TiCL: the Prototype

Didier Verna

Abstract

While TEX is unanimously praised for its typesetting capabilities, it is also regularly blamed for its poor programmatic offerings. 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 the Common Lisp language is used to implement a TEX-like typesetting system. The underlying implementation language serves both at the core of the program and at the scripting level, hence removing the need for programmatic macros on top of typesetting commands. A prototype implementation is presented.

1 Introduction

Last year at TUG 2012, we presented some ideas about using one of the oldest programming languages (Common Lisp [5, 1]), in order to modernize one of the oldest typesetting systems (TEX). The idea behind the term "modernization" here is to retain the quality of TEX's output (the typesetting part) while providing a more consistent and powerful programmatic API. This idea is not new and has led to several attempts at mixing TEX with a full blown programming language, such as with `eval4tex`[1] and sTEXme[2] (Scheme), PerlTEX [6, 7] (Perl) and QaTEX/PyTEX [3] (Python).

While these approaches to modernization are useful for the end-user who wishes to write his own set of macros in a more practical way, there is more to modernization than just the programmatic API. Another room for improvement lies in the design and flexibility (or lack thereof) of TEX's internals. TEX is an old program written in a way that doesn't meet today's standards. For example, it's full of corner case optimization that could arguably be considered obsolete on modern architectures, it lacks a potential Object Oriented design that would make it easier to extend it or even modify it, for example in order to experiment with alternative typesetting algorithms.

Approaches such as LuaTEX[3] attempt to address both aspects of modernization: mixing Lua with TEX provides both a scripting language on the surface layer, and access to TEX's internals at the same time.

[1] `http://www.ccs.neu.edu/home/dorai/eval4tex/`

[2] `http://stexme.sourceforge.net/`

[3] `http://www.luatex.org`

In our opinion, this approach still suffers from the following drawbacks.

1. Lua is *not* a full-blown, industrial scale programming language. It's a scripting language with a limited set of programming paradigms.
2. The resulting system is heterogeneous. It's a complex mixture of two different languages providing some limited level of introspection / intercession. Put differently, accessing TEX's internals from Lua requires manual plumbing.
3. Even with a sufficient amount of introspection / intercession, what you end up with still is "good ol'TEX", with its lack of modern software engineering concerns.

Last year at TUG 2012, we advocated the use of Common Lisp as a better alternative for all these aspects of "modernization". We explained that Common Lisp is both a full-blown industrial language suitable for a rewrite of TEX, and a scripting language suitable as a replacement for TEX macros. We then demonstrated the benefits of this homogeneous approach, in which a single programming language is involved: a re-implementation of TEX with modern software engineering in mind, in a fully reflexive language such as Common Lisp, would provide a system very convenient for extension and / or modification (hence experimentation). For more information, the reader is referred to [8].

The purpose of the present paper is to apply the ideas expressed last year in an actual prototype, in order to assert the validity of the proposed approach. Section 2 presents a general overview of the prototype. Section 3 on the next page gradually builds the prototype's programmatic layer, that is, the interface that an author could use in order to actually *program* a document. Section 4 on page 1005 presents an additional layer on top of the programmatic one, called the "textual layer", in which plain text constitutes the main entry, and calls to the programmatic layer need to be escaped. Finally, section 5 on page 1006 illustrates the advantages of the proposed approach in terms of extensibility, by enriching the prototype with a rivers detection feature in a non-intrusive way.

The code for the prototype itself, along with all the examples presented in this paper is available on GitHub[4].

