Star TEX: The Next Generation

Didier Verna

Abstract

While TEX is unanimously praised for its typesetting capabilities, it is also regularly blamed for its poor programmatic offerings. A macro-expansion system is indeed far from the best choice in terms of general-purpose programming. Several solutions have been proposed to modernize TEX on the programming side. All of them currently involve a heterogeneous approach in which TEX is mixed with a full-blown programming language. This paper advocates another, homogeneous approach in which TEX is first rewritten in a modern language, Common Lisp, which serves both at the core of the program and at the scripting level. All programmatic macros of TEX are hence rendered obsolete, as the underlying language itself can be used for user-level programming.

Prologue

TEX
The `[final]` *frontier.*
These are the voyages,
Of a software enterprise.
Its continuing mission:
To explore new tokens,
To seek out a new life,
New forms of implementation...

1 Introduction

In 2010, I asked Donald Knuth why he chose to design and implement TEX as a macro-expansion system rather than as a full-blown, procedure-based, programming language. His answer was twofold:

1. He wanted something relatively simple for his secretary who was not a computer scientist.
2. The very limited computing resources at that time practically mandated the use of something much lighter than a complete programming language.

The first part of the answer left me with a slight feeling of skepticism. It remains to be seen that TEX is simple to use. Although it probably is for simple things, its programmatic capabilities are notoriously tricky. The second part of the answer, on the other hand, was both very convincing and arguably now obsolete as well. Time has passed and the situation today is very different from what it was 30 years ago. The available computing power has grown exponentially, and so have our overall skills in language design and implementation.

Several ideas on how to modernize TEX already exist. Some of them have actually been implemented. In this paper, we present ours. The possible future that we would like to see happening for TEX is somewhat different from the current direction(s) TEX's evolution has been following. In our view, modernizing TEX can start with grounding it in an old yet very modern programming language: Common Lisp. Section 2 clarifies what our notion of a "better", or "more modern" TEX is. Section 3 on the next page presents several approaches sharing a similar goal. Section 4 on the following page justifies the choice of Common Lisp. Finally, section 5 on page 205 outlines the projected final shape of the project.

2 A better TEX

The TEX community has this particularity of being a mix of two populations: people coming from the world of typography, who love it for its beautiful typesetting, and people coming from the world of computer science, who appreciate it for its automation and similarity to a programming language. It is true that the way TEX works is much closer to that of a compiled language than a WYSIWYG editor.

In both worlds, TEX is unanimously acclaimed for the quality of its typesetting. This shouldn't be surprising, as it has always been TEX's primary objective. The question of its programmatic capabilities, however, is much more arguable. People unfamiliar with programming in general easily acknowledge that TEX's programmatic interface is not trivial to use. For people coming from the world of computer science, it is even more obvious that TEX is no match for a real programming language, partly due to its macro-expansion nature.

Let us recall that TEX was not originally meant to be a programming language. To quote its author, Donald Knuth [8]:

> I'm not going to design a programming language; I want to have just a typesetting language. [...] In some sense I put in many of TEX's programming features only after kicking and screaming.

From this perspective, it seems natural to consider that a "better" TEX, today, would essentially deliver the same typesetting quality from a more modern programmatic interface. More precisely:

- access to the typesetting subset of TEX's primitives should be provided with a consistent syntax (in particular, no more need for `\relax`),
- the programmatic API (`\def`, `\if`, *etc.*) should be dropped in favor of support from a real programming language,

- the system should remain simple to use, at least for simple things, just as TEX is today,
- the system should remain highly extensible and customizable, just as TEX is today,
- and finally, preserving backward compatibility, although not considered mandatory, should also be considered.

3 Alternatives

The idea of grounding TEX into a real programming language is not new. Let us mention some such attempts that we are aware of. `eval4tex`[1] and sTEXme[2] both use Scheme (another dialect of Lisp). PerlTEX [13, 15], as its name suggests, chooses Perl. QaTEX/PyTEX [4] use Python, and finally, also as its name suggests, LuaTEX[3] employs Lua.

These approaches, although motivated more or less by the same general idea, work in very different ways. Some of them wrap TEX in a programming language by giving the language access to TEX's internals. Others wrap a programming language in TEX by allowing TEX to execute code (LuaTEX belongs to this category). Some do both. For instance, sTEXme ships with a extended Scheme engine that can access TEX's internals, as well as a modified TEX engine that can evaluate Scheme code.

Some of them allow authors to write TEX macros in a different language (PerlTEX lets you write TEX macros in Perl). Others, like QaTEX, aim at completely getting rid of TEX macros so that all programmatic functionality is written in another language (Python, with PyTEX in that case). This particular case is closer to what we have in mind.

