1 Coffee(Xe)(La)TeX

1.1 What is it? And Why?

Everyone who has worked with LaTeX knows how hard it can often be to get seemingly simple things done in this Turing-complete markup language. Let's face it, (La)TeX has many problems; the complectedness of its inner workings and the extremely uneven syntax of its commands put a heavy burden on the average user. The funny thing is that while TeX is all about computational text processing, doing math and string processing are *really hard* to get right—or even get done at all—in this environment (not to mention that TeX has no notion of higher-order data types, such as, say, lists).

Often one wishes one could just do a simple calculation or build a typesetting object from available data *outside* of all that makes LaTeX so difficult to get right. Turns out you can already do that, and you don't have to recompile TeX.

Most of the time, running TeX means to author a source file and have the TeX executable convert that into a PDF. Of course, this implies reading and writing of files and executing binaries. Interestingly for us, both capabilities—file access and command execution—are made available to user-facing side of TeX: writing to a file happens via the `\write` command, while input from a file is done with `\input`; command execution repurposes `\write`, which may be called with the special stream number `18` (internally, TeX does almost everything with registers that are sometimes given symbolic names; it also enumerates 'channels' for file operations, and reserves #18 for writing to the command line and executing stuff). This is how the `\exec` command is defined in CoffeeXeLaTeX:

```
\newcommand{\coffeexelatextemproute}{/tmp/coffeexelatex.tex}

\newcommand{\exec}[1]{%
  \immediate\write18{#1 > \coffeexelatextemproute}
  \input{\coffeexelatextemproute}
  }
```

This command says, in essence: given a single argument `#1`, have the OS execute it as a command, and do that immediately (i.e. not at some arbitrary point in the future); redirect its output into the temporary file `/tmp/coffeexelatex.tex`; then, read back the contents of that file, and insert that text into the current TeX source.

With some TeXs, its possible to avoid the temporary file by using `\@@input|"dir"`, but XeTeX as provided by TeXLive 2013 does not allow that. As much as i'd like to eliminate the somewhat unelegant temporary file (that brings one more possible point of failure to the equation; remember to `\renewcommand{\coffeexelatextemproute}{some/temp/location.tex}` where deemed necessary), it also has one advantage: in case TeX should halt execution becuase of an error, and that error is due to a script with faulty output, you can conveniently review the problematic source by opening the temporary file in your text editor.

1.2 Security Considerations

Be aware that executing arbitrary code with a command line like `xelatex --enable-write18 --shell-escape somefile.tex` is inherently unsafe: a TeX file downloaded from somewhere could erase your disk, access your email, or install a program on your computer. This is what the `--enable-write18` switch is for: it is by default fully or partially disabled so an arbitrary TeX source gets limited access to your computer.

If you're scared now, please hang on a second. I just want to tell you *you should be really, really scared*. Why? Because if you ever downloaded some TeX source to compile it on your machine **even without the `--enable-write18` switch** you've already **executed potentially harmful code**. Few people are aware of it, but many TeX installations are quite 'liberal' in respect to what TeX sources—TeX programs, really—are allowed to do *even in absence of command line switches*, and, as a result, even people who are hosting public TeX installations for a living are susceptible to malicious code.*

It is a misconception that TeX source is 'safe' because 'TeX is text-based format' (how stupid is that, anyway?); **the truth is that by doing `latex xy.tex` you're executing code which may do malicious things**. Period. That said, the papers linked below make it quite clear that `--enable-write18` just 'opens the barn door', as it were, but in fact, there are quite a few other and less well known avenues for TeX-based malware to do things on your computer.

And please don't think you're safe just because you're not executing anything but your own TeX source—that, in case you're using LaTeX, is highly improbable: any given real-world LaTeX document will start with a fair number of `\usepackage{}` statements, and each one of those refers to a source that is publicly accessible on the internet and has been so for maybe five or ten or more years. Someone might even have managed to place a mildly useful package on CTAN, one that has some obfuscated parts designed to take over world leadership on Friday, 13th—who knows? **The fact that TeX is a programming language that works by repeatedly re-writing itself does not exactly help in doing static code analysis**; in fact, such code is called 'metamorphic code' and is a well-known technique employed by computer viruses.

I do not write this section of the present README to scare you away, just to inform whoever is concerned of a little known fact of life. The gist of this is: don't have `--enable-write18` turned on except you know what you're doing, but be aware that running TeX has always been unsafe anyway.

