The Tectonic Project: Envisioning a 21st-century TEX experience

Peter K. G. Williams

Abstract

Tectonic is a software project built around an alternative TEX engine forked from XETEX. It has been created to explore the answers to two questions. The first question relates to documents: in an era of 21st-century technologies — where interactive displays, computation, and internet connectivity are generally cheap and ubiquitous — what new forms of technical document have become possible? The second question relates to tools: how can we use those same technologies to better empower people to create excellent technical documents?

The premise of the Tectonic project is that while TEX may be venerable, it is still an ideal system for creating "21st-century" technical documents — but that a project with an independent identity and infrastructure can make progress in ways that can't happen in mainline TEX. Tectonic is compiled using standard Rust tools, installs as a single executable file, and downloads support files from a prebuilt TEX Live distribution on demand.

In the past year, long-threatened work on native HTML output has finally started landing, including a possibly novel Unicode math rendering scheme inspired by font subsetting. Current efforts are fleshing out this HTML support using *XETEX: The Program* as a test case, with an eye towards substantially improving the documentation of Tectonic itself. While Tectonic positions itself as "outside of" traditional TEX in a certain sense, the project could not exist without the efforts of the entire TEX community, to whom the author and the project are gratefully indebted.

1 Introduction

This article will motivate the Tectonic project (§2), discuss some of its distinctive characteristics (§3), delve into how it is implementing HTML output (§4), and briefly discuss the outlook for its future (§5).

2 Motivation

I (PKGW) will motivate the Tectonic project with a somewhat stylized history of my journey through the TEX ecosystem. My background is in scientific research (astronomy), and to the best of my recollection, my first use of TEX was for typesetting problem sets in college. I still remember the satisfaction of creating a beautifully typeset equation, and understanding that there was no other tool in the world

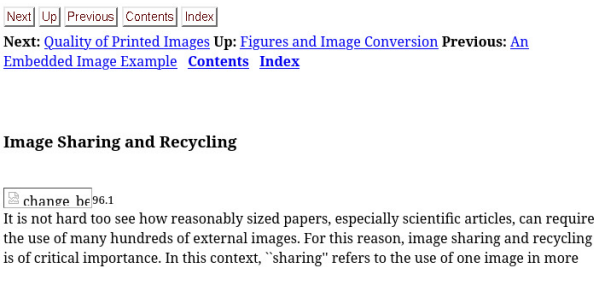
Next Up Previous Contents Index
Next: Quality of Printed Images **Up:** Figures and Image Conversion **Previous:** An Embedded Image Example **Contents** **Index**

Image Sharing and Recycling

change_be96.1
It is not hard too see how reasonably sized papers, especially scientific articles, can require the use of many hundreds of external images. For this reason, image sharing and recycling is of critical importance. In this context, ``sharing'' refers to the use of one image in more

Figure 1: A screenshot of typical LATEX2HTML output. From `www.sci.utah.edu/~macleod/latex/latex2html/Enode8.html`, chosen arbitrarily.

that could typeset math nearly as well — at least, none that could be freely used by a college student.

During my college career (2002–2006, if you must ask), it was clear that the Internet and World Wide Web were on their way to transforming society. But for the most part, design on the web was notoriously poor. LATEX could be converted to HTML and rendered, but the results resembled Figure 1: legible, mostly, but absolutely inferior to what could be accomplished in PDF. And while HTML documents had hypertext capabilities, they were generally *static* documents: words and figures arranged on a page in a facsimile of the "real thing": ink on paper.

For me, there were two major "watershed moments" demonstrating that web documents weren't always going to be inferior. First, the release of Google Maps (February, 2005; `maps.google.com`) showed that websites could be *applications*, not just static documents. In my mind, this opened up exciting possibilities for new forms of scientific and technical communication: not just hypertextual facsimiles of paper, but *interactive* documents.[1] More broadly, I'll define *21st-century documents* as those that leverage the technologies that have been unrolling since then: documents designed for a world where interactive digital displays, computation, and internet connectivity are often cheap and ubiquitous. (I don't love this terminology — smacks of naïve futurism — but don't have anything better.) In principle, 21st-century documents can target any of a variety of technology platforms, but in my opinion, the web is the only one that matters. Web content can be experienced nearly anywhere, from smartphones to billboards, and private industry is spending *billions* of dollars every year to enhance its power and reach. Nothing else comes close.