Finally, some approaches use a synchronous dual-process scheme in which both TEX and the programming language of choice run in parallel, communicating either via standard input/output redirection, or by file or socket I/O. That is the case of sTEXme and PerlTEX. Others, like `eval4tex` use a multi-pass scheme instead. In a first pass, the “foreign” code is extracted and sent to the programming language. The programming language in question executes its code and sends it back to TEX. In the final pass, TEX is left with only regular TEX macros to process.

In spite of all these variations, it is worth stressing that all these approaches have something in common: they are *heterogeneous*. They involve both a programming language engine on one hand, and the original, though possibly modified, TEX engine on the other hand. Even LuaTEX which is somehow more integrated than the other alternatives is built like this: a Lua interpreter is embedded in a more or less regular TEX, written in WEB and C.

[1] `http://www.ccs.neu.edu/home/dorai/eval4tex/`
[2] `http://stexme.sourceforge.net/`
[3] `http://www.luatex.org`

The idea that we suggest in this paper is that another, fully integrated approach is also possible. In this approach, another programming language would be used to completely rewrite TEX *and* provide the desired programmatic layer at the same time. We are aware of at least one previous attempt at a fully integrated approach. NTS[4], the “New Typesetting System” was supposed to be a complete re-implementation of TEX in Java, but the project was never widely adopted. We don't think that this project's demise indicates in any way that the approach is doomed in general. On the contrary, the remaining sections explain why we think that Common Lisp is a very good candidate for it.

4 Common Lisp: why?

Lisp is a very old language. It was invented by John McCarthy in the late 1950s [9]. Contrary to what many people seem to think however, being old doesn't imply being obsolete. In this case, it is a synonym for being mature and modern. When Lisp was invented, it was in fact way ahead of its time, to the point that even its inventor hadn't realized the extent of his creation. A sign of this is the recent incorporation of features that Lisp already provided 50 years ago into so-called “modern” programming languages, such as C# with the addition of dynamic types, or C++ and Java with the addition of lambda (anonymous) functions.

4.1 Standardization

Common Lisp, in particular, was standardized in 1994 [1] (it was in fact the first object-oriented language to get an ANSI standard). Remember how stability was important to Donald Knuth for TEX? This means that even across different implementations (there are half a dozen or so), that the core of the language will *never* change, and in fact hasn't for the last 20 years. Compare this to modern scripting languages such as Python or Ruby, for which the “standard” is essentially *the* current implementation of *the* current version of *the* language written by *the* author of *the* language...

The Common Lisp standard is fairly comprehensive: it includes not only the language core but a large library of functions. Of course, because the standard is 20 years old, it lacks several things that are considered important today (such as a multi-threading API). Every Common Lisp implementation provides its own version of non-standard features, but again,

[4] `http://nts.tug.org`

there are only half a dozen out there, and if one chooses to stick to only one of them, then one gets the same stability as for standardized features, or at least backward-compatibility.

4.2 General purpose *vs.* scripting

Common Lisp also has this particularity of being both a full-blown, general purpose, industrial scale programming language *and* a scripting or extension language at the same time. This is something that cannot be said of most modern languages out there but is nevertheless crucial in the fully integrated approach that we are advocating. This was certainly not the case of Java, in the now dead NTS project.

Common Lisp is indeed a full-blown, general purpose, industrial scale programming language. It is multi-paradigm (functional, object-oriented, imperative, *etc.*), highly extensible (both at the syntactic and semantic levels [12]), highly optimizable (notably with static typing facilities [19, 20]) and has a plethora of libraries (such as Perl-compatible regular expressions, database access, web infrastructure, foreign function interfaces, *etc.*). Today, millions of lines of Common Lisp code are used in industrial applications all over the world.

But Common Lisp is also a scripting language. It is highly interactive (it comes with a REPL: a Read Eval Print Loop), highly dynamic (with features ranging from dynamic type checking to an embedded JIT-compiler and debugger), highly reflexive (something crucial for extensibility and customizability) and at the same time easy to learn (notably out of its minimalist syntax).

This last point constitutes the beginning of our tour of the language. Remember that one of our objectives is to provide a system just as simple as TeX, at least for simple things. Below is a one minute crash course on Lisp syntax.

Literals Common Lisp provides literals such as numbers (`1.254`) or strings (`"foobar"`). Literals evaluate to themselves.

Symbols Common Lisp provides symbols that can be used to name functions or variables (possibly at the same time). `pi` has the expected mathematical value. `identity` is the identity function.

S-Expressions Compound expressions are written inside parentheses and denote function calls in prefix notation. `(+ 1 2)` represents the sum of 1 and 2.

Quotation In Common Lisp, everything has a value. If you want to prevent evaluation, put a quote in front of the expression. For instance, `'identity` is the symbol `identity` itself. `'(+ 1 2)`, instead of being a function call, now represents the list of 3 elements: the symbol `+` and the numbers 1 and 2.