*) see e.g. http://cseweb.ucsd.edu/~hovav/dist/tex-login.pdf, http://cseweb.ucsd.edu/~hovav/dist/texhack.pdf

1.3 Installation

- put a symlink to your CoffeeXeLaTeX directory into a directory that is on LaTeX's search path; on OSX with LiveTeX, that could be `~/Library/texmf/tex/latex`.*

  *: obviously, you could put your CoffeeXeLaTeX installation directly there, but that strikes me as 'wrong'.

1.4 Usage

For a quick test, do

```
cd examples/example-1
perltex --nosafe --latex=xelatex example-1.tex
```

from the command line; this should produce `examples/example-1/example-1.pdf` (along with some other files).

You may want to have a look at `examples/example-1/example-1.lgpl` to get an idea what exactly happened behind the scenes (see PerlTeX: Defining LaTeX macros using Perl for an overview of the process).

1.5 Future Development

If CoffeeXeLaTeX turns out to be a useful tool, i can presently see the following routes for development:

- Using a Perl shell-escape command to start `node` all over for each single JS/CS macro is doubly wasteful—first, a `perl` process is started which in turn starts a `node` subprocess. Depending on specific use, this can mean that two processes (none of them exactly lightweight) are initiated hundreds or thousands of times for a single document. This may be fixed in a number of ways:

- Firstly, we could try and remove the Perl dependency, and call `node` in exactly the way that `perl` is called now. Not sure how to do that at this point in time.

- Secondly, we could opt for a client / server model and make it so that instead of starting a (heavy) process for each JS/CS macro call, a (HTTP?) connection to a long-running NodeJS server is established. This is also attractive as it would simplify state keeping—as it stands, each call to a given macro starts with a clean slate (though one could imagine storing results from past calls in a file or a database).

Both of the above options are only worth implementing when it has been shown that substantial benefits in terms of performance, easy of use, and capabilities can be gained—something that is only meaningful after experience with real-world use cases has been gained.

- The one big incentive for using a 3rd-party language in tandem with LaTeX is to make things easy (or at least achievable) that are difficult (or impossible) using only LaTeX.

Regrettably, our efforts are still limited to what can be communicated 'over the wire' between the LaTeX process and the macro process; we do not have direct access to (La)TeX internals as such, but must package every pertinent facet of the ongoing typesetting process as a textual argument for a macro.

Imagine you had to interact with your HTML page in this way—imagine JavaScript in the browser was stateless and blissfully unaware of the DOM and CSS, imagine HTML was Turing-complete-but-hard-to-use as is the case with TeX. Your capabilities-improved web application page would be littered with ugly `<if condition='...'>...</if>` tags and circumlocutorily reified calls to JS. This is the state of affairs of PerlTeX / CoffeeXeLaTeX, and it is definitely a programming model begging to be improved.

1.6 Good to Know

The PerlTeX Manual warns users that "`perltex.pl` may hang if LaTeX exits right before the final pipe communication", but unfortunately says nothing about the circumstances leading to such a condition nor how to avoid them. For the time being, users of PerlTeX and, by extension, CoffeeXeLaTeX have to live with the fact that every so often, a running LaTeX job will just stop doing anything, typically displaying something like `(./example.pipe)` as last line of output. I then routinely hit `Ctrl-C` and restart the process, keeping fingers crossed.

1.7 Related Work

- PythonTeX is an interesting approach to bringing LaTeX and Python together. Unfortunately, the authors are preconcerned with showing off Pygment's syntax hiliting capabilities (which are … not really that great) and how to print out integrals using SymPy, and fail to provide sample code of interest to a wider audience. Their copious 128-page manual only dwells for one and a half page on the topic of 'how do i use this stuff', and that only to show off more SymPy capabilities. None of their sample code *needs* PythonTeX anyway, since none of it demonstrates how to interact with the document typesetting process; as

such, all their formulas and plots may be produced offline, independently from LaTeX. Given that the installation instructions are too scary and longwinded for my taste, and that PythonTeX is not part of LiveTeX, i've given up on the matter.

(the below taken from http://get-software.net/macros/latex/contrib/pythontex):

- SageTEX allows code for the Sage mathematics software to be executed from within a LATEX document.
- Martin R. Ehmsen's `python.sty` provides a very basic method of executing Python code from within a LaTeX document.
- SympyTeX allows more sophisticated Python execution, and is largely based on a subset of SageTeX.
- LuaTeX extends the pdfTeX engine to provide Lua as an embedded scripting language, and as a result yields tight, low-level Lua integration.