[1] See the slides to my talk, which are in HTML, for an example of an embedded interactive plot (`tug.org/tug2022/assets/html/Peter_K_G_Williams-TUG2022-slides/`). Because *TUGboat* is delivered in PDF format, I can't reproduce the plot here.

doi.org/10.47397/tb/43-2/tb134williams-tectonic

But in 2005, the state of web typography was still pretty poor, and so PDF was still easily the best choice for scientific papers and other kinds of *technical documents*. While I won't attempt to define this category precisely, common characteristics of technical documents include substantial length; use of mathematics, figures, or tables; complex structure; dense internal or external referencing; and more recently, integration with source code and computation. While every kind of document deserves excellent typography, I will assert that technical documents probably suffer *more* from bad typography than non-technical ones. The second watershed moment for me was therefore the release of the pdf.js library for displaying PDFs in the web browser (July, 2011; `mozilla.github.io/pdf.js/`). What better way to demonstrate that you can do high-precision typography on the web than by demonstrating that you can render arbitrary PDF files?

After seeing pdf.js, I was convinced that all of the pieces were in place to start trying to bring the typographic quality of TEX to the world of web-native, 21st-century documents. Even attempting this would surely require new software to be created — but alongside my scientific training, I've been involved in open-source software development since even before I started college, and I'm more than happy to tackle such problems myself.

My first push, undertaken around 2014, was an attempt to do a clean-room implementation of the core TEX engine in JavaScript, using *The TEXbook* as a reference. It went about as well as you might expect. I made decent progress, but quickly came to understand that the TEX engine itself is just the tip of the proverbial iceberg of code needed to compile actual modern LATEX documents, and that *The TEXbook* is only a partial — in fact, sometimes misleading — guide to how modern TEX engines operate, with no discussion of ε-TEX, Unicode, OpenType fonts, and more. I concluded that in order to compile "real" LATEX documents for the web, one would need to build on "real" LATEX.

So I started to look into hacking TEX. More specifically, I looked into modifying the XETEX engine, since it includes support for Unicode and OpenType fonts, which struck me as essential for creating truly web-native documents. As a person with a long history in open-source projects, the experience was frankly frustrating and discouraging. Even the traditional first step for getting to understand a codebase — checking out the source code from version control — felt like an ordeal, primarily due to a lack of clarity about which repository to use, the huge size and deeply nested structure of the TEX Live repository, and the use of Subversion. Modern software development conveniences, above all the availability of some kind of GitHub-like "pull request" mechanism and continuous integration (CI), were missing.

While there's no shortage of fads in the world of software development, there have been some real advances, and as a developer I find them extremely important. For me, undertaking a software project without them is like trying to write a research paper in Microsoft Word rather than LATEX. I can do it if I have to, but I won't enjoy it, and the inferior tools close off entire ways of working that I don't want to give up. I didn't feel that I could work with the TEX code in the way that I wanted to, if I was forced to use the existing infrastructure.

But if you take a software project and rebuild its development infrastructure, it is unlikely to be feasible to merge your changes back into the original source. Such changes result in a long-lived *fork*, not a temporary *branch*. Forking a project is a weighty decision, not to be taken lightly. But as I thought about the experiments I wanted to try, I came to believe that forking was an appropriate path. Besides allowing me to explore newer development tools, it would allow me to explore a new "persona" for the project — a distinct brand identity. This may sound like business jargon, because it is, but it captures the right concept. I wanted the ability to try *all sorts* of things that you couldn't do with traditional TEX: tidy up the output, change default behaviors, drop compatibility with various ancient packages. It wouldn't be right to describe such a system as a regular TEX system. New branding gives an opportunity to reset user expectations all at once, without having to explain the details of individual technical changes.

3 The Tectonic Project

The narrative of the previous section has suggested an interrelated set of gaps in the TEX ecosystem:

- support for creating modern HTML output with a full-featured TEX engine;
- a modernized developer experience;
- a modernized user experience; and
- a project with a distinct brand identity to serve as a platform for experimentation.

