CoffeeXeLaTeX

(CXLTX)

How to

Extend LaTeX

with

A Real Programming Language

Contents

Conventions: In this document, (most) command line / code stuff is printed in red, while (most) remote command output is printed in green.

Click here to read the full documentation

1 CoffeeXeLaTeX (CXLTX)

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 often really hard to get right** (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 (in essence) defined in `coffeexelatex.sty`:

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

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

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. The temporary file does have an advantage: in case TeX should halt execution because 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.

Besides `\exec`, there is also `\execdebug` which captures the `stderr` output of a command and renders it in red to the document in case any output occurred there.

1.2 TeX and NodeJS

1.2.1 Method One: Spawning a `node` process

The whole idea of CXLTX is to have some external program receive data from inside a running TeX invocation, process that data, and pass the result back to TeX.

The command given to `\exec` could be anything; for our purposes, we will concentrate on running NodeJS programs. The simplest thing is to have NodeJS evaluate a JavaScript expression and print out the result; the `\evaljs` command has been defined to do just that:

```
\newcommand{\evaljs}[1]{\execdebug{node -p "#1"}}

% will insert `16 * 3 = 48` in TeX math mode (triggered by `$`):
$ 16 * 3 = \evaljs{16 * 3} $
```

This technique is easily adapted to work with CoffeeScript (or any other language that compiles to JS):

```
\newcommand{\evalcs}[1]{\execdebug{coffee -e "console.log #1"}}

% will insert `[1,4,9]`
\evalcs{ ( n * n for n in [ 1, 2, 3 ] ) }
```

The Command Line Remote Command Interface (CL/RCI) Of course, evaluating one-liners will give you only so much power, so why not execute more sizeable programs? That's what `\nodeRun` is for:

```
\usepackage[abspath]{currfile}
\newcommand{\CXLTXmainRoute}{../../lib/main}
\newcommand{\nodeRun}[2]{
  \exec{node "\CXLTXmainRoute" "\currfileabsdir" "\jobname" "#1" "#2"}}

% will insert `Hello, world!`
\nodeRun{helo}{world}
```

`\NodeRun` will run NodeJS with the following arguments: **(1)** the route to our custom-made executable; **(2)** the route to the parent directory where the currently processed TeX source files are; **(3)** the current job name; **(4)** a command name (first argument to `\nodeRun`); and, lastly, **(5)** optional command arguments (second argument to `\nodeRun`).

Observe that you may want to define your own routes in case the default values do not match your needs; these can be easily done using `\renewcommand`:

```
\renewcommand{\CXLTXmainRoute}{../../lib/main}
\renewcommand{\CXLTXtempOutRoute}{/tmp/CXLTXtempout.tex}
\renewcommand{\CXLTXtempErrRoute}{/tmp/CXLTXtemperr.tex}
```

1.2.2 Method Two: Spawning `curl`

Spawning a subprocess to 'outsource' a computing task is certainly not the cheapest or fastest way to do stuff, as creating and taking down a process is relatively costly.

More specifically, spawning `node` is both comparatively expensive (in terms of memory) and slow. Also, the subprocess must run on the same machine as the main process, and unless that subprocess persisted some data (in a DB or in a file), the subprocess will run in a stateless fashion.

Then again, since we're using NodeJS anyway: What's more obvious than to make TeX communicate with a long-running HTTP server?

Now i've not heard of any way to make TeX issue an HTTP request on its own behalf. But we already know we can issue arbitrary command lines, so we certainly can spawn `curl localhost` to communicate with a server. We still have to spawn a subprocess that way—but maybe `curl` is both lighter and faster than `node`!? It certainly is.

Here are some timings i obtained for running our simple echoing example, once spawning `node`, and once spawning `curl`. The server is a very simple Express application; except for the command line argument and HTTP parameter handling, the same code is ultimately executed:

```
time node "cxltx/lib/main" "cxltx/doc/" "cxltx-manual" \
  "helo" "friends" \
  > "/tmp/CXLTXtempout.tex" 2> "/tmp/CXLTXtemperr.tex"

real  0m0.140s
real  0m0.082s
real  0m0.083s

time curl --silent --show-error \
  127.0.0.1:8910/foobar.tex/cxltx-manual/helo/friends \
  > "/tmp/CXLTXtempout.tex" 2> "/tmp/CXLTXtemperr.tex"

real  0m0.010s
real  0m0.011s
real  0m0.010s
```

In so far these rather naïve benchmarks can be trusted, they would indicate that fetching the same insignificant amount of data via `curl` from a local server is around ten times as performant as doing the same thing by spawning `node`. The reason for this is partly attributable to the considerable size of the NodeJS binary (respectively whatever actions are taken by NodeJS to ready itself).

