The PracTEX Journal, 2008, No. 3
Article revision 2008/11/17

plasTeX: Converting LaTeX Documents to Other Markup Languages

Tim Arnold

Email a_jtim@bellsouth.net

Abstract This article introduces plasTeX, a software package for converting LaTeX documents to other markup languages. It begins with usage details including examples of how to create HTML and DocBook XML from LaTeX sources. Then, it describes development details: how plasTeX works and how developers can use it to create or extend a publishing workflow in a production setting. Finally, it ends with some examples of customizing the parser and renderer as well as suggestions of how others can contribute to this open source project.

1 Overview

plasTeX is an open source software package that converts LaTeX documents to other document formats such as HTML and XML. It is written in Python and is available on SourceForge under the MIT license. See section 5 for links to more information.

plasTeX converts a LaTeX document in two stages. First, it tokenizes and parses the LaTeX source following rules similar to those of the actual TeX tokenizer and macro processor. At this stage the document exists internally as an ordered data structure called a document object model (DOM). Second, plasTeX applies a rendering template to that data structure to create the final output. This second stage can be repeated several times to create multiple output formats from a single parsing stage.

The following diagram shows how a document is transformed from LaTeX to another markup language.

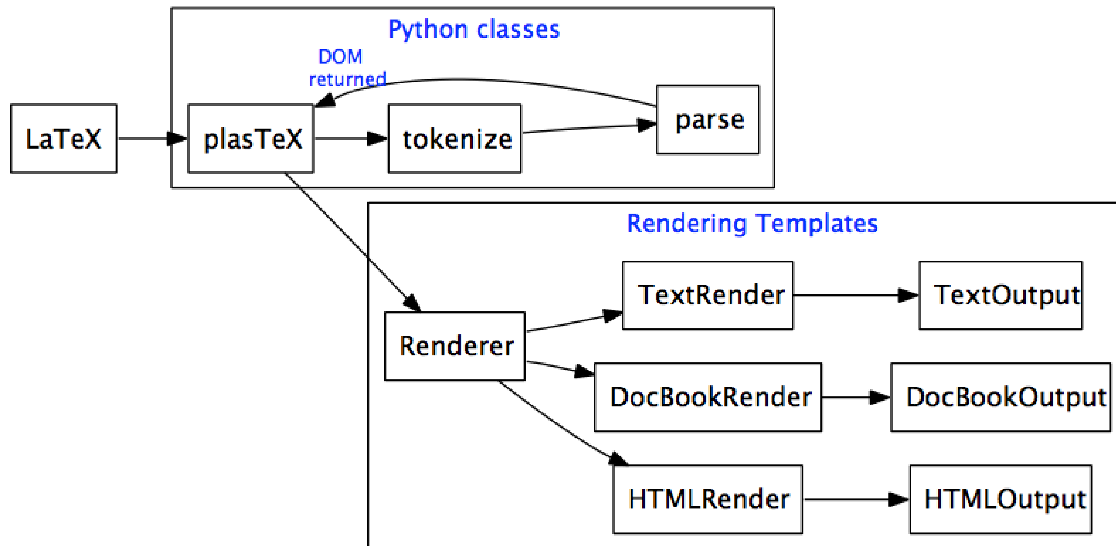

Figure 1: plasTeX Converts LaTeX to Multiple Output Formats

The core of plasTeX is the tokenizing/parsing engine. plasTeX understands the content of a source document through its access to actual LaTeX packages and Python classes. All built-in LaTeX macros are defined as Python classes in plasTeX.

As plasTeX reads the source file, it must understand each macro it encounters. If a Python class exists that matches the name of the macro, that class definition is used. If a matching Python class is not found, plasTeX searches for the actual LaTeX package macro and expands it before continuing with the source document. plasTeX also expands macros defined within your document.

Renderers can be plugged in as back-ends. Several renderers are bundled with plasTeX; if a custom output format is needed, it is straightforward to write a custom renderer. Users who create custom renderers can easily share them with others through the plaTeX SourceForge site. For example, BrlTeX generates Braille output from LaTeX sources (see section 5 a link to more information).

Out of the box, plasTeX cannot convert every LaTeX document. However, using inheritance techniques described in section 3.4, the system can be modified to produce reasonable output from almost any document.

Professional document producers, casual end users, and programmers use different methods of working with plasTeX. This article first describes the command-line interface to plasTeX to show how plasTeX can be used as is. Later it discusses customization methods for documents with "special needs".

