ifmmode`, `\ifnum`, `\ifodd`, `\ifvbox`, `\ifvmode`, `\ifvoid`, `\ifx` — some are made by *replacement*: the tests defined by means of the `\newif` command. When such a test occurs, the generated command must be already set to '`\...true`' or '`\...false`'. For example, the `\ifpdf` command, allowing users to dispatch dfferent commands for TEX and pdfLATEX and provided by the ifpdf package, defaults to `\pdffalse` and is defined as follows:

```
\newif\ifpdf
\ifx\pdfoutput\undefined\else%
 \ifx\pdfoutput\relax\else%
  \ifcase\pdfoutput\else\pdftrue\fi%
 \fi
\fi
```

This way to define tests may look strange now, but let us recall that when TEX came out, such expressions were not unified like today. In FORTRAN IV, the result of a test was a transfert of control ('goto'). In SNOBOL — cf. Figure 1 — tests were based on the success or failure of a statement.

Conclusion

Here is the end of our trip around some main commands of TEX. We have not been exhaustive and did not go thoroughly into the commands for typesetting. We have preferred to focus on aspects close to 'pure' programming. That is not heretic since some theoricians proved that a Turing machine can be programmed using TEX [10, 36]. Moreover, we personally confess that we give some examples using TEX within a lecture entitled *Advanced Functional Programming* [18], as part of emphasising the difference between lexical and dynamic scope.

Often our comparisons with other programming languages have referred to Lisp dialects: that may appear subjective, but let us remark that this class oflanguages aims to express programs and data using the same notation[35]. So does TEX since the content of an input file mixes texts to be typeset and commands. In addition, Lisp dialects have been used successfully to experiment new techniques of programming.

TEX is a complete tool from a theoretical point of view as well as a practical one. Its maturity probably results from a long and complex process. Some commands have a complex definition, but that is not a real drawback for most end-users who can directly type their texts and simply use suitable packages. The problem is the future: who will be able to maintain TEX and make it evolve if new requirements appears? So the TEX community has launched some projects aiming to provide a 'better TEX'. Some people are thinking about a new language for a new word processor. We think that our work can help them, by giving some correspondences with features indisputably belonging to programming. In addition, we have mentioned some directions that have gone on with some TEX's features. We also have mentioned what is old in TEX, in comparison to what is done elsewhere now, what is still in good style.

Acknowledgements

Many thanks to Paweł D. Mogielnicki, who has written the Polish translation of the abstract.

References

[1] Alfred V. AHO, Brian W. KERNIGHAN and Peter J. WEINBERGER: *The awk Programming Language*. Addison-Wesley. 1988.

[35] ... what induces many levels of parentheses, and detractors of Lisp say that this acronym means '**L**ost **I**n **S**tupid **P**arentheses'.

[2] ANSI: *Programming Language* FORTRAN. Technical Report X3.9–1978, American National Standard Institute. 1978.

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

[4] John W. BACKUS: "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference". In: *Proc. of the International Conference on Information Processing,* UNESCO, p. 125–132. 1959.

[5] Henryk Pieter BARENDREGT: *The Lambda Calculus. Its Syntax and Semantics*, Vol. 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland. Revised Edition. 1984.

[6] Michel BIDOIT, Francis CAPY and Christine CHOPPY: "The Design and Specification of the Asspegique data base". In: *Proc. on International Symposium on* DISCO, no. 429 in LNCS, p. 205–214. Springer-Verlag. 1990.

[7] Michel BIDOIT, Peter D. MOSSES, Till MOSSAKOWSKI, Donald A. SANNELLA and Andrzej TARLECKI: CASL *User Manual. Introduction to Using the Common Algebraic Specification Language.* No. 2900 in LNCS. Springer. 2004.

[8] Steve R. BOURNE: *The* UNIX® *System.* Addison-Wesley. 1983.

[9] BUJDOSÓ Gyöngyi – FAZEKAS Attila: *TeX kerdőlépések.* Tertia Kiadó, Budapest. április 1997.

[10] Jean-Côme CHARPENTIER : *turing (v. 0.4). Une simulation des machines de Turing avec TeX.* Février 2003. `http://melusine.eu.org/syracuse/texpng/turing/doc_turing.pdf`.

[11] DoD: COBOL*, Initial Specifications for a Common Business Oriented Language.* Technical Report #1960 0–552133., Department of Defense, Government Printing Office. 1960.