Even doing `time node -p "1"` results in execution times of over 0.07s. Add to this the `curl` timings of around 0.01s, and you roughly get the 0.08s needed to spawn `node` and get results—in other words, `curl` itself seems to be quite fast.

The upshot of this is that using our first method we can call external code at a frequency around 10 per second, but using the `curl` method, we can get closer to almost 100 per second (depending on the machine etc). The difference might matter if you plan to put out a *lot* of external calls, and since the typical way of getting TeX source code right is running and re-running TeX a lot of times, doing it faster may help greatly.

```
% This is the default setting for calling the CXLTX RCI:
\renewcommand{\node}{\nodeCurl}

\node{helo}{friends}
```

1.2.3 Security Considerations

Be aware that executing arbitrary code by way of mediation of 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/texhack.pdf`

1.3 Sample Command Lines

To make it easier for TeX to resolve `\usepackage{cxltx}`, put a symlink to your CXLTX directory into a directory that is on LaTeX's search path. On OSX with LiveTeX, that can be achieved by doing

```
cd ~/Library/texmf/tex/latex
ln -s route/to/cxltx cxltx
```

Here is what i do to build `cxltx/cxltx-manual.pdf`:

(1) use Pandoc to convert `README.md` to `README.tex`:

```
pandoc -o cxltx/doc/README.tex cxltx/README.md
```

(2) copy the `aux` file from the previous TeX compilation step to preserve its data for CXLTX to see:

```
cp cxltx/doc/cxltx-manual.aux cxltx/doc/cxltx-manual.auxcopy
```

(3) compile `cxltx-manual.tex` to `cxltx-manual.pdf` (`--enable-write18`: allows to access external programs form within TeX; `--halt-on-error`: is a convenience so i don't have to type `x` on each TeX error; `--recorder`: needed by the `currfile` package to get absolute routes):

```
xelatex \
  --output-directory cxltx/doc \
  --halt-on-error \
  --enable-write18 \
  --recorder \
  cxltx/doc/cxltx-manual.tex
```

(4) move the pdf file to its target location:

```
mv cxltx/doc/cxltx-manual.pdf cxltx
```

1.4 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

1.5 Related Work

¶ **PyTeX** (also dubbed QATeX) is a laudable effort that has, sadly, been stalling for around 11 years as of this writing (January 2014), so it is likely pretty much outdated. PyTeX's approach is apparently the opposite of what we do in CXLTX: they run TeX in daemon mode from Python,

where we have NodeJS start a server that listens to our independently running TeX.—Just for giggles, a quote from the above page: "XML is hard work to key by hand. *It lacks the mark-up minimization that SGML has*" (my emphasis). Well, eleven years is a long time.

¶ **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; sadly, they 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, whose designers believe that Unicode strings should be stored as UTF-8 bytes (Go does the same, btw). Equally unfortunately, LuaTeX uses pdfTeX, which can't compare to XeLaTeX when it comes to using custom TTF/OTF fonts.

2 Examples

2.1 Spawning NodeJS

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

```
\immediate\write18{node
  "\CXLTXcliRoute"
  "\CXLTXtexRoute"
  ","
  "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
  "\CXLTXcliRoute"
  "\CXLTXtexRoute"
  ","
  "helo"
  "readers (two)"}
```

(3) 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 (if you want to use the CL/RCI at all):

```
\nodeRun{helo}{readers (three)}
```

Outputs:

Hello, readers (one)!

Hello, readers (two)!

Hello, readers (three)!

2.2 Evaluating Expressions

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

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

$23 + 65 * 123 = 8018$

2.3 Spawning cURL

Using the `\exec` command, we can spawn cURL to do a remote procedure call against our NodeJS HTTP server:

```
\StrSubstitute{\CXLTXtexRoute}{/}{\%2F}[\texRoute]
\exec{curl --silent --show-error 127.0.0.1:8910/\texRoute/,/helo/friends}
```

Hello, friends!

Since this method is much faster and more versatile than using `\nodeRun`, the `\node` command is per default set to execute `\nodeCurl`; you may change that by using one of the following lines:

```
\renewcommand{\node}{\nodeRun}
% or (the default):
\renewcommand{\node}{\nodeCurl}
```

The `\nodeCurl` command (and, by extension, `\node` in its default incarnation) simplifies the above by hiding the ugly details; most of the time, you'll get away with two obligatory arguments, namely the command name and the command parameters:

```
\node{helo}{friends}
```

Hello, friends!