Definitions To define a global variable, we can write something like this:

```
(defvar devil-number 666)
```

To define a function:

```
(defun dbl (x) (* 2 x))
```

And that is the end of our Common Lisp crash course. With that knowledge, you know practically all there is to know about the language in order to use it for simple things. The rest is a matter of knowing the names of standard (built-in) variables and functions. In particular, this is certainly not more complicated than learning the basic and inconsistent TeX syntax (do you provide arguments in braces or inline with a final `\relax` to be on the safe side?), and it would also certainly be enough to write basic LaTeX documents like this:

```
(document-class article)
...
(begin document)
(section "Title")
...
(end document)
```

4.3 Built-in paradigms

In a somewhat surprising way, Common Lisp provides programming paradigms, idioms or library-based features that (LA)TeX also provide (or would like to provide). This means that such features are already here for us and would not need to be re-implemented in a Lispy TeX. This section only presents some of them; there are in fact many more.

4.3.1 `key=value` pairs

The success of key/value arguments in LaTeX is proportional to their actual need: there are at least a dozen packages providing this functionality, each and every one of them with its own pros and cons. Common Lisp provides a clean and straightforward implementation of this for free in its function call protocol (the so-called "lambda lists"). The following function takes one mandatory (positional) argument and two optional *keyword* arguments, which are in fact named, floating arguments:

```
(defun include-graphics
  (file &key width height)
  ...)
```

It can be called with just the mandatory argument:

```
(include-graphics "image")
```

or with either or both keywords (prefixed with a '`:`'):

```
(include-graphics "image" :height <value>)
```

Keyword arguments can have default values which themselves may be dynamically computed.

4.3.2 Packages

LaTeX implements the notion of *packages*, essentially a collection of macros stored in a file. The *de-facto* standard for this in Common Lisp is called ASDF[5] (Another System Definition Facility). Common Lisp *systems* resemble LaTeX packages, only much more evolved. For instance, systems are composed of a hierarchy of files with customizable loading order, automatic dependency tracking and recompilation of obsolete object files (compare this to having only interpreted macros) and much more.

4.3.3 Namespaces

A frequent rant about LaTeX is the lack of namespaces. Common Lisp provides a related concept called *packages* (not to be confused with LaTeX packages). A package is a collection of symbols naming functions, variables, or both. Packages have names through which you access their symbols. Packages may declare some symbols as *public* while the others remain *private*. Suppose for instance that there is a package named `ltx`, that implements LaTeX 2_ε, and declares its function `document-class` as public. From the outside, one would canonically refer to this function as `ltx:document-class`. On the other hand, the need to reference all such LaTeX symbols explicitly in a document would probably be cumbersome. In such a situation, one may use `use-package`, in which case all public symbols become directly accessible. Using the `ltx` package makes it possible to call the function `document-class` implicitly, without the package name prefix.

4.3.4 Interactivity

As you likely know, TeX can be used in an interactive fashion. The following example shows a sample conversation between TeX and a user who mistypes a macro name:

```
didier(s003)% tex
This is TeX, Version 3.1415926 [...]
**\relax

*\hule
! Undefined control sequence.
<*> \hule

? H
The control sequence at the end of the top
line of your error message was never \def'ed.
If you have misspelled it (e.g., `\hobx'),
type `I' and the correct spelling
(e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

? I\hrule

*\bye
```

This kind of interaction pretty much resembles a REPL which Common Lisp, like every interactive language, provides out of the box. Where Common Lisp specifically comes into play is that every Common Lisp application may embed a interactive debugger for free, precisely useful for this kind of error/recovery interaction. In Common Lisp, the programmer has the ability to implement his own recovery options (known as *restarts* in Lisp jargon) without unwinding the stack. This is different and much more powerful than regular catch/throw facilities. If you are not interested in using the full-blown debugger, you can implement a function for catching errors and listing the available recovery options in a bare 10 lines of code.

4.3.5 Dumping

Out of the historical concern for performance, TeX has the ability to dump and reload so-called "format" files, which saves a lot of parsing and interpretation (*cf.* the `\dump` command). Given the increase in computing power over the last 30 years, the question of performance is admittedly much less critical than it used to. Even today, though, performance is not a concern that should be completely disregarded. For example, compiling a lengthy Beamer presentation with lots of animations can still be annoyingly slow.

Common Lisp happens to provide a dumping feature out of the box. Although not part of the ANSI standard, all Lisp compilers provide it. In SBCL[6] for instance, the function `save-lisp-and-die` dumps the whole global state (stack excepted) of the current Lisp environment into a file that can later be quickly reloaded with the `--core` command-line option. Instead of dumping a core image, it is also possible to dump a fully functional standalone executable.