1.1 Command Line Interface

The user interface for plasTeX is the `plastex` command. To convert a LaTeX document to XHTML using all of the defaults, type the following on the command line, where `mylatex.tex` is the name of your LaTeX file:

```
plastex mylatex.tex
```

The `plastex` command has several options[1] to help you control the rendered output. Some important optional capabilities are listed below:

- selectable output format (HTML, DocBook XML, and so on)
- generation of multiple or single output files. The automatic splitting of files can be configured by section level or controlled programmatically.
- image generation for portions of the document that cannot be easily rendered in a particular output format (for example, equations in HTML)
- post-processing hooks for simple customization
- configurable input and output encodings
- selectable themes

One of the most powerful options is the theme selection. A theme is a set of templates that plasTeX uses during rendering to give your documents a different "look and feel." The following example shows how to specify a theme from the command line.

```
plastex --theme=plain mylatex.tex
```

The theme alters the appearance and behavior of your HTML documents (to match a corporate style for example). Themes are especially powerful when coupled with CSS style files. plasTeX provides the following themes which can be used as a guide to quickly create custom themes:

1. You can enter the `plastex` command with no arguments to get help on its options and usage.

default full-featured theme with navigation bars, style information, and breadcrumbs.

python a variant of the default theme which renders the look and feel of the standard Python documentation.

plain bare-bones theme, intended for those who prefer to use pure CSS styles with plain HTML for complete separation of style and content.

minimal specialty theme, intended for special cases when segments of HTML are needed for copying and pasting. Contains no navigation.

For more information about themes, see section 3.5.

1.2 Mathematics

As plasTeX parses a LaTeX document, it creates document data and keeps track of the original source. This means that, after parsing, plasTeX has two ways to access each document element: as part of the document data or as the LaTeX source from which the element came. This is important when dealing with equations or picture environments that might not have corresponding markup in the output format (HTML, for example).

plasTeX uses an imager (`dvipng` by default) to create images of document elements that cannot be produced otherwise. Several imagers are available in plasTeX; any program that can be used to convert LaTeX math into an output format can be used.

When plasTeX converts LaTeX to DocBook XML, both the image and the LaTeX source are used in the final document. See section 2.2 for an example.

1.3 Languages and Encoding Support

plasTeX comes bundled with translated terms for document elements in the following languages: Dutch, French, German, Italian, Portugese, and Spanish. It also includes three variants of English: American, Australian, and British.

You can use your own translated terms by specifying a local translation file. These custom terms are substituted into the output document during the rendering stage.

plasTeX can read and write the same encodings as Python itself. For example, it is a simple process to read LaTeX encoded as `Latin-2` and write the output format as `UTF-8`. The output encoding is independent of the format; that is, all output formats support all encodings.

With its encoding support and XML output, plasTeX can play a critical role when producing documents in multiple languages. Figure 2 shows a possible workflow.