2 Overview

One (obvious) conclusion from last year was that re-implementing a complete TEX-like typesetting system is a huge task. In particular, even for just a

[4] `https://github.com/didierverna/ticl-for-TUG2013`

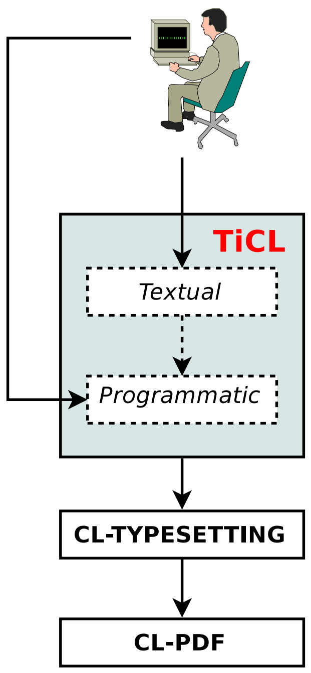

Figure 1: TiCL current architecture

prototype with basic typesetting features and actual output, an important question is: "where to start".

It so happens that a Common Lisp typesetting system already exists. This system, called `cl-typesetting`[5] provides a fair amount of features, including the obligatory paragraph and page breaking algorithms, and also PDF output thanks to a back-end called `cl-pdf`[6], from Marc Battyani, the same author.

`cl-typesetting` is very different from TeX, in ways that we will see later on, but nevertheless provides a convenient starting point for a TeX-like prototype. We would even go as far as claiming that it is even more interesting to start from something completely different from TeX, precisely because "bending" the system towards a TeX-like engine will illustrate the flexibility of the proposed approach.

Figure 1 depicts the current architecture of the prototype. TiCL is built on top of the existing typesetting system in order to reuse its core algorithms and its PDF output capabilities. Please note that at the time of this writing, it is unclear whether `cl-typesetting` would be retained in a real product.

The first TiCL layer, called the "programmatic layer", provides a set of commands mimicking some core LaTeX functionality such as `\documentclass`, `\maketitle`, sectioning commands *etc.* The purpose here is to have just enough features to be able to produce minimal documents such as basic articles. This layer will be described in more detail in section 3.

The second TiCL layer is called the "textual layer". When using this layer, a document author is provided with a text-oriented interface: the main input at that layer is the actual text to be typeset (this is exactly what TeX does). In order to execute commands, one can access the programmatic layer via an escape character (similar to category code 0 in TeX). This layer will be described in more detail in section 4 on page 1005.

As shown in figure 1, the particularity of this architecture is that it is possible to create a document directly at the programmatic layer, that is, without using the textual layer at all. The reasons why we think this is an interesting feature, and why it is possible at all will be explained at the end of section 3.

3 Programmatic Layer

Listing 1 presents a typical `cl-typesetting` Hello World program.

```
1  (defun hello (&key (file "/tmp/output.pdf"))
2    (tt:with-document ()
3      (let ((content (tt:compile-text ()
4                       (tt:paragraph ()
5                         "Some text..."))))
6        (tt:draw-pages content)
7        (when pdf:*page*
8          (tt:finalize-page pdf:*page*))
9        (tt:write-document file))))
```

Listing 1: `cl-typesetting`'s Hello World

As you can see, we are quite far from the world of TeX. The details are unimportant, but there are a couple of interesting things to note.

- The most important one is that in order to create a document, you need to write an actual Lisp *program*, or at least a function which does so.
- Everything happens within a call to a macro called `with-document` (line 2), which bears some resemblance with LaTeX's `document` environment.
- Lines 6 to 9 contain some boilerplate filling pages and writing the result to a file.
- Finally, the only textual content in this document is provided as a Lisp string (line 5) to a macro called `paragraph`.

3.1 Basic LaTeX functionality

The first thing we are going to do is provide some basic LaTeX-like functionality. To do that, we create