2.4 Character Escaping

The CXLTX 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:

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

opening brace	{
closing brace	}
Dollar sign	$
ampersand	&
hash	#
caret	^
underscore	_
wave	~
percent sign	%

2.5 Unicode

The next few examples demonstrate that Unicode characters—even ones from outside the Unicode Basic Multilingual plain, which frequently cause difficulties—can be transported to and from the server without losses or Mojibake / squiggles:

```
\nodeCurl{helo}{äöüÄÖÜß}
```

Hello, äöüÄÖÜß!

Chinese characters from the Unicode BMP ('16 bit'):

```
\nodeCurl{helo}{ 黎永強 }
```

Hello, 黎永強!

Chinese characters from the Unicode SIP ('32 bit'—these needed a little trick to make XeLaTeX choose the right font; see `coffeexelatex.sty`):

```
\nodeCurl{helo}{ 乛虫丽 }
```

Hello, 乛虫丽!

2.6 The `aux` Object

To facilitate data exchange between the TeX process and the server, CXLTX provides facilities to read and write data from and to the `aux` file assoated with the current job:

- the TeX commands `\aux`, `\auxc`, `\auxcs`, and `\auxpod`, which write to the `aux` file;
- the method `CXLTX.main.read_aux = ( handler ) ->`, which reads (and parses) data written with one of the above commands.

2.6.1 The `\aux*` Commands

Because they're quite straightforward, let's have a look at the actual definitions of the `aux*` commands:

```
\makeatletter
\catcode`\%=11
\newcommand{\aux}[1]{\immediate\write\CXLTX.auxout{#1}}
\newcommand{\auxc}[1]{\immediate\write\CXLTX.auxout{% #1}}
\newcommand{\auxcs}[1]{\immediate\write\CXLTX.auxout{% coffee #1}}
\newcommand{\auxpod}[2]{\immediate\write\CXLTX.auxout{% coffee #1: \{ #2 \}}}
\catcode`\%=14
\makeatother
```

We see our old friend `\immediate\write` here, this time accessing channel `\CXLTX.auxout`. All commands will write a single line to the `aux` file.

¶ `\aux` is the most basic command and will write text as-is to the `aux` file;

¶ `\auxc` puts whatever is written behind a `%` (percent sign), so it appears as a comment when TEX re-reads the `aux` file;

¶ `\auxcs` writes text behind a `% coffee` marker, facilitating recognition on the server side;

¶ `\auxpod` takes a name and a CoffeeScript Plain Old Dictionary literal (*without* the braces) to the `aux` file; to the server, this will become available as `CXLTX.aux[ name ]`;

The `\auxgeo` / `@\nodeCurl{show-geometry}{}` command pair is a good example how to use `\auxpod`.

2.6.2 Geometry

Use geometry data from aux file to render a table of layout dimensions into the document; note the we could have used the `\auxgeo` command anywhere in the document and that this currently only works for documents with a single, constant layout.

Also note we're using a dash instead of an underscore here—in TEX, underscores are special, so we conveniently allow dashes to make things easier. The CXLTX command `show-geometry` does not take arguments, which is why the second pair of braces has been left empty:

```
\nodeCurl{show-geometry}{}
```

firstlinev	15.00 mm
footskip	10.54 mm
headheight	4.22 mm
headsep	8.79 mm
marginparsep	3.51 mm
marginparwidth	12.30 mm
paperheight	297.00 mm
paperwidth	210.00 mm
textheight	246.36 mm
textwidth	155.00 mm
topmargin	–23.40 mm
voffset	25.40 mm