Because the dumping mechanism is accessible to the end-user, interesting applications could be envisioned with very little programming, such as mid-document dumping. By outputting to in-memory strings instead of files, for example, a document author could be given the possibility to dump in the middle of a large document's processing. The resulting facility would be quite similar to `\includeonly` — except that the whole document would be typeset every time.

[5] `http://www.common-lisp.net/project/asdf`

[6] `http://www.sbcl.org`

4.3.6 Performance

Let us tackle the problem of performance from a more general point of view. One frequent yet misinformed argument against dynamic languages is: "they are slow". From that point of view, it may seem odd to even begin to envision the reimplementation of a program such as TEX in a dynamic language.

One first and frequent misconception about interactive languages is that as soon as they provide a REPL, they must be interpreted. This is in fact not the case. Nowadays, many Common Lisp implementations such as SBCL don't even provide an interpreter. Instead, the REPL has a JIT (Just In Time) compiler which compiles the expressions you type and only then executes them. To put this in perspective, compare the processes of interpreting TEX macros by expansion and executing Lisp functions compiled to machine code...

Yet, starting with the assumption that performance should indeed be a concern (this is not even necessarily the case), the argument of slowness may still make some sense in specific cases. For example, it is obvious that performing type checking at run-time instead of at compile-time will induce a cost in execution speed. In general however, this argument, as-is, is meaningless. For starters, let us not confuse "dynamic language" with "dynamically typed language". A dynamic language gives you a lot of run-time expressive power, but that doesn't necessarily mean that you have to use all of it, or that it is impossible to optimize things away.

Let us continue on type checking in order to illustrate this. Look again at the definition for our "double" function:

```
(defun dbl (x) (* 2 x))
```

This function will no doubt be relatively slow, because Common Lisp has a lot of things to do at run-time. For starters, it needs to check that `x` is indeed a number. Next, the multiplication needs to be polymorphic because you don't double an integer the same way you double a float or a complex. That is in fact not the whole story, but we will stop here.

On the other hand, consider now the following version:

```
(defun dbl (x)
  (declare (optimize (speed 3) (safety 0))
           (type fixnum x)
  (the fixnum (* 2 x)))
```

In this function, we request that the compiler optimizes for speed instead of safety. The result is that the compiler will "trust" us and bypass all dynamic checks. Next, we actually provide static type information. `x` is declared to be a `fixnum` (roughly equivalent to integers in other languages) and so is the result of the multiplication. This is important because there is no guarantee that the double of an integer remains the same-size integer. Consequently, in general, Lisp would need to allocate a bignum to store the result.

As it turns out, compiling this new version of the function leads to only 5 lines of machine code. The compiler is even clever enough to avoid using the integer multiplication operator, but a left shift instruction instead. What we get in this context is in fact the behavior of a statically and weakly typed language such as C. Consequently, it should not be surprising that the level of performance we get out of this is comparable to that of equivalent C code. Recent experimental studies have demonstrated that this is indeed the case [19, 20].

This particular example is also a nice illustration of what we meant earlier by saying that Common Lisp is both a full-blown, industrial scale, general purpose programming language, *and* a scripting language at the same time. When working at the scripting level, the first version of `dbl` is quick and good enough. When working in the core of your application however, you appreciate it when the language provides a lot of tools (optimization ones notably) to adjust your code to your specific needs.

4.4 Extensibility and customizability

Another aspect of the language well worth its own section is its level of extensibility (adding new behavior) and customizability (modifying existing behavior). We know how important this is in the (LA)TEX world, which is a complicated ball of intermixed threads all interacting with each other [21], which wouldn't be possible without the level of intercession that TEX macros offer. Similarly, at least part of the success of LuaTEX is due to its ability to provide access to TEX's internals, so it seems that there is also a lot of interest in this area.

4.4.1 Homoiconicity and reflection

Once again, Common Lisp is here to help. We mentioned earlier how the root of extensibility and customizability in Common Lisp is its highly reflexive nature. Reflection is usually decoupled into *introspection* (the ability to examine yourself) and *intercession* (the ability to modify yourself).

In Lisp, reflection is supported in the most direct and simple way one could think of. Remember the expression `(+ 1 2)`, with or without evaluation? As we said before, this expression can either represent a call to the function "sum" with the arguments 1 and 2, or the list of three elements: the symbol `+` and the numbers 1 and 2. What this really means is that every piece of code, if not evaluated, can be seen as a piece of data, and hence can be manipulated at will. In fact, every piece of Lisp code is represented as a list, which happens to be a user-level data structure. This property of a programming language is known as *homoiconicity* [5, 11].

Another important distinction in this notion is structural *vs.* behavioral reflection [10, 17]. While structural reflection deals with providing a way to reify a program, behavioral reflection deals with accessing the language itself. Lisp is one of the very few languages to provide both kinds of reflection to some extent, as we'll now discuss.