[5] http://www.fractalconcept.com/asp/cl-typesetting
[6] http://www.fractalconcept.com/asp/cl-pdf

some global variables for storing the title, author *etc.* Following the Lisp convention, these variables will be called `*title*`, `*author*` and so on. Then, we create several functions such as `document-class`, `make-title`, `table-of-contents` *etc.*, the purpose of which should be obvious. Next, we provide a `with-document` macro taking care of the boilerplate mentioned previously, and several sectioning macros, among which `with-section` and `with-subsection` and `with-par`. Finally, just for the fun of it, we add two functions, `textbf` and `textit`, the purpose of which should also be obvious. With this infrastructure in place, we are able to create a basic article as depicted in listing 2.

```
(document-class :article
                :paper :letter :pt 12)

(setq *title* "An Article")
(setq *author* "Didier Verna")

(with-document

  (make-title)
  (table-of-contents)

  (with-section "Lorem Ipsum"
    (with-subsection "Sit Amet"
      (with-par "Lorem " (textbf "ipsum") " "
        (textit "dolor") " sit amet, ...")
      (with-par "Lorem " (textbf "ipsum") " "
        (textit "dolor") " sit amet, ...")))


  ;; other sections ...)
```

Listing 2: A basic article

Note that this document is in fact a Lisp *program.* In order to generate the corresponding PDF, you need to execute (evaluate) this program. In Common Lisp, one can simply evaluate expressions at the REPL (the "Read Eval Print Loop"), or store programs in files and `load` them. By convention, we will store this document in a file with extension `ticl`. The current prototype allows you to compile your documents interactively at the REPL, but also provides a standalone command-line executable for creating `foo.pdf` automatically from `foo.ticl`.

This program is full of Lisp idioms, and of course very far from what a casual LaTeX user would be willing to type in an alternate typesetting system. One first, very simple improvement is to wrap global variables access into function calls, so that we can use `(title "An Article")` instead of `(setq *title* "An Article")`. The `title` function looks like a getter instead of a setter, so it is not very lispy, but that is the way LaTeX does it. In the remainder of this section, we gradually move away from the Lisp Way$^{\text{TM}}$ and get closer the LaTeX one.

3.2 Lisp macros for symbolic options

The next thing we can do is to make command arguments look a little better. Consider again our equivalent to `\documentclass`:

```
(document-class :article
  :paper :letter :pt 12)
```

This function takes one mandatory argument, and a list of optional, named ones. Symbols of the form `:name` in Lisp are called "keywords" and are used for naming optional arguments to functions. They also have the property that they evaluate to themselves, which makes them a good choice for denoting symbolic constants. This, however, would probably confuse a casual user. For example, `:article` is an option, but `:paper` is an option's name, and `:letter` is its value. We can arrange this by creating a Lisp wrapper *macro* on top of the `document-class` function. Lisp macros look like functions but don't evaluate their arguments unless you require so. This allows us to write something like this:

```
(documentclass article
  :paper letter :pt 12)
```

The `documentclass` macro will prevent the evaluation of the symbols `article` and `letter`, which would otherwise be interpreted as (unknown) variable names.

3.3 Symbol macros for 0-ary functions

In Lisp, `(foo)` is a function call whereas `foo` is a variable call. This syntactic difference doesn't exists in TeX, which is a macro expansion system. The actual behavior of `\foo` simply depends on how the macro is defined. We can get closer to this idea in Lisp by using so-called *symbol macros.* A symbol macro is a symbol which behaves as a macro, that is, which expands into an arbitrary Lisp form, including function calls. We can use this facility to define `maketitle` and `tableofcontents` as symbol macros expanding to `(make-title)` and `(table-of-contents)` respectively.

3.4 Less environments, less macros

Listing 2 introduced a number of `with-foo` macros, a typical Lisp idiom. Such macros usually wrap around a body of code, doing some pre- and post-processing around it. This idea is actually quite close to the concept of "environment" in LaTeX, but we will get back to this later.

Even though we think that sectioning *environments* are a better approach (for one thing, they are more friendly to PDF), LaTeX only provides sectioning *commands*. Replacing our `with-` macros with regular functions is not very difficult. Most of the time, it boils down to getting rid of the macro layer and just use its expansion directly. We hence provide regular functions such as `(section ...)` and `(par)`, and we also take the opportunity to implement a `par` symbol macro which will expand to the `(par)` function call.