After `show-geometry` has been performed, the `CXLTX.aux` object has been populated with data from the `aux` file; it then looks like this for the current document:

```
\nodeCurl{show-aux}{}
```

```
{ 'is-complete': false,
  texroute: '/Volumes/Storage/cnd/node_modules/cxltx/doc',
  auxroute: '/Volumes/Storage/cnd/node_modules/cxltx/doc/cxltx-manual.auxcopy',
  jobname: 'cxltx-manual',
  splitter: ',',
  'method-name': 'show_aux',
```

```
parameters: [],
labels:
 { 'coffeexelatex-cxltx':
    { name: 'coffeexelatex-cxltx',
      ref: '1',
      pageref: '3',
      title: 'CoffeeXeLaTeX (CXLTX)',
      'is-duplicate': false },
   'what-is-it-and-why':
    { name: 'what-is-it-and-why',
      ref: '1.1',
      pageref: '3',
      title: 'What is it? And Why?',
      'is-duplicate': false },
   'tex-and-nodejs':
    { name: 'tex-and-nodejs',
      ref: '1.2',
      pageref: '4',
      title: 'TeX and NodeJS',
      'is-duplicate': false },
   'the-command-line-remote-command-interface-clrci':
    { name: 'the-command-line-remote-command-interface-clrci',
      ref: '1.2.1',
      pageref: '4',
      title: 'The Command Line Remote Command Interface (CL/RCI)',
      'is-duplicate': false },
   'the-http-remote-command-interface-hrci':
    { name: 'the-http-remote-command-interface-hrci',
      ref: '1.2.2',
      pageref: '6',
      title: 'The HTTP Remote Command Interface (H/RCI)',
      'is-duplicate': false },
   'security-considerations':
    { name: 'security-considerations',
      ref: '1.2.3',
      pageref: '6',
      title: 'Security Considerations',
      'is-duplicate': false },
   'sample-command-lines':
    { name: 'sample-command-lines',
      ref: '1.3',
      pageref: '7',
      title: 'Sample Command Lines',
      'is-duplicate': false },
   'useful-links':
    { name: 'useful-links',
      ref: '1.4',
      pageref: '7',
      title: 'Useful Links',
      'is-duplicate': false },
   'related-work':
    { name: 'related-work',
      ref: '1.5',
      pageref: '7',
      title: 'Related Work',
      'is-duplicate': false },
   examples:
    { name: 'examples',
      ref: '2',
      pageref: '9',
      title: 'Examples',
```

```
   'is-duplicate': false },
spawningnodejs:
 { name: 'spawningnodejs',
   ref: '2.1',
   pageref: '9',
   title: 'Spawning NodeJS',
   'is-duplicate': false },
evalcs:
 { name: 'evalcs',
   ref: '2.2',
   pageref: '9',
   title: 'Evaluating Expressions',
   'is-duplicate': false },
spawningcurl:
 { name: 'spawningcurl',
   ref: '2.3',
   pageref: '10',
   title: 'Spawning cURL',
   'is-duplicate': false },
esc:
 { name: 'esc',
   ref: '2.4',
   pageref: '10',
   title: 'Character Escaping',
   'is-duplicate': false },
unicode:
 { name: 'unicode',
   ref: '2.5',
   pageref: '11',
   title: 'Unicode',
   'is-duplicate': false },
geo:
 { name: 'geo',
   ref: '2.6.2',
   pageref: '12',
   title: 'Geometry',
   'is-duplicate': false },
labels:
 { name: 'labels',
   ref: '2.6.3',
   pageref: '15',
   title: 'Labels',
   'is-duplicate': false },
'error-detection-works':
 { name: 'error-detection-works',
   ref: '2.6.3',
   pageref: '16',
   title: 'Duplicate Labels (2)',
   'is-duplicate': true },
functions:
 { name: 'functions',
   ref: '2.7',
   pageref: '16',
   title: 'Implementing and Calling Functions',
   'is-duplicate': false },
epilogue:
 { name: 'epilogue',
   ref: '3',
   pageref: '18',
   title: 'Epilogue',
   'is-duplicate': false },
```

```
    outline:
     { name: 'outline',
       ref: '4',
       pageref: '23',
       title: 'Project Implementation Outline',
       'is-duplicate': false } },
  'duplicate-labels':
   { 'error-detection-works':
      [ { name: 'error-detection-works',
          ref: '2.6.3',
          pageref: '15',
          title: 'Duplicate Labels (1)',
          'is-duplicate': true },
        { name: 'error-detection-works',
          ref: '2.6.3',
          pageref: '16',
          title: 'Duplicate Labels (2)',
          'is-duplicate': true } ] },
  geometry:
   { paperwidth: 210.0001,
     paperheight: 297,
     textwidth: 155.0001,
     textheight: 246.3597,
     headheight: 4.2176,
     headsep: 8.7865,
     footskip: 10.5438,
     marginparsep: 3.5146,
     marginparwidth: 12.3011,
     voffset: 25.4,
     topmargin: -23.404,
     firstlinev: 15.0001 } }
```

2.6.3 Labels

As it stands, CXLTX will try and collect all pertinent data from the `CXLTX.aux` file when `@read_aux` is called; this currently includes the namew of the labels, their 'pageref' and 'ref' attributes as well (thanks to the hyperref package) the respective associated titles.

Remember that you will have to re-run XELATEX whenever there have been changes to your labels as CXLTX can only read from the (copy of the) `aux` file of the previous XELATEX invocation, and LATEX itself needs more than one pass in many cases, too.

