Design decisions for a structured front end to LaTeX documents

Barry MacKichan
MacKichan Software, Inc.
barry dot mackichan at mackichan dot com

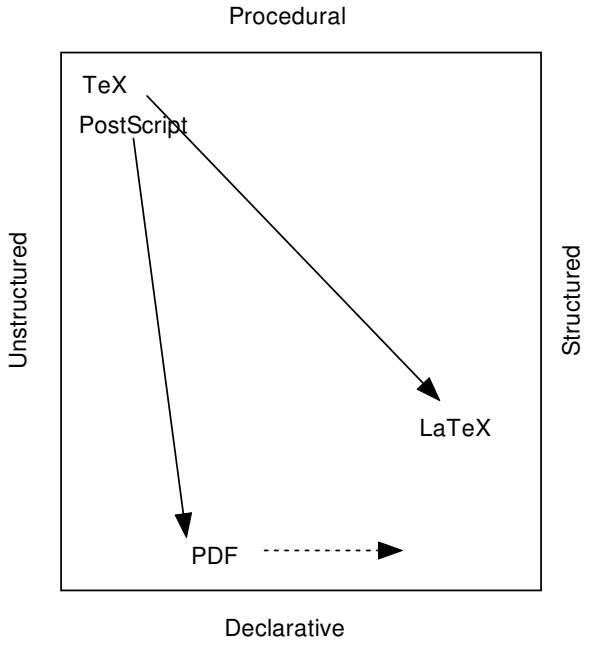
1 Logical design

Scientific WorkPlace and *Scientific Word* are word processors that have been designed from the start to handle mathematics gracefully. Their design philosophy is descended from Brian Reid's *Scribe*,[1] which emphasized the separation of content from form and was also an inspiration for LaTeX.[2] This *logical design* philosophy holds that the author of a document should concern him- or herself with the content of the document, and with identifying the *role* that each bit of text plays, such as a header, a footnote, or a quote. The details of formatting should be ignored by the author, and handled instead by a predefined (or custom) style specification.

There are several very compelling reasons for the separation of content from form.

- The expertise of the author is in the content; the expertise of the publisher is in the presentation.
- Worrying and fussing about the presentation is wasted effort when done by the author, since the publisher will impose its own formatting on the paper.
- Applying formatting algorithmically is the easiest way to assure consistency of presentation.
- When a document is re-purposed it can be reformatted automatically for its new purpose. This can happen when a document is put on the Web in addition to being published, or even when the author sends the document to a new publisher.

The most powerful typesetting programs tend to be programming languages themselves. The two most prominent examples are PostScript and TeX. Although these are extremely powerful, they are not always simple, and they do not separate content from form. Consequently, there is a migration on the following plot from the top to the bottom, and from the left to the right.



Thus, PostScript is a powerful programming language, but it was later supplemented by PDF, which is not a programming language, but instead contains declarations of where individual characters are placed. PDF is not structured, but Adobe has been adding a structural overlay. LaTeX is quite structured, but it still contains visible signs of the underlying programmability of TeX, so I haven't quite placed it at the bottom of the plot. The pattern is that power and flexibility generally get supplemented or replaced in some circumstances with structured and declarative alternatives.

The original design philosophy for *Scientific WorkPlace* and *Scientific Word* was to make visual word processors that live at the bottom right of the diagram, and produce their output by generating LaTeX using one of over a hundred typesetting styles. This is the optimal solution for publishing, at least when we support a publisher's style, or when a publisher's style uses the same tags as one of the standard LaTeX document types.

2 Enter the customer

Although this philosophy works very well for publishing, many of our customers want to have greater control over the appearance of their documents. The

[1] Brian K. Reid, "Scribe: A Document Specification Language and its Compiler," Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, PA, Oct. 1980.

[2] Leslie Lamport, LaTeX: A Document Preparation System, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, Second Edition, 1994.

truth is that not all mathematical documents are written for publication in a journal. The author might want to post a document on the Web or to send out preprints, or to prepare reports that will not be published, or to prepare handouts for students. The cold hard truth is that programs like Microsoft Word — despite its intellectual roots being also in *Scribe* — have over the years encouraged users to fiddle and futz with formatting. The experts may all agree that the result is ugly, but the customer is the one who pays our salaries.

In the past, *Scientific Word* users had a hard time if they wanted to change or add to a style. The advice of our tech support staff has been:

- You don't want to do that
- You shouldn't do that
- You can use package X to do that
- You can rewrite the style file

We no longer give the first two responses, and our users are not going to be able to use the fourth bit of advice. Due to the large number of useful packages, we now encourage users to start with a standard LaTeX document type and to use packages. This works, but it is not the most elegant way to solve the problem, since you shouldn't have to write options for the `geometry` package in order to change a margin.