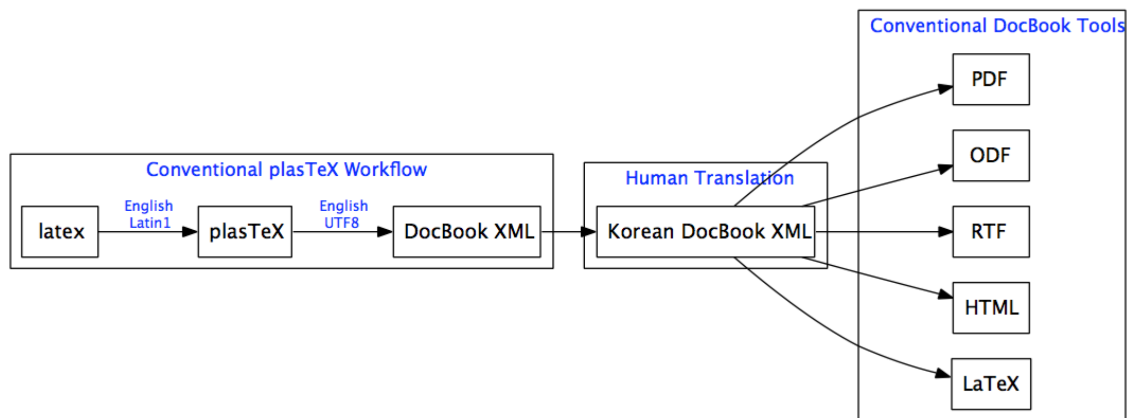

Figure 2: Producing Documentation in Multiple Languages using plasTeX

Suppose a document is authored in English LaTeX using `Latin-1` encoding, final documents are needed in both English and Korean, and that the translators are unfamiliar with LaTeX or possess translation memory in XML format. In such a situation, you can use plasTeX to convert the LaTeX `Latin-1` source document to DocBook XML in `UTF-8` encoding. Translators can then work with the XML document and return it in the new language. Off-the-shelf tools then convert the DocBook document to produce a final output document in the new language (see section 2.2 for more information on the DocBook output).

1.4 Supported LaTeX Packages

plasTeX defines all built-in LaTeX macros, as well as some of the more popular packages. The following shows some of these supported packages.

alltt	float	keyval	subfig
amsmath	graphicx	lipsum	textcomp
changebar	hyperref	longtable	url
color	ifthen	natbib	wrapfig

2 Usage

To get started with plasTeX, you need the following:

- Python distribution 2.4 or later
- plasTeX distribution
- Python Imaging Library
- LaTeX distribution

The Python Imaging Library and the LaTeX distribution are used together to create images for the rendered document (for example, math images in HTML)

2.1 HTML

Figure 3 displays the default HTML output generated by converting the well-known `sample2e.tex` file. The command to generate the output is:

```
plastex sample2e.tex
```