4.4.2 Structural reflexivity

This section gives only a couple of examples of structural reflexivity, again, to demonstrate how some well-known TEX idioms map to Common Lisp in a straightforward way.

The functional nature of Lisp implies that functions are first-class citizens in the language [3, 18]. In general, this means that functions can be used like any other object in the language. In particular, this means that functions can be modified, stored in variables, *etc.*

Storing a functional value in a variable will be useful in order to implement a variant of the function which needs to call the original one at some point. This is equivalent to the following common TEX idiom:

```
\let\oldfoo\foo
\def\foo{... \oldfoo ...}
```

Defining a function several times is simply equivalent to overriding the previous definition(s). This behavior matches that of `\def` or (more or less) `\renewcommand`. It is in fact more powerful for at least two reasons:

1. Since Common Lisp has a proper notion of scope (and in fact provides both dynamic and lexical scoping; something that very few other languages, can do), function or variable redefinition can be performed at different scoping levels, not only local or global.

2. Because of its interactive nature, there is no real distinction between functions defined in a core image or executable and those defined in the REPL and that consequently, they can be redefined in the exact same way. Consider what this really means for a minute: one could redefine *any* function in the TEX program just as easily as any TEX macro...

Finally, reflexivity in Common Lisp goes as far as allowing both introspection and intercession at the level of package internals. In most other languages, it is simply not possible to access the so-called "private" parts of a namespace, class, or whatever encapsulation scheme is supported. In Common Lisp, remember that a public symbol is accessed by prefixing its name with the name of the package and a colon separator, for example: `ltx:document-class`.

It turns out that it is just as easy to both introspect and intercede a package's internals. One just needs to use a double colon instead of a single one. This is not unlike the `@` character convention used by LATEX, along with the macros `\makeatletter` and `\makeatother`. The double colon really is a warning that you are prying on private property, but nothing technically prevents you from doing so.

4.4.3 Behavioral reflexivity

Lisp goes even further by providing some level of behavioral reflection as well.

Lisp macros (functions executed at compile-time) provide a form of intercession at the compiler level, allowing one to program language modifications in the language itself (what [16] calls a "homogeneous meta-programming system", as opposed to C++ templates for instance, which are heterogeneous: a different language).

CLOS [2, 6], the Common Lisp Object System is written *in itself*, on top of a so-called Meta-Object Protocol, known as the MOP [14, 7]. Using the CLOS MOP permits intercession at the object system level, allowing the programmer to modify the semantics of the object-oriented layer.

Finally, it is also possible to extend the Lisp syntax, which is a form of intercession at the parser level, allowing to modify the language's syntax directly. This is the only concrete example that we will provide in this section, although a striking one. We assume that the reader is familiar with TEX's double superscript syntax, allowing to denote characters that are not normally printable.

Suppose that we are given a function called `^^-reader` which performs TEX's double-superscript syntax to character conversion (this function is 10 lines long). The following code effectively installs the corresponding syntax in the Common Lisp reader:

```
(make-dispatch-macro-character #\^)
(set-dispatch-macro-character #\^ #\^
  #'|^^-reader|)
```

The first line informs the Lisp reader that the `^` character is to be treated in a special way (do you see a relationship to active characters and catcodes?). The second line informs the Lisp reader that if two such characters are encountered in a row, then, the regular parser should stop and pass control to our user-provided function. We can now verify that this extended syntax works:

```
CL-USER> ^^M
#\Return
CL-USER> ^^00
#\Nul
```

This particular example is a bit simplified, but it conveys the idea. By modifying the way the Lisp parser behaves, we are able to modify the language itself, not only the program we are executing. And again, we are not very far from TEX's notion of active characters.

5 Common Lisp: how?

Section 2 on page 199 listed five objectives for a modern reimplementation of TEX. Section 4 on page 200 demonstrated how Common Lisp can help fulfill three of these goals: providing real programming capabilities, extensibility and customizability, all of this while maintaining a relative ease of use.

This section is devoted to the last two objectives, namely providing a more modern and consistent API while at the same time (although not mandatory) maintaining backward compatibility. In actuality, this section provides a more concrete view of the project itself. Although very little has been implemented already, the project *does* have a name: TiCL (the acronym for "TEX in Common Lisp").

5.1 API

Mapping the typesetting TEX primitives (that is, the non-programmatic ones) to Common Lisp would be rather straightforward.