At that point, we are able to rewrite our original document generating program into a new, intermediate form, depicted in listing 3. We are getting closer to LaTeX, although we're not quite there yet.

```
(documentclass article :paper letter :pt 12)

(title "An Article")
(author "Didier Verna")

(with-document

  maketitle
  tableofcontents

  (section "Lorem Ipsum")
  (subsection "Sit Amet")
  "Lorem " (textbf "ipsum") " "
  (textit "dolor") " sit amet, ..."
  par
  "Lorem " (textbf "ipsum") " "
  (textit "dolor") " sit amet, ..."


  ;; other sections ...)
```

Listing 3: Intermediate program state

3.5 OO hot-patching for empty lines

One very convenient syntactic trick in TeX, and one of its rare concessions to the WYSIWYG approach, is its treatment of empty lines as calls to `\par`. Doing the same thing in TiCL would be nice, but is less straightforward than what we've done until now, because strings are processed by `cl-typesetting` directly. We hence need to modify it instead of just build on top of it.

This, however, turns out to be not so difficult, which is in fact the whole point of this article. The two key factors in making this modification trivial are the facts that 1. `cl-typesetting` is designed on top of CLOS, the Common Lisp Object System [2, 4], and that 2. Lisp is a dynamic language.

```
(defmethod tt::insert-stuff ((obj string))
  `(put-string ,obj))
```

Listing 4: A method for typesetting strings

Listing 4 shows how `cl-typesetting` handles regular strings of text. There is a global *generic function* called `insert-stuff` in charge of typesetting all sorts of Lisp objects (strings, glues, boxes *etc.*). This function has a different *method* (in the object-oriented sense) for every kind of typesettable object.

As a consequence, what we need to do is simply to override the method for strings with our own, and also provide our own version of `put-string`, dealing with consecutive newlines. This new function can also reuse the original one for strings containing only one paragraph. Needless to say, the simplicity of this approach comes from the fact that `cl-typesetting` is an object-oriented library, and as such, our modifications are localized and contained only in very specific software components.

Another important thing here is the fact that Lisp is a dynamic language. Consider for a minute the equivalent problem in a language such as C++. One would need to get the source code of the original library, modify it, recompile it, and ship the modified version along with the new system (in either source or executable form). In the case of a dynamic language, the required modifications can be done at run-time, when the system loads-up. The original library is not needed (not even in source form) in order to implement the modifications to it. One just overrides one of its components on the fly, pretty much like any TeX macro can be rewritten at any time.

At that point, we are able to rewrite our document generating program into yet another intermediate form, depicted in listing 5. Note the string starting on line 14 and ending on line 16. It contains an implicit paragraph.

```
1   (documentclass article :paper letter :pt 12)
2
3   (title "An Article")
4   (author "Didier Verna")
5
6   (with-document
7
8     maketitle
9     tableofcontents
10
11    (section "Lorem Ipsum")
12    (subsection "Sit Amet")
13    "Lorem " (textbf "ipsum") " "
14    (textit "dolor") " sit amet, ...
15
16    Lorem " (textbf "ipsum") " "
17    (textit "dolor") " sit amet, ..."
18
19
20    ;; other sections ...)
```

Listing 5: Another intermediate program state

3.6 Syntax extension with macro-characters

The final problem we need to address is that of the `document` environment. For technical reasons that are too complex to explain here, it is not possible, or at least, it would be very difficult to get rid of the macro `with-document` and replace it with commands like `begin-document` and `end-document`. However, we would still like to have something closer to LaTeX's environment syntax.

Fortunately, Lisp offers a way out via syntax extension. The syntax of Lisp code is governed by so-called *readtables*. A readtable defines the syntactic status of every character the parser can encounter. It is possible to modify or extend the standard Lisp syntax by providing a custom readtable. You can then make any character syntactically "active" by turning it into a so-called *macro-character*. Every time the Lisp reader encounters such a character, a user-provided routine (called a *"reader macro"*) takes over parsing, and returns a new syntactic form modified at will.