You can easily have CXLTX print you out a table with an overview of the currently defined labels, but observe that `show-labels` is currently implemented in a very simple-minded fashion, meaning that it will make no adjustments to paper size or breaking the table across several pages. You may want to have a look at the source of `sample-provider.coffee` to get an idea of how to use the data provided for your own purposes.

```
\paragraph{Duplicate Labels (1)}\label{error-detection-works}
```

Duplicate Labels (1) One immediate benefit you can generate with CXLTX is to make it hunt for duplicate lables. TEX source is not expecially convenient to write and macros can be hard to get

right, so it's sort of a survival strategy to do a lot of copy-and-pasting—which may inadvertently lead to duplicate labels.

```
\paragraph{Duplicate Labels (2)}\label{error-detection-works}
```

Duplicate Labels (2) Now LaTeX is nice enough to warn you *that* duplicate labels have occurred, but not bold enough to tell you *which* labels were affected—all you get is a cursory `LaTeX Warning: There were multiply-defined labels`. It could do that as the data is plain to see in the `aux` file, but it doesn't do that for you.

Here is an overview of the labels in the current document; you'll immediately notice those duplicate labels since they are always put at the very top of the table, so no matter how wide or long the table gets, you'll get to see them right away in the output:

```
\nodeCurl{clear-aux}{}
\nodeCurl{show-aux}{}
```

```
{ 'is-complete': false,
  texroute: '/Volumes/Storage/cnd/node_modules/cxltx/doc',
  auxroute: '/Volumes/Storage/cnd/node_modules/cxltx/doc/cxltx-manual.auxcopy',
  jobname: 'cxltx-manual',
  splitter: ',',
  'method-name': 'show_aux',
  parameters: [] }
```

2.7 Implementing and Calling Functions

It is simple to define own functions for CXLTX; let's have a look on a particularly easy one:

```
% in sample-provider.coffee:

@add = ( P..., handler ) ->
  P[ idx ] = parseFloat p, 10 for p, idx in P
  R   = 0
  R  += p for p in P
  return handler new Error "unable to sum up #{rpr P}" unless isFinite R
  handler null, R

% in examples.tex:

\node{add}{2,3,5,8}
```

When we run this code, we get back 18, as expected. The three dots in the function signature, `P...`, are a signal that the function expects any number of arguments; the last parameter, `handler`, is the callback function that each CXLTX provider method must call upon error or completion.

```
{\renewcommand{\CXLTXparameterSplitter}{!}
\node{add}{2!3!5!8}}
```

18

Table 1: Output of `show-labels`

	name	pageref	ref	title
!	error-detection-works	15	2.6.3	Duplicate Labels (1)
!	error-detection-works	16	2.6.3	Duplicate Labels (2)
	coffeexelatex-cxltx	3	1	CoffeeXeLaTeX (CXLTX)
	what-is-it-and-why	3	1.1	What is it? And Why?
	tex-and-nodejs	4	1.2	TeX and NodeJS
	the-command-line-remote-command-interface-clrci	4	1.2.1	The Command Line Remote Command Interface (CL/RCI)
	the-http-remote-command-interface-hrci	6	1.2.2	The HTTP Remote Command Interface (H/RCI)
	security-considerations	6	1.2.3	Security Considerations
	sample-command-lines	7	1.3	Sample Command Lines
	useful-links	7	1.4	Useful Links
	related-work	7	1.5	Related Work
	examples	9	2	Examples
	spawningnodejs	9	2.1	Spawning NodeJS
	evalcs	9	2.2	Evaluating Expressions
	spawningcurl	10	2.3	Spawning cURL
	esc	10	2.4	Character Escaping
	unicode	11	2.5	Unicode
	geo	12	2.6.2	Geometry
	labels	15	2.6.3	Labels
	functions	16	2.7	Implementing and Calling Functions
	epilogue	18	3	Epilogue
	outline	23	4	Project Implementation Outline

3 Epilogue

> TEX was designed for typesetting, not for programming; so it is at best "weird" when considered as a programming language.[1]

We all know that all's well that ends well, *but the astute reader will have noticed that through the lines of this manual there is a certain amount of despair shining through.*