- TEX parameters become Lisp variables. For instance, `\badness` is represented by a Lisp (global, dynamically scoped) variable `badness`.
- TEX quantities become Lisp objects. The term "object" is to be taken in a broad sense, that is, depending on the exact requirements, either an object of some class from the object system, of some structure, or anything else. In Lisp, it is customary to provide abstract constructor functions following a specific naming scheme. For instance, creating a TEX glue item could be done with a call to a function such as `make-glue`:

  ```
  (defun make-glue (b &key plus minus)
    ...)
  ```

  Since functions like this are bound to be used quite often, a syntactic shortcut may come in handy, such as the rather idiomatic one following, which also demonstrates the Lisp way to set some variable to a specific value (but see section 5.1.1):

  ```
  (setf baselineskip
        #g(b :plus x :minus y))
  ```

- Obviously, every TEX primitive command becomes a Lisp function. Again, the point here is to both simplify the syntax and make it more consistent at the same time (no more `\relax`!). Here are a couple of examples:

  ```
  (input file)
  (hbox material)
  (hbox material :to dim)
  (hbox material :spread dim)
  ```

  The reader familiar with TEX will notice immediately that the arguments in the last two examples are in reverse order, compared to the regular TEX versions. If this is really too much to get accustomed to, variants are easy to implement:

  ```
  (hbox-to dim material)
  (hbox-spread dim material)
  ```

  Such syntactic variants are usually implemented with Lisp macros, evaluated at compile-time, so that there is no additional run-time cost.

5.1.1 Lisp-2

Let us go back to the `baselineskip` assignment example for a minute:

```
(setf baselineskip <glue>)
```

This assignment may seem odd to a TEXnician, who is more accustomed to direct assignments such as

```
\baselineskip 10pt
```

In fact, TEX has this way of using the same macro for both denoting its value and setting it.

We intentionally omitted one point in section 4.3 on page 201 in order to put it here: the fact that Common Lisp is a "Lisp-2" (as opposed to Scheme for instance, which is a Lisp-1). What this means is that Common Lisp has 2 different namespaces for functions and variables. In other words, the same symbol can be used to both refer to a function and a variable *at the same time*.

An interesting consequence of this is that it is possible to define a *function* `baselineskip` the purpose of which is to assign a value to the eponymous *variable* `baselineskip`. Assuming this function exists, the above Lisp expression can hence be simplified as follows:

```
(baselineskip <glue>) ; set it!
```

Again, this aspect of Common Lisp brings us even closer to one of TEX's idioms: that of quantity assignment.

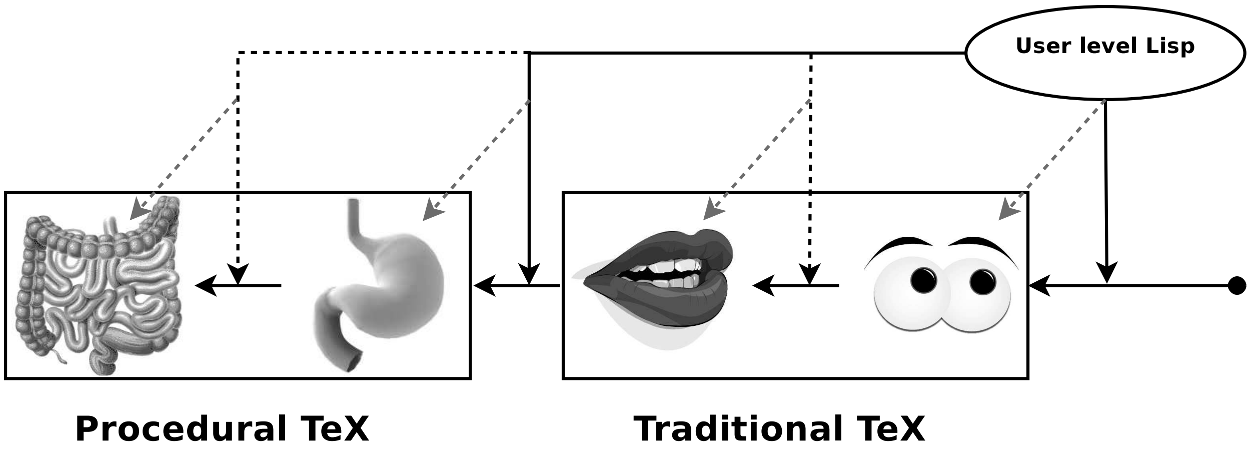

Figure 1: TiCL architecture

5.2 Backward compatibility

The programmatic interface to the typesetting part of TEX described in the previous section is what we call "procedural TEX". Along with the actual typesetting engine, this mostly corresponds to TEX's stomach and bowels. On top of that, it is in fact possible to design either completely new typesetting systems or programmatic versions of Plain TEX, LATEX, *etc.* entirely in Lisp.

In order to maintain backward compatibility, we also need to implement "traditional" TEX (still in Lisp), that is, the surface layer consisting of both the macro versions of Lispified TEX primitives, and the rest of TEX's programming API. This is mostly located in TEX's mouth and eyes. Figure 1 depicts this architecture (disclaimer: this is an overly simplified, extremely naive view; see section 5.3 for details). As mentioned earlier, the whole point of this architecture being implemented in Lisp, in terms of extensibility and customizability, is that the user of TiCL can basically interfere at every level of the system, whether by adding personal functions, rewriting built-in functions or even internal typesetting algorithms.