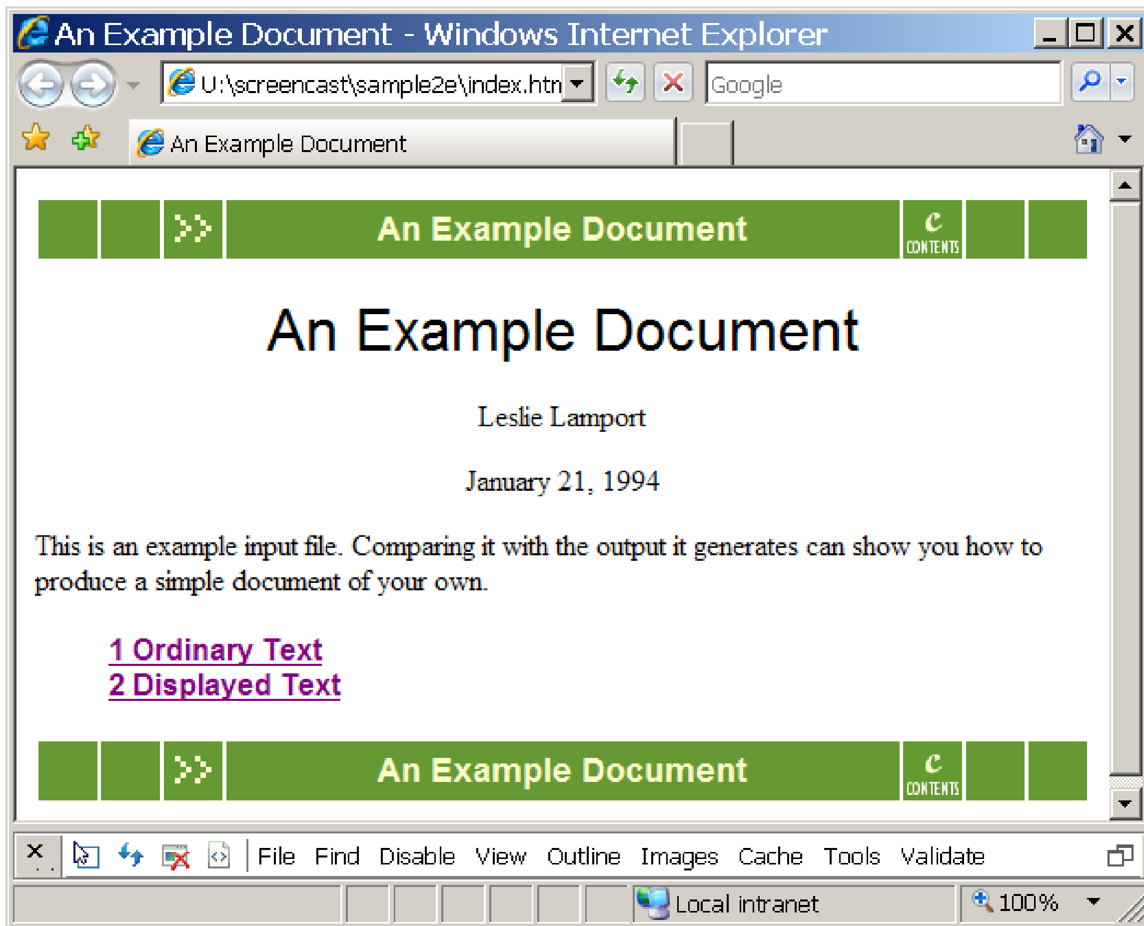

Figure 3: Default Output with HTML Renderer

An added bonus with plasTeX HTML generation is that it generates supplemental files so you can easily create further processed HTML, such as Windows compiled help, EclipseHelp, and JavaHelp™. Any of these formats can be produced by downloading the free compiler for the corresponding format and invoking it with the files generated by plasTeX. For links to downloads, see section 5.

Windows compiled help (CHM) is easily generated with Microsoft® HTML Help Workshop:

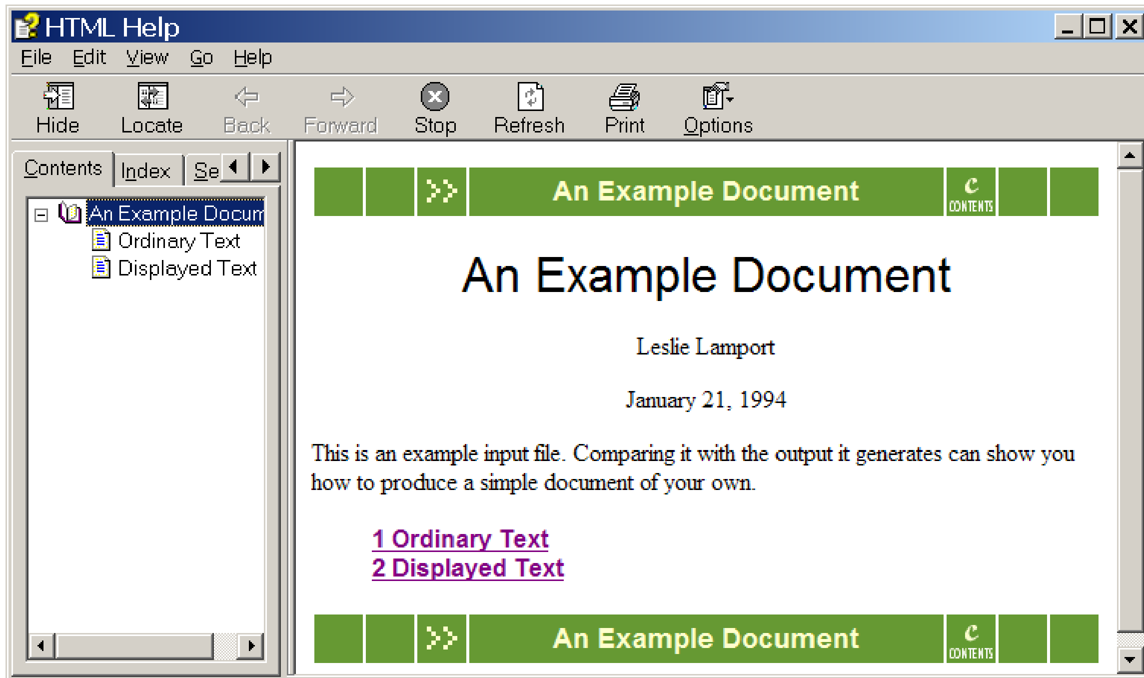

Figure 4: Windows Compiled Help View

JavaHelp is generated with the `hsviewer` available from Sun Microsystems®. The following command creates the JavaHelp output, where `documentname.hs` is a supplemental file created by plasTeX during HTML conversion:

```
java -jar path_to_hsviewer.jar -helpset documentname.hs
```