The resemblance of this with TeX's notion of category codes should be striking. We can use this facility to solve our problem as follows: we will make the brace characters (`{` and `}`) active, and let them read forms such as `{begin env}` and `{end env}`. Our reader macro will then read the environment's contents, and wrap it inside a call to the corresponding `with-env` macro. Essentially, we are turning this:

```
{begin env} ... {end env}
```

into this:

```
(with-env ...)
```

And again, note that this is done during the parsing phase, so this really is some kind of source-to-source transformation. One drawback of this approach is that it makes the syntax a little bit more complicated, less regular. Another slightly more intrusive possibility is to modify the original reader macro for the left parenthesis character, in order to recognize forms like `(begin env)` and `(end env)`. This requires modifying the Lisp engine itself with the exact same technique, but we now know we can do that.

At that point, we are able to rewrite our document generating program into its final form, depicted in listing 6. Whether we use our specific brace syntax or modify the standard parenthesis one will only affect lines 6 and 20.

If we wanted to get even closer to TeX, we could also use the same reader technique to change the comment character from `;` (standard Lisp) to `%`...

```
1  (documentclass article :paper letter :pt 12)
2
3  (title "An Article")
4  (author "Didier Verna")
5
6  {begin document} ;; or (begin document)
7    maketitle
8    tableofcontents
9
10   (section "Lorem Ipsum")
11   (subsection "Sit Amet")
12   "Lorem " (textbf "ipsum") " "
13   (textit "dolor") " sit amet, ...
14
15   Lorem " (textbf "ipsum") " "
16   (textit "dolor") " sit amet, ..."
17
18
19   ;; other sections ...
20 {end document} ;; or (end document)
```

Listing 6: Final programmatic form

3.7 Wrap-up

Listing 6 is probably the closest we can get to a LaTeX-looking *program*, while maintaining a conventional and reasonably regular Lisp syntax. Of course, it is doubtful that non-programmers (let alone non-lispers) would ever want to use a system like that. There are nevertheless some interesting points to be made here.

First, the fact that this document really is a program brings a lot of perspectives in terms of automation. One can envision all sorts of typesetting applications made easy, such as reference manual generators or database dumpers, in which instead of hand-typing all strings of text manually, you *program* the extraction of the text to be typeset.

Next, and still because we're in front of a Lisp program, we can reuse the standard Lisp tool-chain on this document. For example, generating the actual PDF is a simple matter of calling the function `load` on the document's source file. But we can also do more. We can also `compile` the code and even generate a standalone executable out of it, that we can distribute. Relate this to the idea of creating a LaTeX document that people could actually typeset without having to install LaTeX locally...

4 Textual Layer

Section 3 described the programmatic layer of TiCL. In this section, we present the textual layer, that is, the layer which an end-user would use to create documents. In this layer, just as in TeX, text is the main component, and access to the programmatic layer needs to be escaped. The idea is actually quite simple and requires only a 42 lines long function in the TiCL prototype.

- A `convert` function reads a textual source file (extension `ltic`, for "literate TiCL") character by character.
- Every time a backslash is encountered, the subsequent expression is read in as a piece of Lisp code (presumably accessing the programmatic layer). Note that we don't have to write any code for reading Lisp source, as Lisp itself already provides the required API.
- When the expression in question is of the form `(begin ...)`, we turn that into `(with-...`. When it is of the form `(end ...)`, we turn that into a closing parenthesis.
- In any other case, the characters are accumulated as Lisp strings.

That is basically it. This function allows us to rewrite our document as depicted in listing 7.

```
\(documentclass article :paper letter :pt 12)

\(title "An Article")
\(author "Didier Verna")

\(begin document)
  \maketitle
  \tableofcontents

  \(section "Lorem Ipsum")
  \(subsection "Sit Amet")
  Lorem \(textbf "ipsum")
  \(textit "dolor") sit amet, ...

  Lorem \(textbf "ipsum")
  \(textit "dolor") sit amet, ...


  %% other sections ...
\(end document)
```

Listing 7: The textual form

Given the simplicity of the syntax, we believe it would be easy for a casual LaTeX user to adapt to this new surface. Note in particular that thanks to symbol macros (section 3.3 on page 1003), calls like `\maketitle` look *exactly* the same in TiCL and in LaTeX.