Another advantage of this fully integrated approach is that it is actually quite simple to provide "mixed" functionality, that is, using Common Lisp code directly in an otherwise regular TEX source file. The only requirement is an escape syntax allowing Common Lisp to take over interpretation of Lisp code, and insert the result back into the regular TEX character stream. A prototype for this has already been implemented. It simply consists in a Common Lisp implementation of TEX's eyes with an additional bit of syntax: the appearance of two consecutive and equal subscript characters in a regular TEX source file will trigger the evaluation of the subsequent Lisp expression. Since this syntax is invalid in TEX, it cannot break any existing document.

Below is an example illustrating this idea (taken from [15]). The Lisp function `ast` is used to define a TEX macro `\asts` outputting a specified number of asterisks.

```
\documentclass{article}

\newcommand\asts{}
__(defun ast (n)
    (format nil "\\renewcommand\\asts{~A}"
      (make-string n :initial-element #\*)))

\begin{document}
__(ast 10)
\asts
\end{document}
```

For the curious, our current implementation of TEX's eyes in Common Lisp is roughly 200 lines. The support for the double subscript syntax (including parsing it, reading the Lisp code, evaluating it and inserting the result back into the regular TEX stream) amounts to only 16 lines, that is, around 8% of the total.

5.3 Expected problems

After all those "would" and "could", let us get to a more pessimistic (realistic?) view of the project. This section sheds some darkness on the too-bright picture we have drawn of TiCL until now. As mentioned earlier, TiCL is in fact pretty much only an idea at present. If this project ever comes to fruition, a number of problems are expected.

5.3.1 A huge task

Completely reimplementing TEX is a huge task. This is probably one of the reasons all alternative projects

(NTS excepted) chose the hybrid approach instead of the fully integrated one. One thing that may help is the existence of foreign function interfaces, notably for the C language. The project could be developed gradually by linking to the WEB/C implementation of the not-yet-reimplemented parts.

5.3.2 Compatibility

Another question to consider is whether the TEX-incompatible part would eventually be accepted by *the* (or *a new*) community. This question is in fact pertinent for LATEX3 as well and we don't have an answer. The existence of a compatibility mode with pluggable Lisp, as described in section 5.2 on the facing page, would probably help getting people accustomed to the benefits of using Lisp, while remaining in a reassuring context of traditional TEX.

5.3.3 TEX's organs

Figure 1 on the preceding page was already pointed out to be overly simplified. In reality, we know that the TEX organs don't constitute a pipeline. Some levels from "down below" do affect the upper stages which makes things much more complicated. In the long run, this may imply that it would be impossible to have new programmatic typesetting systems (accessing "procedural TEX" directly) work in conjunction with traditional TEX. Currently, we don't know for sure, although we are aware of the fact that this was one of the reasons the NTS project failed.

5.3.4 Mixed mode

Along those same lines, "mixed" mode, that is, the ability to mix regular TEX macros with Lisp code is expected to be tricky. If we want to provide more interaction than just the double subscript syntax, for example, the ability to define TEX macros in Lisp with Lisp access to the macro's arguments, we are bound to encounter issues related to the time of expansion.

This problem is in fact not related to Lisp at all, but rather to *any* approach aiming to provide the ability to define TEX macros in another language. Section 4 of [15] describes these kinds of problems in more detail.

5.3.5 Sandboxing

As with any scriptable application, the security problem is an important one. How much programmatic access to the application itself, or its surrounding environment, are we willing to give the end-user? Some versions of TEX are equipped with means to address this issue (*cf.* the `-shell-escape` command-line option and the `\write18` command). Using a true, comprehensive programming language with scripting capabilities makes this problem even worse because we have more than a couple of "macros" to control access to. We have a whole stack of language features to control, such as file I/O, operating system interfaces, *etc.* Currently, we are not aware of any sandboxing library already available for Lisp.

5.3.6 Intercession

This is in a similar vein as the sandboxing problem. In [21], we were complaining about (rejoicing in?) the huge intercession mess that the LATEX world is. Well, now you should *really* be afraid because we are going to make things worse! Remember that any Lisp executable granting scripting privileges to the end-user effectively provides write-access to the whole executable. This surely makes it trivial to extend or customize the system. Whether this is a good thing for the system in the long run, there is no way to tell. After all, LATEX *is* alive and kicking, in spite of (because of?) its high intercession capabilities...

6 Conclusion

In this paper, we advocated a "homogeneous" approach to TEX modernization in which TEX itself is first rewritten in a programming language serving both at the core of the program and at the scripting level. All programmatic macros of TEX are hence rendered obsolete, as the underlying language itself can be used for user-level programming. In a rather puzzling way, our notion of "modernity" consists of reimplementing a 30-year-old language using a 50-year-old one!