Launched in 2016, the Tectonic project aims to fill these gaps. Key elements of its design are as follows.

3.1 Form factor

Tectonic is delivered as a single executable, named tectonic, that bundles the capabilities of XETEX, bibtex, xdvipdfmx, and supporting machinery for driving

these *engines*. The executable is designed to be as self-contained as possible, with minimal dependencies on system libraries, environment variables, user configuration files, or external tools. In particular, dependencies on Ghostscript have been removed for security, eliminating PostScript capabilities.

3.2 Engine implementation

The engines, most notably XeTeX, are implemented with C/C++ code obtained from the standard WEB2C pipeline implemented by TeX Live. The C/C++ files extracted from the pipeline have been extensively reformatted and refactored to make them more human-readable and, for instance, reintroduce symbolic constants that do not survive the WEB2C workflow. A few refactorings have been conducted automatically with coccinelle [2]. Due to these customizations, however, engine updates from TeX Live cannot be automatically incorporated into the Tectonic codebase. This is the price of forking. To aid the process of synchronizing Tectonic with TeX Live, a framework called tectonic-staging (code repository at `github.com/tectonic-typesetting/tectonic-staging/`) contains a pipeline that can automatically generate a readable set of C/C++ "reference sources" from the TeX Live repository. When updates from a new TeX Live release are to be incorporated into Tectonic, the pipeline is run and changes to the reference sources are manually imported into Tectonic's codebase. This system heavily leverages the change-tracking features of the git version control system.

3.3 Use of Rust

Excepting the engines, Tectonic is implemented in the Rust language. Rust is a systems-level language focusing on performance, reliability, and productivity. Rust's packaging and compilation model is an excellent fit for a project like Tectonic: while Rust offers a sophisticated package ecosystem that makes it easy to import support for anything from the HTTPS protocol to image loading, it compiles by default into self-contained executables that lack external dependencies. Rust also has excellent support for cross-platform work and bridging with C and C++ code. Rust's packaging tool, cargo, allows codebases to be organized into "crates" with well-defined interfaces, and has a "feature" system for the management of build options. The main Tectonic codebase currently consists of 22 crates.

More broadly, the Rust language has a similar spirit to TeX. Both are regarded as best-in-class tools that can be demanding, but rewarding as well. Both have a reputation for being complicated and hard to learn. Despite this reputation, Rust has achieved a tremendous level of success in a relatively short period of time, being named the "most loved language" by `stackexchange.com` for seven consecutive years as of this writing. Rust supporters generally attribute this success to several factors. First, Rust is technically excellent: it actually delivers on its promises in a rigorous way, and third-party Rust packages are often well-designed, performant, and reliable. Second, Rust has excellent tooling, with high-quality built-in support for package management (cargo), documentation, testing, and more. Third, the Rust user community explicitly values being welcoming and inclusive. Many aspects of the Rust design aim to support new users, most famously the Rust compiler's error messages, which generally offer impressively clear diagnoses of problems and useful advice for fixing them. Spanning these factors is a theme of *experience-centered design*: elements of the Rust ecosystem are designed primarily around a vision of what it will be like for people to use them, with technical goals flowing from that vision. Tectonic explicitly aims to emulate these characteristics of the Rust ecosystem and community.

3.4 Bundles

Tectonic can be delivered as a single executable because it can download files from a backing TeX distribution on the fly, during document compilation. This functionality is implemented by virtualizing the I/O subsystem underlying the engines so that it can search not just the local filesystem, but also remote "bundles", for files. Files from bundles are cached locally, and the implementation is designed such that the network is needed only if a new file must be fetched. Bundles are created using a reproducible, automated pipeline based on the TeX Live installation process (`github.com/tectonic-typesetting/tectonic-texlive-bundles/`). The bundle file as served over the network is essentially a large Unix tar file with an associated index, which the tectonic program downloads in pieces using HTTPS byte-range requests.