There is a somewhat icky metaphor popular among TEXnicians which compares the inner workings of TEX to the digestive work done by the mouth and the bowels, a metaphor that never fails to leave me both unimpressed and uninformed. Kind souls answering the question of neophytes and noobies often draw on this picture to explicate why some `\foo{\bar}` *invocation miserably flopped where the seemingly equally innocuous* `\foo{\baz}` *succeeded, causing TEX to spit out unhelpfully obfuscated lines, often ones spewn with @-signs, all decorated with a mysterious-slash-nonsensical error message and a line number that quite frequently does* not *refer to the file currently being processed, before offering the user the choice to either give up or else give up.*[2] *As if that wasn't bad enough, those kind people that hang out on discussion websites will likely start talking about how TEX chews, digests, ruminates, and, well, sometimes regurgitates its inputs when you go and ask them for help.*

TEX's behavior must be classified as singularly unhelpful and irritating; you might want to say that TEX cannot be successfully run without the Internet, for everything save the shortest and most boring documents will, with a chance of near-certainty, have some *part—maybe something like a* `\usepackage{x}` *statement that was inadvertently placed* before *another, superficially unrelated* `\usepackage{y}` *instead of* after *it—that makes TEX screw up.*

Well-meaning fellow TEX-users will tell you to read The TEX Book,[3] *but in case you can count you will quickly see that—despite its purported reputation of being a fine manual—it actually is more a challenge to read than anything else, and i'm specifically referring here to the fact that it has, on the roughly 300 pages of the main part, no less than 284 exercises (of which only 71 are 'plain' exercises; 91 are labelled 'dangerous' and 122 'ddangerous'). To be fair, answers are given. Alas, when you peruse* Appendix I: Index, *it'll make you think when you realize that there plainly is no keyword of interest with less than, say, three or four, often six or ten page numbers next to it, meaning that to look up anything at all you'll always have to sort of embark on a small grand tour. Add to that the author's penchant for a witty style that often gets lost in obscure details*[4] *before getting off at a tangent and you'll more often than not rather inquire about a problem on the Net right away rather than consult* The Book.

---

[1] [Knu99], p235

[2] reminds one of the now-classic Last Words Dialog featured in early versions of Microsoft Word (which in essence said 'Something went bad' and offered the choice to either 'ignore', 'retry' or 'abort', none of which buttons did react to mouse clicking)

[3] see [Knu90]

[4] there are almost 30 pages dedicated to 'Dirty Tricks'; that chapter is guarded with an eleven-fold 'Danger!'-sign

Maybe the best thing that can be said about TEX is that 'it is good at typesetting', which, ultimately, is of course why so many people are ready to put up with all that TEX is not so good at. Which is a lot, to say the least. Like, adding numbers. Or, doing string processing. Or, doing any one of the fundamental things that distinguish a markup language from a programming language: naming, looping, branching, function building, and sensible data types (minimalists will claim much less is needed for Turing-completeness, but practioners know that with much fewer tools, doing things will become an inordinately arduous task).[5]

When i say functions, i mean functions, that is,

> optionally named bundles of functionality which accept certain classes of inputs (as arguments) and yield outputs (as return values) with or without having so-called 'side effects'.[6]

Noticeably absent from this definition is the ability to do the one thing when called in one environment, and the other when called from within another environment—which is what too many TEX macros actually do. In a Real programming language, you do not want a function `f = ( x ) -> ...` *to return 42 when called as* `5 / ( f 3 )` *but 71 when called as* `( f 3 ) / 5` *just because* `f` *happened to have been situated once in the divisor and once in the dividend position of an expression; that would be more than weird. There sure* may *be cases where such a behavior makes sense, but those are few and far between.*[7] *In particular, given the knowledge that* `f 3` *returns some number, you do not want that same call* `f 3` *to blow up the VM with an error message when used in the context of, say, a multiplication instead of a division.*[8]

TEX macros frequently do exactly that, and while LATEX sure is a feat of a great thinker and many ingenious and diligent minds have struggled to come up with useful packages to make typesetting with TEX easier, presumably few have succeeded in providing 'real' functions in the sense outlined above; in my experience, there's always a good chance that a given command will break whenever used outside of the select environments it has been tested in / was intended for.

As a case in point, consider the monospaced words appearing in some of the section titles of the

---

[5] It can at times be bewildering to see people 'doing it the hard way' when easier ways are readily available, and why? —'just because they can'. While mental training is certainly a laudable way to spend time, i cannot help but feeling that TEX has a way to waste mine, for the precise reason that for all the effort i put into mastering it i get back so little in terms of re-usable knowledge. In a similar way, many in the field of CS are ready to jump from 'it's Turing-complete' to 'you can and should use it to write programs with'. Hell, even CSS+HTML and certain card games may be considered Turing-complete (see http://beza1e1.tuxen.de/articles/accidentally_turing_complete.html, http://boingboing.net/ 2012/09/12/ magic-the-gathering.html). For a short summary of my view on the topic, see http://programmers.stackexchange.com/a/223933/114502 (i received downvotes because i deliberately mistook the 'minimal' in the OP's question as implying 'for practical purposes'. Formally, my answer is, therefore, wrong.)