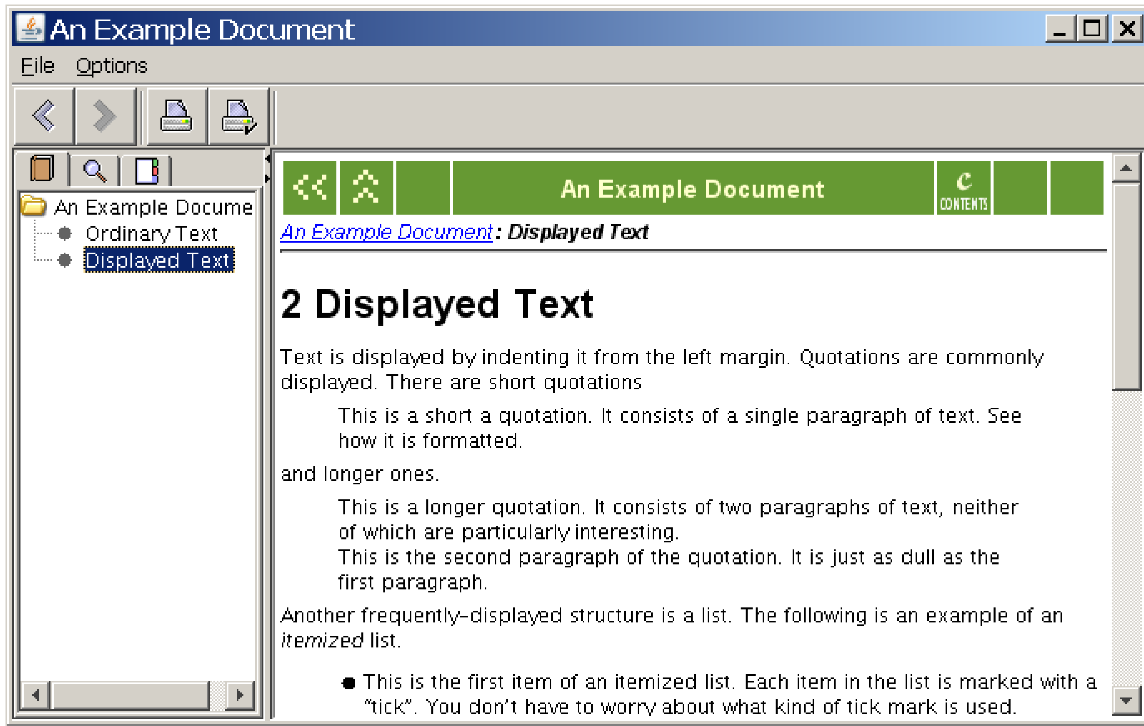

Figure 5: JavaHelp View

2.2 DocBook XML

plasTeX can produce DocBook 4.5 XML from many LaTeX documents. However, not all the elements available in DocBook have corresponding LaTeX macros. Although plasTeX handles much of the mapping, some customization might become necessary for documents with complex formatting or processing needs. Section 3 describes this type of customization.

To convert a LaTeX document to DocBook XML, type the following on the command line, where `mylatex.tex` is the name of your LaTeX file.

```
plastex --filename=mylatex.xml mylatex.tex
```

The option `-filename=mylatex.xml` specifies that the XML output be generated in a single file called `mylatex.xml`.

The following figure displays a view of the `sample2e` document as formatted in DocBook XML (and displayed by the XMLMind Editor).