The bundle scheme is also the backbone of Tectonic's approach to reproducible document builds. Bundle files are intended to be immutable, and it is possible to associate a given Tectonic document (see below) with a specific bundle, identified by its url or a SHA256 cryptographic digest based on its contents. It is thus possible to specify the exact TeX distribution that a document should be built against. There is an associated loss of flexibility: to update or extend a package contained in the bundle, you must generate your own bundle or install the package files locally, currently on a per-document basis.

3.5 Document model

Tectonic offers a "document model" for defining compilations. Its design is heavily indebted to that of Rust's cargo tool. Tectonic documents are directory structures indicated by the existence of a file named Tectonic.toml at the root. This file, in the TOML structured-data format (`toml.io`), declares basic metadata about a document and how it should be built. By default, the top-level document source code is contained in a subdirectory named src in files named _preamble.tex, index.tex, and _postamble.tex, that are processed in that order.

A document can be built by running the command tectonic -X build anywhere in its source tree. This will create one or more versions of the document in a build directory below the Tectonic.toml file. The -X flag marks the use of Tectonic's "version 2" command-line interface, which uses a "subcommand" or "multi-tool" paradigm like git or svn. For compatibility, the default mode of operation is still "version 1": tectonic myfile.tex will compile the specified input file without invoking the document model. This form of one-shot compilation is accessible in the version 2 interface with tectonic -X compile myfile.tex. The version 1 interface will eventually be deprecated and the -X flag will become optional.

The main purpose of the Tectonic document model is to make document builds automatable, reproducible, and analyzable by rendering document-specific choices as configuration, rather than (e.g.) a string of command-line options. For instance, the Tectonic.toml file can record what TEX format file a build requires, or whether it needs shell-escape functionality. (A runtime flag can override this setting when the document to be built is not from a trusted source.) As alluded to above, the specification can define multiple outputs for a single document, such as PDFs in both US Letter and A4 sizes.

While the document model has not yet been developed thoroughly in Tectonic, it is expected to provide a platform for additional utilities in the future. For instance, a future tectonic -X format command might automatically reformat a document's sources into a consistent style, or tectonic -X doc might generate meta-documentation about available control sequences customized to a particular document's selection of packages.

4 HTML output

Although high-quality HTML output has been a goal of the Tectonic project from its inception, little work has happened on this front — until this year. Interesting progress has begun to occur.

The overall approach taken while implementing HTML output for Tectonic has been to focus on achieving high-quality results with documents that specifically target that output format. While the eventual goal is to be able to produce good HTML output from arbitrary input documents, that is a larger problem that is being avoided for the time being. Current efforts also prioritize visual appearance over proper semantic tagging, and focus on the English language.

As a broad approach, when HTML output is called for, a special flag in the XeTeX engine is activated that alters various aspects of its behavior. Linebreaking of paragraphs is disabled to avoid dealing with hyphenation, and `\specials` are inserted to indicate engine-suggested locations for insertions of HTML tags such as `<p>`. The resulting output file is essentially in the XeTeX XDV format, but it is relabeled as a new "SPX" format to avoid confusion. (SPX stands for "semantically-paginated XDV", but it is a misnomer because semantic pagination turned out to be technically infeasible.)

A new processing step written in Rust, spx2html, uses the Tera templating framework (`tera.netlify.app`) to convert the single SPX file into one or more HTML outputs, and creates or copies associated files such as CSS stylesheets, JavaScript user interface code, and font files. The spx2html stage is designed under the assumption that the input document uses OpenType fonts everywhere, including mathematics, via the unicode-math package. This dramatically reduces the problem space by allowing the code to only work with fonts that can be rendered directly by the browser.

4.1 Precise typography in canvases

Initial HTML work focused on demonstrating the precise character sizing and positioning needed to render constructs such as "TEX". In Tectonic, the bulk of document text is emitted directly into the HTML, but areas needing careful typographic layout are handled specially as *canvases*. Layout in the canvas mode can either be activated automatically by the engine (for instance, in math mode) or manually by the author (with `\specials`).