[6] see https://en.wikipedia.org/wiki/Side_effect_(computer_science); essentially: "a function is said to have a side effect if [...] it also modifies some state or has an observable interaction with [...] the outside world. For example, a function might [...] modify one of its arguments, raise an exception, write data to a display or file [...]." In practical computing, side effects are important because we want to 'do something' (change state) and 'produce something' (namely output).

[7] One exception to the rule is the output of many command line tools that will use colors when writing to the monitor, but omit colors when writing to a file, as most viewers will fail to render colors and instead display unsightly control codes.

[8] except when an overflow occurs

document you're now reading, like 'The `aux` *Object'. In most cases where the monospaced font appears, the* `\verb#foo#` *command has been used (in fact, the last instance of monospaced text appears as* `\verb!\verb#foo#!` *in the source; now you know what style* The TEX Book *is written in). But* `\verb#x#` *happens to break when used in the argument to a* `\subsection{}` *command:*

```
\subsection{The \verb#aux# Object}
```

The above just leads to tears:

```
Illegal parameter number in definition of \GetTitleStringResult.
```

Incidentally, you can't use `\verb` *in a* `\footnote{}`*, either, but this time you get a completely unrelated and equally opaque message:*

```
You can't use `macro parameter character #' in horizontal mode.
```

I have no idea why there are quotes in this phrase, but maybe we've hit upon something here: Both messages share the word 'parameter', and indeed, `#` *is TEX's way to mark parameters.*[9]

OK great, not sure why that snazzy `#` *works in a plain paragraph but not in arguments to (some? all?) commands, but, hey, we sure can use another character in its place—after all,* `\verb#x#` *can be rewritten as* `\verb&x&` *or* `\verbXxX` *(or with any other hedge character, so we can always find one that does not interfere with the text meant to be displayed), so, sure enough!, we can drop that problematic* `#` *and swap it for something else, can we?*

```
\footnote{The \verb+aux+ Object}
```

Nope. Gets you this:

```
Missing } inserted.
```

Ah well. Let's try that with `\subsection` *then:*

```
\subsection{The \verb+aux+ Object}
```

Nope. Gets you this:

```
LaTeX Error: \verb illegal in command argument.
```

I tried this with several hedge characters but to no avail. Needless to say the kind people that hang out on websites dedicated to solving problems with TEX do provide solutions. Also needless to say that the respective solutions suggested for `\footnote` *and* `\subsection` *that i saw are rather orthogonal to each other—apparently, the observed breakages are due to different aspects of the underlying code that define* `\footnote` *and* `\subsection`*.*

Let's face it, programming is hard. Programming gets even harder when you have to work against the grain with every step you take. Programming in TEX is especially hard because of the insanely high

[9] in its own in-imitable ways: `#1` refers to the first argument to a macro, but `##1` must be used inside another macro that is nested inside that first macro to refer back to the same argument. In case you should care: you'd use `####1` to refer to the same argument if you were to try and define a macro nested inside a nested macro, *not* `###1`.

degree of complectedness[10] *the language features. What you do to manage the considerable overhead that comes with a system of this complexity is using so-called 'design patterns', that is, bluntly, 'coping strategies for programmers', and, even more bluntly, 'copying strategies'.*

Programming TEX, it would seem, is really only manageable (to a degree) by doing nothing but *applying design patterns, precisely because the language is only marginally more ergonomic than, say, INTERCAL.*[11]

In fact, it has been said that design patterns—while certainly being helpful in that they show people how to tackle classes of problems—are really 'anti-patterns': signs that a given language is poorly suited for a given way of doing things.

Art Atwood puts it this way:

> If you find yourself frequently writing a bunch of boilerplate design pattern code to deal with a "recurring design problem", that's not good engineering– it's a sign that your language is fundamentally broken. [...] [design patterns] turn the programmer into a fancy macro processor.[12]

One comment i read stated that its writer could only come to grips with TEX by way of 'uttering incantations' (i.e. copying black-box solutions found on the Internet and keeping fingers crossed they'll just work); in other words, people are—by the very design of the environment they're put into—encouraged to resort to that malicious, bad habit that is Cargo Cult Programming (which in this world must be having more adherents than its close cousin, Object Oriented Programming).

