Toward Subtext

A Mutable Translation Layer for Multi-Format Output

Abstract
The demands of typesetting have shifted significantly since the original inception of TEX. Donald Knuth strove to develop a platform that would prove stable enough to produce the same output for the same input over time (assuming the absence of bugs). Pure TEX is a purely formal language, with no practical notion of the semantic characteristics of the text it is typesetting. The popularity of LATEX is largely related to its attempt to solve this problem. The flexibility of ConTEXt lends it to a great diversity of workflows. However, document creation is not straight-forward enough to lend itself to widespread adoption by a layman audience, nor is it particularly flexible in relation to its translatability into other important output formats such as HTML. Subtext is a proposed system of *generative typesetting* designed for providing an easy to use abstraction for interfacing with TEX, HTML, and other significant markup languages and output formats. By providing a *mutable* translation layer in which both syntax and the actual effects of translation are defined within simple configuration files, the infinitely large set of typographic workflows can be accomodated without being known in advance. At the same time, once a workflow has been designed within the Subtext system, it should enjoy the same long-term stability found in the TEX system itself. This article briefly explains the conditions, motivations, and initial design of the emerging system.

Keywords
generative typesetting, multi-output, translation layer, pre-format

Conditions for Subtext

Subtext arose as a practical conclusion during the writing of my masters thesis in New Media at the Universiteit van Amsterdam.[1]The initial impulse for the thesis itself was to investigate what available media theories existed that could articulate the dynamics of a *generative workflow* pre-occupied with outputting itself in multiple formats. In the case of the thesis, this meant PDF and HTML. Having heard about the translation software Pandoc,[2]I chose to utilize this software in my quest to produce a thesis whose materiality spanned not a single document but multiple files, programs, and 'glue' scripts. In other words, the thesis would not be a *product*, set in proprietary software like MS Word, but a *process* that could self-correct later in the future should a new format come into existence.

The raw fact of text on the computer screen is that, overall, the situation is awful. Screenic text can be divided into three categories: semantic, formal, and WYSIWYG. The semantic formats, for example HTML and XML, are notoriously machine-readable. Text can easily be highlighted, copied, pasted, processed, converted, etc. Yet the largest "reading" software for semantic formats is the web browser. Not a single web browser seems to have bothered to address line-breaking with any sort of seriousness.[3]The ubiquity of HTML, tied with its semantic processibility, means that its importance cannot be ignored as an output format. At this point, not producing an HTML version of a document that one wishes to see widely read is tantamount to removing such widespread reading as an achievable goal. To top off the complexity of the situation, the machine-readability of a semantic document is offset by a distinct reduction of human readability. Asking anyone to write a thesis directly in XML. for instance, is going to be a non-starter.

The second class of text are those defined by their *formal* nature. This is not referring to some buttoned-down attitude, but rather to an opposite directionality in terms of how the text is presented. In semantic markup, the format is not itself responsible for how a display program arranges the text---rather, the display program digests the text in light of its semantic qualities and then lays that text out according to algorithms that can and do vary between programs. The easiest way to describe this approach is that it is *top-down.*

Formal markup, on the other hand, is *bottom-up.* The final display of text is defined by discrete instructions to a program that assembles that text in a highly specific way. TEX is one obvious example of this. Likewise, PostScript and PDF are formal specifications for typesetting text. The immediate drawbacks of formal markups include an often byzantine syntax and a lack of processibility into anything other than the output formats that the formal system knows how to handle. To this day, copy-pasting from a PDF document often leads to awkward extra characters such as linebreaks in the pasted text.

The third class of screenic text system is WYSIWYG. While WYSIWYG is first and foremost a user interface design pattern (and thus can be used to output files in both formal and semantic formats), it is also defines the extremely pervasive Microsoft Word file formats. By positioning the comfort of the user above all other considerations, WYSIWYG finds its strengths in its ease of use and its inherent predictability: whatever you se on the screen should appear exactly that way on paper. By privileging the human to such an extent, however, both translatability and the typographic quality of the text suffer. Since text is intended to always appear exactly as it was input, MS Word can do no calculations for line breaks other than on a per-line basis.[4]Worst of all, WYSIWYG formats (especially those derived from Microsoft products) are difficult to integrate into a generative typesetting workflow which targets many output formats.