Because CSS commands can be used to move individual HTML elements arbitrarily, the actual positioning is not difficult, although some care needs to be taken to achieve proper alignment relative to the text baseline for inline expressions. More challenging is the fact that the SPX file specifies how *glyphs* in a font should be placed, while the HTML output must be Unicode text — and these are distinct concepts. In many cases, there is a direct mapping

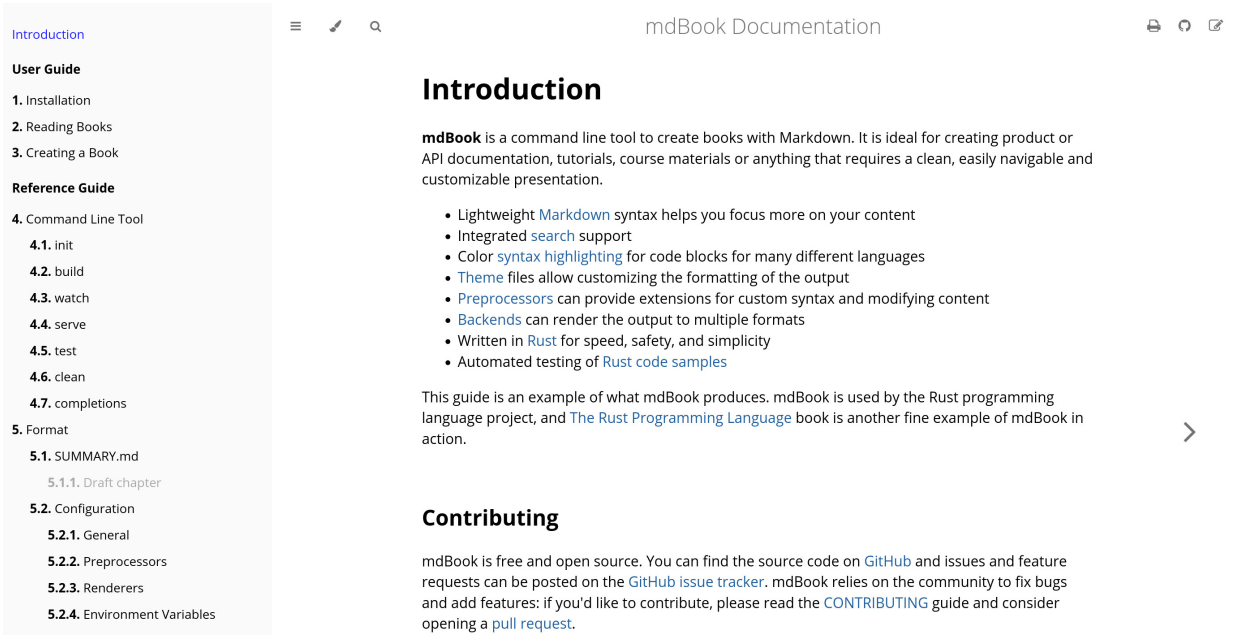

Figure 2: The standard mdBook layout, discussed in subsection 4.2. From `rust-lang.github.io/mdBook/`.

between the two, encoded in a font's Unicode CMAP table; but not always. For instance, a display math environment might call for a large version of an integral sign that cannot be "reached" by emitting a U+222B INTEGRAL character, which maps to a smaller version of the glyph.

When Tectonic encounters this problem, it addresses it by creating an additional version of the relevant font file with a customized CMAP table. This *variant glyph* approach is a slightly generalized form of font subsetting, although Tectonic's implementation is much more naïve than a "real" font subsetter. In the example above, the new font's CMAP table might replace the mapping of U+222B INTEGRAL from the default integral glyph to the large version needed by the display math in question. The HTML for that canvas will then include a `<span>` tag styled to load that font, sized and positioned appropriately, containing a single U+222B INTEGRAL character.

In order to determine whether a new variant-glyph font must be created, Tectonic must parse and invert the CMAP tables of the fonts used by the document that it is processing. This process is potentially fragile, since in full generality it essentially requires inverting character "shaping" algorithms. Note, however, that it only needs to occur for characters that occur within canvases. For the main text of a document, in almost all cases Tectonic can emit Unicode output directly from the ActualText information emitted by the XeTeX engine.