5 Extensibility

One last thing we want to illustrate here is the virtues of Lisp for extensibility. As an example of this, our prototype extends the original system with a rivers detection feature in less than 75 lines of code. The reader interested in the technical details may look at the example files called `rivers.ticl` or `rivers.ltic` in the distribution. In this article, we only provide an outline of the implementation.

As we have already seen, `cl-typesetting` provides different kinds of typesettable objects (strings, glues *etc.*). One such kind of objects is the `vbox` (vertical box). Our idea is to provide a specific kind of vertical box on which to perform rivers detection. In order to do that, we define a new *class* of typesettable object called `riversbox`, which we implement as a subclass of `cl-typesetting`'s `vbox` class (again, we see here the importance of the OO design):

```
(defclass riversbox (tt::vbox) ())
```

Next, we implement a macro called `with-rivers` which collects all of its content and puts it inside a fresh riversbox. This automatically makes it possible to use the corresponding `rivers` environment as follows:

```
\(begin rivers)
Paragraph material...
\(end rivers)
```

The next thing we want to do is to draw rivers in red in this environment (in addition to drawing the content as usual). `cl-typesetting` draws its material thanks to a generic function called `stroke`. There is a specific method for vertical boxes that riversboxes will automatically inherit, as they are a subclass of vertical boxes. Hence, we don't have anything to do in this regard.

What we have to do is actually draw the rivers on top of the regular text. We can plug this into the typesetting engine easily, thanks to CLOS's notion of *before method*: methods that are called before the primary ones, in addition to them:

```
(defmethod tt::stroke :before
    ((box riversbox) x y)
  (draw-rivers box x y))
```

Finally, the additional function `draw-rivers`, introspects the contents of the riversbox, collects the spaces and draws red lines where appropriate in 33 lines of code.

Again, the important thing here is that this extension is hot-plugged into the system, in a non intrusive way, thanks to the dynamic nature of the underlying language. In fact, in the prototype's distribution, this extension is not implemented in the typesetting system, but directly in the preamble of one of the sample documents!

6 Conclusion

Last year at TUG 2012, we presented some ideas about using one of the oldest programming languages (Common Lisp), in order to modernize one of the oldest typesetting systems (TeX), the idea being to retain the quality of TEX's output (the typesetting part) while providing a more consistent and powerful programmatic API.

We advocated the use of Common Lisp as a better alternative for all aspects of modernization: Common Lisp is both a full-blown industrial language suitable for a rewrite of TEX, and a scripting language suitable as a replacement for TEX macros. The use of a single language allows to design an implement a homogeneous system: a re-implementation of TEX with modern software engineering in mind, in a fully reflexive way, providing easy extension and / or modification capabilities (hence experimentation).

Our purpose this year was to validate the proposed approach by implementing a working prototype, which we did. TiCL has a simple syntax making the transition from regular TEX easy. It lets you use the whole Lisp language for programming. It can be accessed both at the programmatic and at the textual layer, depending on the kind of practical application you have in mind. Finally, it is inherently extensible, due to the dynamic and reflexive nature of Lisp, as demonstrated in section 5 on the preceding page.

We find those results encouraging, and we definitely want to continue investigating in that direction.

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] Jonathan Fine. TEX forever! In *Proceedings EuroTEX*, pages 140–149, Pont-à-Mousson, France, 2005. DANTE e.V.

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

[5] 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`.

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

[7] Scott Pakin. PerlTEX: Defining LATEX macros using Perl. *TUGboat*, 25(2):150–159, 2004.

[8] Didier Verna. Star TEX: the next generation. In Barbara Beeton and Karl Berry, editors, *TUGboat*, volume 33. TEX Users Group, 2012.

◇ 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`