Problematics Within Generative Typesetting

Generative typesetting itself emerges from a very specific set of problematics. A primary concern is a reduction in syntax complexity. This is solved by the introduction of a pre-format that provides *sight-level semantics* for specifying desired outcomes in the output formats. For example, the Markdown pre-format was designed such that "a Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions."[5]

To demonstrate, while a top-level header in Markdown reads as

```
# My Header #
```

Once converted into HTML the above turns into

```
<h1>My Header</h1>
```

Sight-level semantics rely on visually distinct identifiers. This stands in sharp contrast to both HTML/XML and TEX, which rely on distinct tags combined with reserved characters. In short, this approach to semantic formatting relies on utilizing *more* reserved characters than these other systems. Which characters are chosen and the nature of their organization is an attempt to strike a balance between both readability and processibility. Like WYSIWYG, sight-level semantics represent a redistribution of agency between the human and the machine. Unlike WYSIWYG, the utilization of Markdown implies an intention for translating it into other formats.

The second problematic is an inevitable result of the first: there is *always* an edge case. Take as an example a variation on the code I've already shown. Say that instead of converting to HTML, one would rather generate a PDF using ConTEXt. Seems straight-forward right?

```
# My Header #
```

The above should simply convert into the top-level equivalent in ConTEXt. But wait.. That would be a matter of what one was trying to accomplish, wouldn't it?

After all, the above Markdown snippet could easily refer to

```
\subject{My Header}
```

or

```
\section{My Header}
```

or

```
\chapter{My Header}
```

or even

```
\title{My Header}
```

What is the solution here? Should a reserved character be adopted for each of these cases? Questions of how to deal with such edge cases are intrinsically tied to the translation layer itself: because all format translation occurs within the translation layer, it is the decisions which that layer makes that determine how edge cases are handled.

`Pandoc` provides command-line switches for turning on numbered sections and for determining the top-level "section." However, were one to desire that a custom command or macro be used in place of any of the above, a knowledge of Haskell is required to write scripts or otherwise modify the way that `Pandoc` converts its inputs. Other tricks can be employed, such as the introduction of a 'glue' layer based in a script which solves certain edge cases with regular expressions and `if` statements. From the standpoint of a generative typesetting workflow that does not require programming expertise, these solutions for edge cases are far from optimal.

One Mutable Interface to Produce Them All

Today the largest demands of digital publishing revolve around flexibility. The vast array of existing and on-coming e-readers is but one example of this. More general concerns include the necessity of both machine-readable formats and typographically sound documents. Currently this means HTML/XML and PDF.

Yet once e-readers are brought into the mix, the ePub format becomes imperative.

Yet while ePub is the most accepted format for e-reader publishing today, there is always the chance (one might even say inevitability) that a new format will become standard in the future. *Future-proofing* is a significant advantage of a generative typesetting workflow, but the programming-required nature of edge cases--and, indeed, any modification to the translation layer--decreases the adoptability of generative typesetting for non-technical fields such as the humanities.

The solution that `Subtext` proposes is to disengage both the interface to the translation layer as well as the effects of that layer. In this way `Subtext` can be seen as a very thin layer, one that takes interface primitives from a configuration file and translates them into an AST. The effects of this AST are then interpreted according to rules defined in a seperate configuration file. This file explains what should literally appear in the output file for any given AST element.

One immediately obvious benefit of this approach is the capacity to internationalize the pre-format with ease. For standard Markdown, `Subtext` would define the effect of American quotation marks ("x") as `\quotation{x}`. The interface file could be quickly modified to intepret double angle quotation marks (« x ») in the same way (`\quotation{x}`).

The effects configuration can also incorporate 'setup' requirements. If a generative typesetting workflow involved dealing with documents of either English or French, then it would be known that when double-angle quotation marks are used in the pre-format that the resulting document should have French style punctuation and spacing. The `Subtext` interpreter would then add

```
\mainlanguage[french]
\setcharacterspacing[frenchpunctuation]
```

to the pre-amble of a ConTEXt document. Likewise, specific character spacing settings could be added to the CSS of an HTML or ePub output file.

The mutability of this system is its primary characteristics. Specific text elements need not fit a pre-existing notion, as new rules can be invented and interpreted within configuration files. This capacity to 'unlock' the translation layer into an instrinsically customizable tool not only guarantees future-proofing: it also allows for highly specific workflows to be developed, as the interface and effects can be custom-crafted according to the requirements of the task.