4.2 The chrome: HTML, CSS, JavaScript

Tectonic's approach aims to minimize the amount of web design occurring at the TeX level. Instead, HTML content derived from the TeX input is inserted into predesigned templates. It is important to emphasize that in modern web design, such templates inevitably consist of interdependent pieces of HTML, CSS, and JavaScript code. These pieces combine to form the *chrome* of the resulting web document. Chrome encompasses everything from the high-level page layout to interactive functionality such as search, hideable sidebars, and non-linear navigation. High-quality default chrome is an essential component of the web document production pipeline.

Current efforts focus on a clean design emulating that of the tool mdBook (`rust-lang.github.io/mdBook/`), a Markdown-based, web-native documentation system. A screenshot of the default mdBook layout is shown in Figure 2. On a large screen, the default view is divided into a main content area and a sidebar. The main body text is centered within the main content area, with a maximum width to prevent line lengths from becoming excessive. An unobtrusive title bar sticks to the top of the page, but auto-hides while the reader scrolls through the main content. On mobile displays, the sidebar remains hidden by default, and can be opened using the "hamburger menu" of the title bar.

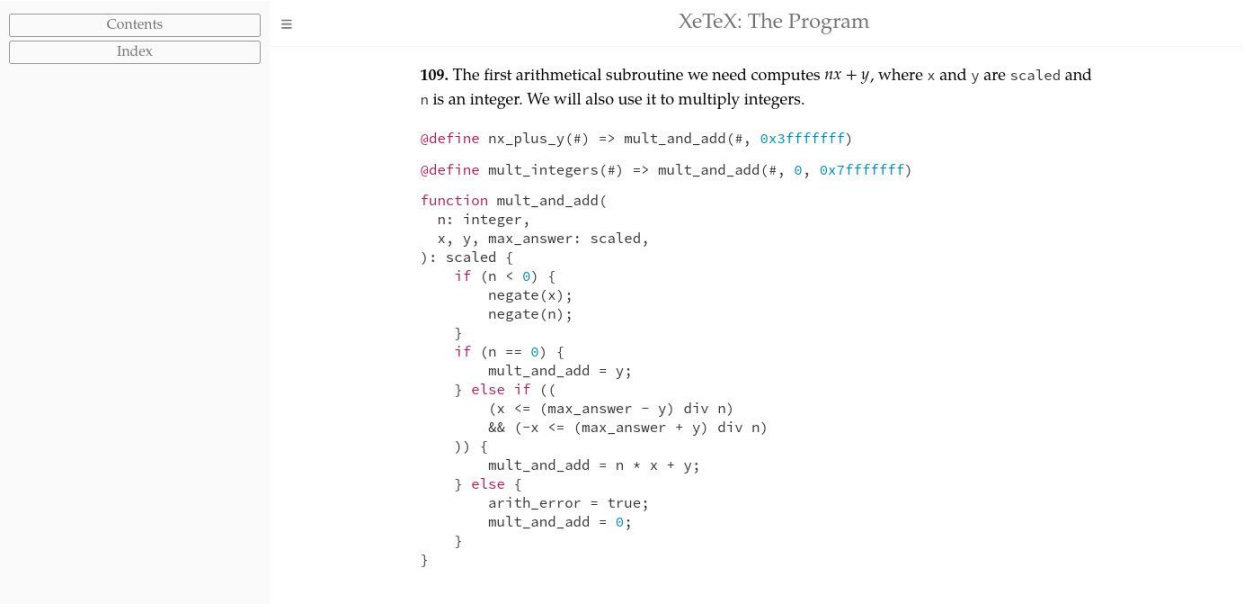

Figure 3: A snapshot of the in-development tt-weave presentation of *XeTeX: The Program*, with obvious debts to mdBook (Figure 2).

After a great deal of exploration, I believe that this layout may well be *the* optimal design for presenting general-purpose technical documents on the web. On wide screens, the positioning of the main body text becomes awkward if it is not nearly centered in the browser window. If the maximum width of the text is not limited, lines become too long, as seen on Wikipedia. On mobile, there is little enough room that there should be virtually nothing else on screen besides the main text while reading — a constraint that is accommodated well by the combination of the sticky, auto-hiding title bar and toggle-able sidebar.