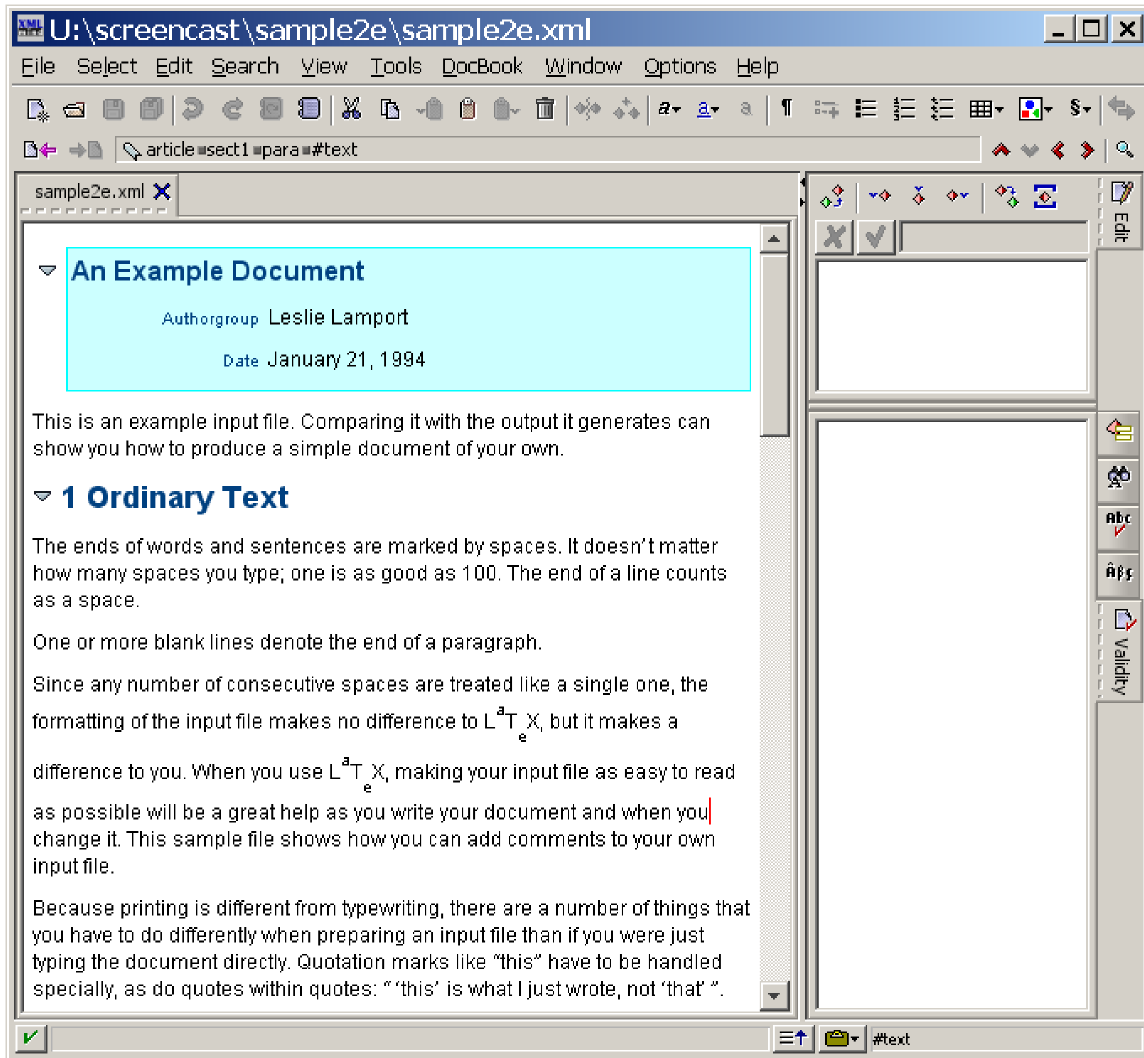

Figure 6: DocBook XML Output

A powerful toolchain exists to create many formats from DocBook, as can be seen in Figure 2. For that reason plasTeX doesn't directly produce bibliographies, resolve cross references, or produce indexes for DocBook. However, it does gen-

erate the supplemental files the DocBook tools expect for processing these document elements.

2.2.1 Bibliography

If you use BibTeX in your LaTeX workflow, you can generate an XML version of your BibTeX database. DocBook processing tools can then combine that database with your DocBook document to create the bibliography on the fly. One tool that can converts BibTeX to XML (among other things) is JabRef, see section 5 for a link.

2.2.2 Cross References

Cross references are rendered directly in the document as `xref` elements. Furthermore, an external `xref` database is generated. DocBook processing tools use this database to resolve cross references across documents.

2.2.3 Index

Index terms are rendered directly in the document. DocBook processing tools automatically create an index for your document on the fly.

2.2.4 Mathematics

plasTeX uses both the rendered image and the LaTeX source when representing mathematics in DocBook. For example, an inline equation such as $x + y = \frac{1}{z}$ would be converted to the following docbook element:

```
<inlineequation>
  <inlinemediaobject remap="math">
  <imageobject>
    <imagedata role="math" fileref="image002.png"" />
  </imageobject>
  <textobject role="tex">
    <phrase>x+y=\frac{1}{z}</phrase>
  </textobject>
  </inlinemediaobject>
```

```
</inlineequation>
```

A document that uses this system validates against the DocBook DTD, and yet postprocessing extensions have easy access to the original LaTeX mathematics.

2.3 Other Formats

In addition to the HTML and DocBook XML renderers, the following renderers are also available:

Text renders LaTeX to plain text in the specified encoding. For example, the rendered output of the first part of section 2 from this document is as follows:

```
2 Usage Details

 To get started with plasTeX, you need the following:

   * Python distribution 2.4 or later
   * plasTeX distribution
   * Python Imaging Library
   * LaTeX distribution.
```

Figure 7: Text Renderer Output

ManPage renders LaTeX to the UNIX man page format, which can then be displayed with the commands `groff` or `nroff`.