Donald Knuth, famous inventor of TEX, writes, on page 400 in the aforementioned chapter 'Dirty Tricks', that "[i]t would be possible to write an entire book about TEX ouput routines"; this is probably more than a mild understatement in the face of comments like these:[13]

```
% We empty any left over kludge insert box here; this is a temporary fix.
% It should perhaps be applied to one page of cleared floats, but
% who cares?  The whole of this stuff needs completely redoing for
% many such reasons.
```

which anyone with a LATEXinstallation can read on their own machine by opening `ltoutput.dtx` *in their text editor. These words (alongside with other delicious commentaries left by members of the LATEX dev team) are both testament to the inner complexity of TEX and the limited capabilities of us frail humans, beings who can only juggle with so many balls and hold on to only so many loose ends at any given time.*

Let us pinpoint a few observations on what makes TEX so hard.

---

[10] watch Rich Hickey on http://www.infoq.com/presentations/Simple-Made-Easy

[11] see https://en.wikipedia.org/wiki/INTERCAL_programming_language

[12] see http://www.codinghorror.com/blog/2007/07/rethinking-design-patterns.html

[13] recommended reading: http://tex.stackexchange.com/a/8736/28067; this answer contains links to a four-part article series in the magazine *TUGboat* which may or may not substantiate the implicit claim made here (that some parts of TEX are too hard even for the most dedicated).

XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX

horrible error messages that drown in a sea of largely inconsequential informative output and occasional warnings:

```
! Missing control sequence inserted.
<inserted text>
                inaccessible
l.70 wrong.)}
```

no namespacing

catcodes (active characters would have sufficed)

macro expansion, `\expandafter`, `\noexpand`

dependency of outcome / success of macros on environment they are called from

some characters such as `_` *(underscore) make only ever sense in math mode—so why does TEX complain when used* outside *of math mode? There's even a specific catcode assigned to the underscore, i.e. one out of 16 catcodes is specifically there to identify the role of the underscore in math mode (which makes subscripts). Wouldn't a more general solution have been more beneficial here? Active characters, anyone?*

no simple formulation of looping and branching

unobvious procedures; e.g. googling for latex references with section number toc *gives me:*

> For standard classes (article, book, report), adding
> `\usepackage[nottoc,numbib]{tocbibind}`
> to your document preamble should work.

What kind of language is this where mumbling 'nottoc numbib tocbibind' makes the title of the References section magically appear with a section number and in the table of contents? How is that better than, say, 'hax pax max deus adimax'? If it wasn't for the Internet, where would i learn this stuff? on Hogwart High?

4 Project Implementation Outline

- `cxltx/` — application folder
  - `cxltx.sty` — LaTeX module to enable CXLTX
  - `src/` — CXLTX CoffeeScript sources
    - `cli.coffee` — provides command line RPC interface
    - `main.coffee` — CXLTX backend; mainly an RPC dispatcher
    - `sample-provider.coffee` — sample functionalities for CXLTX RPC
    - `ids-provider.coffee` — dto.
    - `server.coffee` — provides HTTP RPC interface
  - `cxltx-manual.pdf` — this manual
  - `doc/` — source for this manual
    - `cxltx-manual.tex` — documentation master file
    - `README.tex` — TEXified version of `README.md`; input to manual
    - `intro.tex` — input to manual
    - `conventions.tex` — dto.
    - `examples.tex` — dto.
    - `epilogue.tex` — dto.
    - `outline.tex` — dto.
    - `cxltx-manual.auxcopy` — auxiliary file produced by CXLTX
    - `cxltx-manual.aux` — temporary TEX-file
    - `cxltx-manual.toc` — dto.
    - `cxltx-manual.bbl` — dto.
    - `cxltx-manual.bib` — dto.
    - `cxltx-manual.blg` — dto.
    - `cxltx-manual.fls` — dto.
    - `cxltx-manual.log` — dto.
    - `cxltx-manual.out` — dto.
  - `lib/` — contents of `src` transpiled to JavaScript
  - `node_modules/` — JS libraries
  - `monitor-options.json` — configuration of server monitor; enables restart on source change
  - `package.json` — Common JS / npm package configuration
  - `README.md` — source of section 1 of the present manual
  - `start-monitored-server` — Bash script to start CXLTX HTTP RPC server

5 References

[Knu90] Donald E. Knuth. *The TEXbook*, volume A of *Computers & Typesetting*. Addison–Wesley, Reading, MA, 1990. 10th printing.

[Knu99] Donald E. Knuth. Mini-indexes for literate programming. In *Digital Typography*, pages 225–245. CSLI Publications, Stanford, 1999.