[12] Jim DUNCAN: "FORTRAN Pitfalls". *The Risks Digest*, Vol. 5, no. 66. `http://catless.ncl.ac.uk/Risks/5.66.html`. November 1987.

[13] Victor EIJKHOUT: *TeX by Topic. A TeXnician's Reference.* Addison-Wesley Publishing Company. 1992.

[14] Joseph A. GOGUEN and Timothy WINKLER: *Introducing* OBJ*3.* Technical Report SRI-CSL-88-9, SRI. Projects 1243, 2316 and 4415. August 1988.

[15] Ralph E. GRISWOLD, J. F. POAGE and Ivan P. POLONSKY: *The* SNOBOL *4 Programming Language.* 2nd edition. Prentice Hall, Englewood Cliffs, New Jersey. 1971.

[16] Hans HAGEN: *ConTeXt, the Manual.* November 2001. `http://www.pragma-ade.com`.

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

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

[19] Jean-Michel HUFFLEN: *MlBIBTeX in Scheme.* To appear in Proc. BachoTeX conference. April 2005.

[20] Jean-Michel HUFFLEN and Nicole LÉVY: *The Algebraic Specification Language* GLIDER*: a Two-Level Language.* Presented at the 10th ADT Workshop and 6th General COMPASS Meeting, Santa Margherita (Italy). June 1994.

[21] IBM—APPLIED SCIENCE DIVISION: *Preliminary Report Specifications for the* IBM *Mathematical FORmula TRANSlating system* FORTRAN. November 1954. Programming Research Group.

[22] IBM SYSTEM 360: PL*/1 Reference Manual.* March 1968.

[23] IBM SYSTEM 360: FORTRAN IV *Language Reference.* Technical Report GC 28-6515-9, Programming Research Group. January 1971.

[24] Kenneth IVERSON: *A Programming Language.* Wiley, ed. 1962.

[25] Jean-Pierre JACQUOT and Agnès VALDENAIRE: "Trading Legibility against Implementability in Requirement Specifications: an Experimental Assessment". In: *Proc.* RE *'95*, p. 181–189. IEEE Computer Society. March 1995.

[26] David KASTRUP: "*De Ore Leoni*. Macro Expansion for *Virtuosi*". In: *EuroTeX 2002*, p. 36–45. Bachotek, Poland. April 2002.

[27] 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: *Revised⁵ Report on the Algorithmic Language Scheme.* February 1998. `http://www.cs.indiana.edu/scheme-repository/`.

[28] John KEMENY and Thomas KURTZ: BASIC. October 1964. Darthmouth College, Computation Center.

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

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

[31] Donald Ervin KNUTH: *Digital Typography*. No. 78 in Lecture Notes. Center for the Study of Language of Information. 1999.

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

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

[34] 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`.

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

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

[37] Peter NAUR, ed.: "Report on the Algorithmic Language Algol 60". *Communications of the ACM*, Vol. 3, no. 5, p. 299–314. May 1960.

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

[39] John PLAICE and Yannis HARALAMBOUS: *Draft Documentation for the Ω System*. March 1998. `http://www.loria.fr/services/tex/english/moteurs.html`.

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

[41] Jean E. SAMMET: *Programming Languages: History and Fundamentals*. Prentice-Hall, Englewood Cliffs, New Jersey. 1969.

[42] Jules I. SCHWARTZ: "The Development of JOVIAL". ACM SIGPLAN *Notices*, Vol. 13, no. 8, p. 203–214. June 1978.

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

[44] 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.

[45] Guy Lewis STEELE, JR. and Richard P. GABRIEL: "The Evolution of Lisp". ACM SIGPLAN *Notices*, Vol. 28, no. 3, p. 231–270. In HOPL-II Conference. March 1993.

[46] Bjarne STROUSTRUP: *The C++ Programming Language*. 2nd edition. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts. 1991.

[47] Frédéric VOISIN: "Cigale: a Tool for Interactive Grammar Construction and Expression Parsing". *Science of Computer Programming*, Vol. 7, no. 1, p. 61–86. 1986.

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

[49] Adriaan VAN WIJNGAARDEN, Barry James MAILLOUX, John E. L. PECK, Cornelis H. A. KOSTER, Michel SINTZOFF, Charles LINDSEY, L. G. T. MEERTENS and R. G. FISKER: *Revised Report on the Algorithmic Language Algol 68*. Springer-Verlag. 1976.

[50] Niklaus WIRTH: "The Programming Language Pascal". *Acta Informatica*, Vol. 1, no. 1, p. 35–63. 1971.