S5 renders your LaTeX document (Beamer class only) to S5 output. S5 is a standards-based slideshow system.

Braille renders LaTeX documents to Braille. This format is not bundled with plasTeX distribution but is available as BrlTex on the web. See the BrlTeX entry in section 5 for details.

Direct to XML not a true renderer, but a representation of your document in plasTeX's internal XML format. This representation can be useful to review if you encounter problems in rendering.

Producing this output format requires a slight modification of the `plastex` command line script: uncomment the following lines in the script and run `plastex` again.

```
## Write XML dump
#outfile = '%s.xml' % jobname
#open(outfile,'w').write(document.toXML().encode('utf-8'))
```

An XML file is created that displays the structure of the document. The following LaTeX markup used in this document to display Figure 6

```
\begin{figure}[H]
   \includegraphics[width=\textwidth{dbk}
   \caption{DocBook XML Output}\label{fig:docbook}
\end{figure}
```

has the following internal representation:

```
<par id="a0000000909">
  <figure id="a0000000910">
    <plastex:arg name="loc">H</plastex:arg>
    <par id="a0000000911">
      <includegraphics style="width:5in" id="a0000000912">
        <plastex:arg name="options">{u'width':u'5in'}</plastex:arg>
        <plastex:arg name="file">dbk</plastex:arg>
      </includegraphics>
    </par>
    <par id="a0000000913">
      <caption ref="4" id="fig:docbook">
        <plastex:arg name="toc" />
        <plastex:arg name="self">DocBook XML Output</plastex:arg>
      </caption>
    </par>
    <par id="a0000000914">
```

```
        <label id="a0000000915">
          <plastex:arg name="label">fig:docbook</plastex:arg>
        </label>
      </par>
    </figure>
  </par>
```

3 Customizing plasTeX

3.1 Process Overview

plasTeX might need modification to convert documents that contain custom macros, complex macros, or macros that have no mapping to the output format. The three tasks in creating the output are parsing, rendering, and imaging. Each task is described as follows.

To convert a LaTeX document, plasTeX must first tokenize and parse the source. It understands how to do this for each document element from the corresponding Python class definitions. plasTeX finds these definitions in its standard path and the locations defined in the environment variable `PYTHONPATH`.

Next, plasTeX hands off the resulting data structure (DOM) to the renderer. The rendering templates must display (or more correctly, *mark up*) the document elements for the intended output format. plasTeX finds these templates in its standard path and the locations defined in the environment variables `XHTMLTEMPLATES` and `DocBookTEMPLATES`.

Finally, for any elements that must be handled externally (that is, by the imager), plasTeX hands off an internally produced `images.tex` file to the configured imager, which employs LaTeX itself. plasTeX finds the necessary files by using the `kpsewhich` command and the locations defined in the environment variable `TEXINPUTS`.

Thus, the sequence of tasks to customize plasTeX is as follows.

1. write class definitions for your new commands or environments. Set the environment variable `PYTHONPATH` to include the location of your class definitions. Also, see section 3.2.1 which describes a fallback mechanism that you can use to redefine simple macros in LaTeX itself.

2. Write templates that correspond to your new commands or environments. Set the environment variable `XHTMLTEMPLATES` (or `DocBookTEMPLATES` for the DocBook renderer) to include the location of the templates.
3. Confirm that LaTeX can find the packages necessary to create the images. If the `kpsewhich` command cannot find the packages automatically, set the environment variable `TEXINPUTS` to include the location. Note that plasTeX typically handles imaging automatically when a standard LaTeX distribution is installed.

3.2 Parsing Details

The goal in this stage of customization is to enable plasTeX to recognize and correctly parse commands and environments that it encounters in the source document.

The parsing task usually involves Python classes, because when a source document loads a package, plasTeX first looks in the `PYTHONPATH` environment variable for a Python module of the same name. For example, if the package loaded is `mylocals`, plasTeX checks for a Python file `mylocals.py`. However, if it does not find that file, it falls back to the LaTeX packages by using the `kpsewhich` command to find a package called `mylocals.sty`.

These two alternatives lead to the two possible methods of parser customization.