We also allow the user to enter snippets of raw TeX or LaTeX code in what we call a "TeX button" (which is how we enter "TeX" and "LaTeX") but this runs counter to the design philosophy, and can't address problems when a user wants to change, for example, how list items are generated (since the code to be added would be in the middle of code we have generated).

3 A statement of the problem

This discussion now allows a statement of the problem we are solving.

1. We want an internal form for our documents that is both rich and extensible, and a rendering engine that is rich enough to render a LaTeX document and which is extensible.

2. We want to convert a LaTeX document to our internal form in a way that is extensible and preferably uses standard, well-documented tools, and in particular does not require access to our source code.

3. We want to convert our internal form to LaTeX in a way that is extensible and uses standard tools, and does not require access to our source code.

Part of the motivation for not needing access to our source code is that extending these operations will be easier for us if it is not necessary to change and re-build C++ code in order to support a new tag or to change the behavior of a standard tag. The other part of the motivation is that if the tools are standard and well-documented, then advanced users can make their own changes.

3.1 Internal form of a document

Scientific Word has an internal form that is not LaTeX but looks superficially like LaTeX, and we have an adequate rendering engine for it. However, it is not extensible — that is, to extend it means rewriting C++ code and extending the rendering engine. To avoid this problem and get the extensibility we need, we choose an internal form that is rich enough and extensible (and it must also be declarative and structured). The obvious candidate (at least in this century) is XML. We are basing future versions of our software on the Mozilla Gecko rendering engine for HTML and XML. Tags can be introduced at will, and CSS (Cascading Style Sheets) are used to determine how these tags appear on the screen.

Some of the features of Gecko that are very useful to us are:

- The rendering engine is open-source under a license that allows us to extend it if necessary.
- The rendering engine is rich and powerful (the program user-interface is in fact a Gecko document).
- XML is a standard that is easily converted to and from LaTeX.
- A powerful scripting language is integrated into Gecko.
- A technology (XBL–XML Binding Language) allows attaching behavior to (new) XML tags.
- A system of broadcasters and observers simplifies coordinating the behaviors of objects.
- Support for infinite undo and redo is built into the document-modifying functions.

3.2 Conversion from LaTeX

Scientific Word does not process TeX or LaTeX files with TeX. It simply determines the structure of the file by recognizing tags such as `\section` and `\subsection`. In the past, it has caused problems when users defined their own macros: we did not recognize them and loaded the macro invocation as a TeX button. Beginning with version 5.5 (two years ago) we now run a version of the TeX macro processor, and we evaluate macros defined by the user, but we do not evaluate macros defined in LaTeX or any of the standard packages. The result should be

a document that contains only the standard macros, and which can be read by *Scientific Word*.

We continue with this same approach in our new architecture, except that the definitions of the standard LaTeX macros converts them to XML. The resulting files are complicated, but most of the complication is in some utility macros that make the final macros quite easy to understand. Some sample code from one of these files is:

```
\def\out@begin@abstract{%
 \msitag{^^0a}%
 \msiopentag{abstract}{<abstract>}
}
\def\out@end@abstract{%
 \msitag{^^0a}%
 \msiclosetag{abstract}{</abstract>}
}
```

This is all that is required to convert the `abstract` environment to XML.

3.3 Conversion to LaTeX

The conversion to LaTeX is done using XSLT (XML Stylesheet Language Transformations). As the name implies, XSLT was designed as part of a method of applying styles to XML objects, which sometimes requires making some transformations or re-ordering the XML elements. It has evolved into a powerful standalone transformation language for XML documents. It can be used to transform XML into XML, or XML into text, which includes TeX.

For instance, here is the XSLT rule that generates the `abstract` environment:

```
<xsl:template match="abstract">
  \begin{abstract}
    <xsl:apply-templates/>
  \end{abstract}
</xsl:template>
```

When XSLT finds the `<abstract>` tag, it first generates `\begin{abstract}`, then applies any rules needed for the content of the tag, and finally generates `\end{abstract}` when it reaches the end of the `abstract` node. The tag may have attributes, which might affect the TeX generated, and the rules can depend on the context of the tag.