Finally, many chrome designs attempt to cram information into multiple sidebars, headers, and footers. These clutter the page and are difficult to use on mobile. A better alternative is to provide these sorts of extras as "modals", overlays that can be quickly brought up and dismissed using icons in the title bar (on all platforms) or keystrokes (on desktops). This is an example of the way in which chrome consists of more than just page layout: 21st-century documents can have full-blown user interfaces.

4.3 tt-weave

Current work on Tectonic's HTML output is focusing on a very specific test case: *XeTeX: The Program*, the variant of *TeX: The Program* [1] produced from XeTeX's patched `WEB` code. It is close to ideal because it is long, highly structured, densely cross-referenced, used frequently by the author, and available as TeX source.

The TeX source generated by the traditional weave program is highly tuned for print output. Although I could potentially have worked to create HTML output from the weave-generated TeX code, I had an additional goal. I regret to say that I have always found the code listings generated by weave extremely difficult to read, even though I know that a great deal of care has gone into their design. I wanted to see if I could create a weave-like tool that could reformat the XeTeX `WEB` code into the sort of monospaced, syntax-colorized format that I'm more familiar reading.

The result of that work is a Rust tool called `tt-weave` (`github.com/pkgw/tt-weave`). It serves essentially the same function as weave, but parses the Pascal portions of `WEB` code with a high level of semantic awareness and emits them as blocks of specialized TeX code in an indented, monospaced format with embedded commands controlling syntax colorization and interlinking. The syntax of the emitted code is rewritten to superficially resemble C and Rust. For instance, logical "and" is represented using `&&` rather than $\wedge$. No semantic transforms are attempted, however. Indexing information is emitted into JSON data files that can be used by the web chrome. The tt-weave program is not intended to be a general-purpose `WEB` processor, and contains numerous hacks specific to the patched xetex.web input file. A snapshot of its output is shown in Figure 3. The design shown here is updated relative to the

version included with the HTML slides associated with this presentation.

As of this writing, the corresponding chrome is under development. The entire book text can be rendered into a single HTML file (∼10 MB) that is actually comfortable to use in the browser, and the online slides associated with this article include a snapshot of this form of output. Such a large page is an impractical delivery mechanism for general use, however, and work is underway to subdivide the output for dynamic loading. This will also help with integrating Tectonic's output into industry-standard web development frameworks (e.g., npm, webpack), which would significantly boost the productivity of development by making it convenient to adopt technologies such as SASS (sass-lang.com) or TypeScript (typescriptlang.org).

5 Outlook

The Tectonic project has been successful thus far, gathering ∼2,800 "stars" on GitHub and registering 47 distinct project contributors as of the time of this writing. While much work remains to be done to make the HTML output framework generally usable, the variant-glyph technique successfully addresses the most technically demanding problem in the current system.

The documentation of the Tectonic project — somewhat ironically, given its subject matter and aspirations — is lacking. Once the tt-weave effort has demonstrated good success with the existing *XeTeX: The Program* book, the intention is to start creating new documentation to remedy this situation.

Thus far, the prime person driving work on Tectonic has been the author of this article. Since the project's inception, however, the hope has been to make it a welcoming place for new contributors, and as the project matures, that is more important than ever. There are *numerous* areas — non-Latin scripts, accessibility, non-LaTeX workflows, TeX internals — where more expertise from around the TeX world would be hugely beneficial. People interested in engaging with the Tectonic community should visit the Tectonic discussion forum attached to its GitHub repository at github.com/tectonic-typesetting/tectonic/discussions.

Of course, Tectonic only exists because it is building on the work of the hundreds, if not thousands, of people who have collaborated to build the TeX ecosystem over the past few decades. While Tectonic positions itself as "outside of" traditional TeX in a certain sense, the sincere intent is to credit and celebrate the work of all those people as fully as possible. With immense gratitude, I thank you for sharing your wonderful creation with the world.

References

[1] D.E. Knuth. *TeX: The Program*, vol. B of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

[2] J. Lawall, G. Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 601–614, Boston, MA, July 2018. USENIX Association. www.usenix.org/conference/atc18/presentation/lawall

◇ Peter K. G. Williams
60 Garden St. MS-20
Cambridge, MA 02138
USA
pwilliams (at) cfa dot harvard dot edu
https://newton.cx/~peter/
ORCID 0000-0003-3734-3587