Preliminary Thoughts on Implementation

There has yet to be a line of code committed to `Subtext`. At present it is a simple design impulse, with a variety of expectations and desires tied into a proposed means of accomplishing a more fluid and responsive generative typesetting workflow. This does not mean, however, that there has yet to be any thought put into the platforms that will underpin `Subtext`.

The first choice is the programming language. Considering the importance of parsing, grammar, and metaprogramming functionality to the implementation of a mutable translation layer, my first impulse is to write `Subtext` in Perl 6. This might come as a slight shock, but that shock should not last beyond an exploration into the power of Perl 6 grammars.[6]A robust, rules-based grammar engine was one of the top details for which Perl 6 was designed. Combined with features such as multi-method dispatch and other metaprogramming conveniences, Perl 6 is primed to host `Subtext`. Barriers to entry include a lack of documentation, but at the same time the "scene" around the programming language is small and extremely helpful. Another downside is the current speed of the language, though that is an aspect which is addressed with each monthly release. In general, the idea of Perl 6 is that it presents a mutable interface to its own programming capacities. The sympatico between the two projects is thus too significant to deny.

The configuration files themselves present a slight complication, as they need to be highly parse-able despite potentially containing every reserved character known to any programming language or syntax currently known. Thankfully, there have been many attempts to achieve this robustness. One that fits particularly well into the generative typesetting mindset which `Subtext` exemplifies is YAML (Yet Another Markup Language[8]). YAML is intended to facilitate everything from configuration files to object persistence through a human-friendly syntax. The flexibility of such a system will no doubt provide a solid foundation for implementing `Subtext`.

Additionally, there will be a standard syntax for `Subtext`. That is, there will be a defined pre-format that ships with the system. This standard syntax will include bibliographic functionality that is currently limited or non-existent in most multi-output workflows.

Longer-term goals include a web interface for dealing with the input files. Such a system would likely integrate the newly-open sourced Etherpad software for online editing. This would be tied to a version control interface based on `git` that would fill in the functionality that MS Word's 'Track Changes' system currently provides. Ideally, integrated into this system would be a real-time parser such as exhibited in the AJAX-ified interface of the WMD[7]editor, which renders the HTML output of Markdown text in real-time within the same browser window. This functionality is likely constrained by the speed of the Rakudo Perl 6 implementation. However, it is conceivable that the standard `Subtext` syntax can be parsed in JavaScript. This means that highly customized workflows would not be able to enjoy a real-time feedback interface in the near-term future. This seems to be a small trade-off for the kind of flexibility this system can enable in generative typesetting, and could easily find itself solved over the course of the continuance of Moore's Law.

Request For Comments

Though `Subtext` aims to be useful for dealing with *n+1* different output formats, initial development will concern itself with simply HTML and ConTEXt outputs. Together these two encompass the primary formats of concern. LATEX, ePub, and others can easily be added by simply defining a new set of effects.

The standard syntax has yet to be designed. Any comments or suggestions in this regard (or concerning any of what has been discussed) will be very useful. At this early conceptual stage where nothing is locked down except for the core ideas, there is a great potential for shaping the eventual system without worrying about any legacy functionality. Please do not hesitate to send me your thoughts!

Notes

1. The thesis, titled *Grammars of Process: Agency, Collective Becoming, and the Organization of Software* is available at http://mastersofmedia.hum.uva.nl/2010/09/17/grammars-of-process-agency-collective-becoming-and-the-organization-of-software-2/.
2. `Pandoc` is the only text format translation tool that currently translates into CONTEXT. It is written by John MacFarlene and is available at http://johnmacfarlane.net/pandoc/.
3. For an easy example of this, just set `text-align: justify;` in the CSS for <p> tags in an HTML document.
4. A clearly notable exception to this is Adobe InDesign and other WYSIWYG Desktop Publishing tools, in which line-breaking must be taken more seriously. In terms of "end-user" level document creation, however, the statement that linebreaking is lacking in WYSIWYG stands.
5. Markdown: http://daringfireball.net/projects/markdown/.
6. Perl 6: http://perl6.org. For an example of Perl 6 grammars, see http://perl6advent.wordpress.com/2009/12/21/day-21-grammars-and-actions/ from the 'Perl 6 Advent Calendar,' a great place to start learning about the potentials of this language.
7. WMD - The WYSIYWM Markdown Editor: http://wmd-editor.com/.
8. YAML: http://yaml.org.

John C. Haltiwanger
john.haltiwanger@gmail.com