The point here is that it is relatively easy to add support for new tags, or to change the TeX that gets produced by a tag. In older versions of our products, these operations took place in compiled code, but now they are controlled by text files that can be replaced or modified without rewriting or recompiling C++ code. It is now feasible to support different flavors of TeX for *Math Reviews*, or to support something like ConTeXt.

The next section addresses the question of how you can tailor the on-screen presentation of a tag.

4 Some examples

4.1 Displaying 'LaTeX' on screen

This is a brief discussion of how you can display a new tag, such as `<latex/>`, on the screen. This is done by using XBL. We'll skip lightly over the details.

In a CSS file there is a line that tells Gecko that special rules apply to this tag:

```
latex {
  -moz-binding: url(
    "resource://app/res/xbl/latex.xml#latex");
}
```

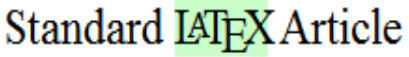
In the file `latex.xml`, there is a section that says how to display the tag:

```
<xbl:content>
  <sw:invis><xbl:children/></sw:invis>
  <sw:latex2>L<sw:latexa>A</sw:latexa>
  <sw:latext>T</sw:latext>
  <sw:latexe>E</sw:latexe>X</sw:latex2>
</xbl:content>
```

Each letter in LaTeX (almost) is in a separate tag, which allows us to change the style for each letter. Here is the style rule for the 'A' (the tag `latexa`):

```
latexa {
  font-size: smaller;
  position: relative;
  bottom: .15em;
  left: -0.20em;
}
```

This rule shrinks the 'A' and moves it up and left. A style rule for the `latex2` rule changes the letter spacing to squeeze them together a bit. The final result on the screen is:



Actually, what appears in the internal format is `<latex>LaTeX</latex>`. The content of the tag ('`LaTeX`') is thrown away, except when the XML document is viewed by some other browser, such as Internet Explorer, or even Firefox. Internet Explorer, when it sees the `-moz-binding` statement in the CSS file will ignore it completely. Firefox will understand it, but will be unable to find the `latex.xml` file, which is internal to our program. As a result, they will ignore the `latex` tag and will simply display the contents. Thus, the above displayed on Firefox will appear as:

Standard LaTeX Article

Of course, the LaTeX generated by this tag, no matter what its content, will be `\LaTeX`.

4.2 Spaces

LaTeX provides a wide choice of spaces, both horizontal and vertical. It is possible to make them visible by selecting a menu item "Show Invisibles". This is accomplished in the same way as the above example, with special CSS rules to apply in the case when "Show Invisibles" is on.

5 User interface enhancements

The next two items are not particularly related to the new architecture for *Scientific Word*; rather, they can be looked on as one solution to the problem of converting a rich keyword-value interface to a friendlier (to the novice) dialog-based interface. The result is marginally less powerful, but still allows the advanced user to get access to almost all the features of the keyword-value interface.

The dialog shown in figure 1 is for selecting OpenType fonts. Before the user gets to this point, he will be warned that if he proceeds, his document will have to be compiled with XeTeX and therefore will not be completely portable.

This allows the user to pick the three main fonts: the main (roman) font, the sans serif font, and the monospaced font. He can also choose other fonts and give them names. We have a `rtlpara` tag for right-to-left text, and this uses the `rtl` font, for which the user has chosen Narkisim.

There are many font attributes, and many are not widely supported in available fonts, so we have chosen only two for access by checkboxes: old-style numerals and swash italics. Other attributes are accessible, but only by falling back on the keyword-value interface and clicking on the "Go native" link; see figure 2.

The first line in the "Go native" box was provided automatically since the user had clicked on "Old style nums" and "Swash". The user added the next line to use the MinionPro-Bold font as his bold Roman font rather than the default Semibold. This interface allows almost complete access to the power of the `fontspec` package but gives more casual users the ability to choose basic fonts easily.

Another dialog interface to package options is the page layout dialog, as shown in figure 3. Here the user is adjusting the left margin by pressing or holding the up or down arrow key in the left margin width field.