We demonstrated why we think that Common Lisp is a very good choice for doing this. Common Lisp is both a scripting language and a full-blown, industrial scale, general-purpose programming language at the same time. Common Lisp also provides several paradigms or idioms that are actually quite close to features that TEX either provides as well, or would like to provide.

Will the TiCL project ever come to birth? That is not sure at that point, as it will require a tremendous effort. We would like to see it happening, of course. Amongst all the current alternatives, LuaTEX seems to be the only one vigorously alive and gaining momentum.

Of course, the other project that needs to be mentioned is LATEX3. We only do so in an anecdotal fashion because it does not make use of a real programming language, but continues in the tradition of building directly on top of TEX. Still, LATEX3 is not experimental anymore and is light years ahead of what LATEX 2_ε is. In particular, it is much better than its ancestor in terms of syntax consistency and programmatic capabilities. Nevertheless, since it is still restricted to TeX's macro-expansion world, every time a new programming paradigm is needed, it will have to be implemented manually.

To sum up, the "big TeX modernization plan" currently seems to follow two different paths: staying strictly in the TeX area or creating hybrids of TeX and another real programming language. The approach we would like to suggest with TiCL is a third one: a homogeneous approach in which the implementation language of TeX is also the one which serves at the scripting level. Will this project see the light of day? Can these three approaches co-exist in the long term? Only the future will tell...

Epilogue

These were the voyages,
Of a software enterprise.
Its continuing mission:
To explore new tokens,
To seek out a new life,
New forms of implementation.
To `\textbf{go}`*,*
Where no TeX has gone before!

References

[1] Common Lisp. American National Standard: Programming Language. ANSI X3.226:1994 (R1999), 1994.

[2] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System specification. *ACM SIGPLAN Notices*, 23(SI):1–142, 1988.

[3] Rod Burstall. Christopher Strachey — Understanding programming languages. *Higher Order Symbolic Computation*, 13(1-2):51–55, 2000.

[4] Jonathan Fine. TeX forever! In *Proceedings EuroTeX*, pages 140–149, Pont-à-Mousson, France, 2005. DANTE e.V.

[5] Alan C. Kay. *The Reactive Engine*. PhD thesis, University of Hamburg, 1969.

[6] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, 1989.

[7] Gregor J. Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[8] Donald E. Knuth. *Digital Typography*. CSLI Lecture Notes. CSLI, September 1998.

[9] John MacCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3:184–195, 1960. Online version at `http://www-formal.stanford.edu/jmc/recursive.html`.

[10] Patty Maes. Concepts and experiments in computational reflection. In *OOPSLA*. ACM, December 1987.

[11] M. Douglas McIlroy. Macro instruction extensions of compiler languages. *Commun. ACM*, 3:214–220, April 1960.

[12] Marjan Mernik, editor. *Formal and Practical Aspects of Domain Specific Languages: Recent Developments*, chapter 1. IGI Global, 2012.

[13] Andrew Mertz and William Slough. Programming with PerlTeX. *TUGboat*, 28(3):354–362, 2007.

[14] Andreas Paepcke. User-level language crafting — Introducing the CLOS metaobject protocol. In Andreas Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, chapter 3, pages 65–99. MIT Press, 1993. Downloadable version at `http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps`.

[15] Scott Pakin. PerlTeX: Defining LaTeX macros using Perl. *TUGboat*, 25(2):150–159, 2004.

[16] Tim Sheard. Accomplishments and research challenges in meta-programming. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer, 2001.

[17] Brian C. Smith. Reflection and semantics in Lisp. In *Symposium on Principles of Programming Languages*, pages 23–35. ACM, 1984.

[18] J.E. Stoy and Christopher Strachey. OS6 — An experimental operating system for a small computer. Part 2: Input/output and filing system. *The Computer Journal*, 15(3):195–203, 1972.

[19] Didier Verna. Beating C in scientific computing applications. In *Third European Lisp Workshop at* Ecoop, Nantes, France, July 2006.

[20] Didier Verna. CLOS efficiency:instantiation. In *International Lisp Conference*, pages 76–90, MIT, Cambridge, Massachusetts, USA, March 2009. Association of Lisp Users.

[21] Didier Verna. Classes, styles, conflicts: The biological realm of LaTeX. *TUGboat*, 31(2):162–172, 2010. `http://tug.org/TUGboat/tb31-2/tb98verna.pdf`.

◇ Didier Verna
EPITA / LRDE
14-16 rue Voltaire
94276 Le Kremlin-Bicêtre Cedex
France
`didier (at) lrde dot epita dot fr`
`http://www.lrde.epita.fr/~didier`