LuaTeX is one of the most interesting projects in this field as it represents an attempt to provide a close coupling of a real programming language with LaTeX. Unfortunately, that language is Lua, a language that (like Go btw) believes that Unicode strings should be stored as UTF-8 bytes. Equally unfortunately, LuaTeX uses pdfTeX, which can't compare to XeLaTeX when it comes to using custom TTF/OTF fonts.

1.8 Useful Links

http://www.ctan.org/tex-archive/macros/latex/contrib/perltex

http://ctan.space-pro.be/tex-archive/macros/latex/contrib/perltex/perltex.pdf

http://www.tug.org/TUGboat/tb28-3/tb90mertz.pdf

https://www.tug.org/TUGboat/tb25-2/tb81pakin.pdf

2 Introduction

(1) The original technique to execute an arbitrary command:

```
\immediate\write18{node
  "\CXLTXmainroute"
  "\currfilepath"
  "helo"
  "readers (one)"
  > /tmp/temp.dat}\input{/tmp/temp.dat}
```

(2) With ugly details largely hidden, the `\exec{}` command is still fully general:

```
\exec{node
  "\CXLTXmainroute"
  "\currfilepath"
  "helo"
  "readers (two)"}
```

(3) `\noderunscript{}` will execute NodeJS code that adheres to the call convention established by *Coffee*XeLaTeX:

```
\noderunscript
  {\CXLTXmainroute}
  {\currfilepath}
  {helo}
  {readers (three)}
```

(4) Like the previous example, but with standard values assumed as shown above. This is the form that you will want to use most of the time:

```
\noderun{helo}{readers (four)}
```

Outputs:

3 Configuration

To use *Coffee*XeLaTeX, put

```
\usepackage{coffeexelatex}
```

into the header section of your LaTeX file.

You may also want to include these lines in your LaTeX document; these define, respectively, the route to the *Coffee*XeLaTeX executable (`coffeexelatex/lib/main.js`) and the temporary file that is used to communicate between TeX and your scripts (relative routes are resolved with respect to the current working directory, so if you set a relative route, you must always run TeX from within the same directory):

```
\renewcommand{\CXLTXmainroute}{../../lib/main}
\renewcommand{\CXLTXtempoutroute}{/tmp/coffeexelatex.tex}
```

4 Character Escaping

The *Coffee*XeLaTeX command `show-special-chrs` demonstrates that it is easy to include TeX special characters in the return value. The simple rule is that whenever the output of a command is meant to be understood literally, it should be `@escape`d:

Command:

```
\noderun{show-special-chrs}{}
```

Output:

5 The aux Object

5.1 Labels

*Coffee*XeLaTeX will try and collect all labels from the `*.aux` file associated with the current job; from inside your scripts ▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬

Command:

```
\noderun{show-aux}{}
```

Output:

5.2 Evaluating Expressions

The commands `\evalcs{}` and `\evaljs{}` allow you to evaluate an arbitrary self-contained expression, written either in CoffeeScript or in JavaScript:

Command:

```
$23 + 65 * 123 = \evalcs{23 + 65 * 123}$
```

Output:

$23 + 65 * 123 = 8018$

6 Curl

```
\curlRaw{127.0.0.1:8910/foobar.tex/helo/friends}
```

Hello, friends!

6.1 Unicode

```
\curl{helo}{黎永強}
```

Hello, 黎永強!

```
\curl{helo}{äöüÄÖÜß}
```

Hello, äöüÄÖÜß!

6.2 The URL environment

Typing URLs in LaTeX can be quite a chore, given the number of active and otherwise 'special' characters to take care of: not only does TeX itself define some special characters, not only do the RFCs that govern URL syntax consider ▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬ special —when we communicate with our *Coffee*XeLaTeX server, we do so by executing a `curl ...` command via the OS shell (normally `sh`), which again has its own rich set of specials. In order to alleviate the burden on the casual user, we define a new environment, 'URL', that somewhat simplifies writing (parts of) URLs:

```
\begin{URL}
\curl{helo}{`\{ [ $ ~ % \# ^ | \} ] '?}
\end{URL}
```

Hello, '{ [$ ~ % # ^ | }] '?!