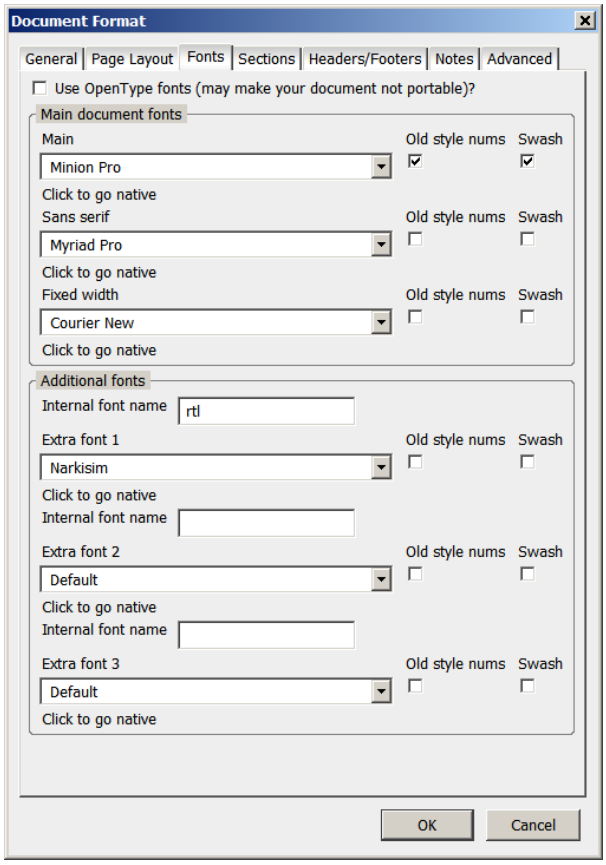


Figure 1: Dialog for selecting OpenType fonts.

6 Conclusion

Scientific WorkPlace and *Scientific Word* are designed to make it easy for authors to write a beautiful LaTeX document with skills they already have. To keep its simplicity from becoming a limitation, we have to provide ways for more advanced users to override the default decisions that *Scientific Word* makes. This paper has covered a few of the new technologies we are using to make a more modular system, with the interconnections provided by stable and well-documented standards in a way that we, or a knowledgeable user, can easily customize. We expect this new platform to allow us to be more nimble than before in responding to the changing needs of our customers, and to serve as a solid base for the next ten years of development.

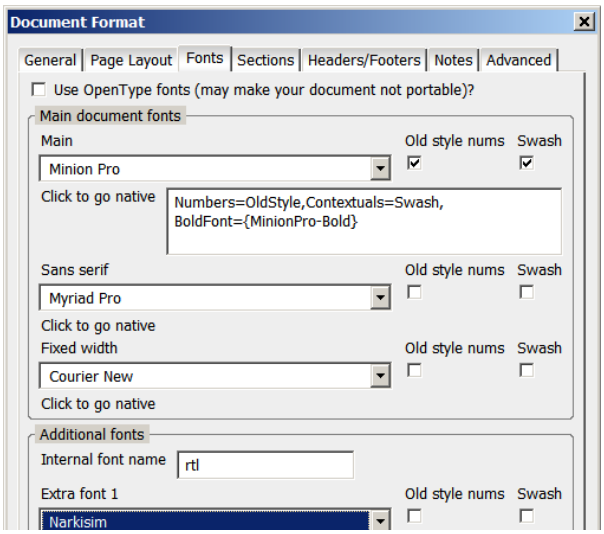


Figure 2: Selecting OpenType font attributes.

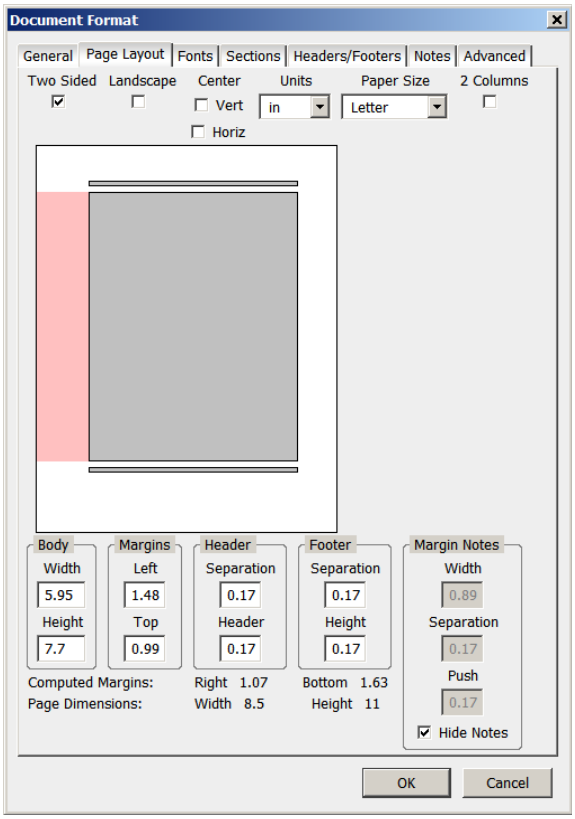


Figure 3: Page layout dialog.