- Create a simplified version of the original LaTeX macro that plasTeX uses (LaTeX uses the more complex version).
- Implement the macro as a Python class.

3.2.1 Conditional LaTeX Macros

The simplest customization method is to modify the LaTeX macro with conditional processing. That is, create a simplified version of the macro by using the `\ifplastex` command. The command is built into plasTeX; during the plasTeX processing the command is set to `true` and typically invokes simpler instructions than those used when LaTeX processes the document.

In the following example[2], suppose you have a local LaTeX package called `mylocals.sty` which defines a new command `\foo` that takes a single argument.

```
\newcommand{\foo}[1]{
   \ifplastex\else\vspace*{0.25in}\fi
   \textbf{\Large{#1}}
   \ifplastex\else\vspace*{1in}\fi
}
```

In the LaTeX version the `\foo` command inserts some vertical space before and after the argument, which it typesets in bold at a `Large` size. The plasTeX version bypasses the vertical space commands but retains the typesetting of the argument. With the macro so defined, plasTeX will recognize and correctly parse the `\foo` command.

Suppose that `mylocals.sty` also defines a new environment called `mybox`.

```
\ifplastex
   \newenvironment{mybox}{}{}
\else
   \newenvironment{mybox}
      {\fbox\bgroup\begin{minipage}{5in}}
      {\end{minipage}\egroup}
\fi
```

In the plasTeX version, the environment is merely defined and no action is taken; in the LaTeX version a box is set around a minipage. With this information, plasTeX can now parse the environment correctly. It can then be displayed by the renderer (perhaps to put a box around the contents).

In this example, the separation of tasks can be clearly seen. The LaTeX code handles both the parsing knowledge and the appearance; the plasTeX code separates the tasks into the macro definition seen here (parsing knowledge) and the rendering template (appearance), which is described in section 3.3.

2. The examples presented in this paper are simple enough that plasTeX could parse and render them correctly with no additional effort. They are presented here only as learning examples.

3.2.2 Class Definitions

The alternative to writing macros with conditional definitions is to write a Python class that corresponds to the command or environment. This is the recommended method for defining complex macros.

The plasTeX documentation describes in detail how to write methods or functions to override plasTeX's normal behavior or to add new behavior. For example, you can define methods to manipulate the document structure, create and change counters, and store data for use in postprocessing.

More commonly however, a new macro requires normal parsing behavior. The only requirement for parsing the new macro is to define the macro's signature (that is, specify the options and arguments that the macro takes). The `args` variable in the class definition specifies this information. You can specify any number of optional arguments and mandatory arguments.

To indicate that an argument is optional, surround the optional arguments with matching square brackets([]), angle brackets (<>), or parentheses (()), just as it is written in the LaTeX source. The following list describes some examples:

`args = 'title'` specifies a single mandatory argument named `title`.

`args = 'id title'` specifies two mandatory arguments named `id` and `title`.

`args = '[toc] title'` specifies an optional argument named `toc` and a single mandatory argument named `title`.

`args = '[options:dict] title'` specifies an optional list of keyword-value pairs named `options` and a single mandatory argument named `title`.

After the arguments are parsed,they are set in the `attributes` dictionary of the document element. These attributes can then be used in the rendering stage as `self/attributes/`*`argumentname`*. plasTeX's `arg` string provides a powerful mechanism for quickly defining complex macros. The plasTeX documentation has complete details with several examples.

To continue with the example described in the previous section, suppose that the macro definitions remain as they were originally written (that is, without using the `\ifplastex` command) and the plasTeX class definitions are written in the Python class file (Python module) `mylocals.py`. The corresponding Python code might look like this:

```
from plasTeX import Command, Environment
class foo(Command):
    args = 'footext'

class mybox(Environment):
    pass
```

The first line imports some of the basic plasTeX functionality—the foundational classes `Command` and `Environment`. For many macros, you need little else to teach plasTeX how to recognize and parse the new macros it encounters. With the `\foo` command for example, you can tell plasTeX that this is a command with a single argument named `footext`. This argument name is used by the renderer after the parsing stage. With the `mybox` environment, you define the macro as an environment.

These definitions, written in `mylocals.py`, make up all the information plasTeX needs in order to recognize and parse the new command and new environment.

3.3 Rendering Details

A renderer is a collection of templates[3] that take the data from the parsed document and can display or markup each document element encountered.

Assuming that you have a CSS stylesheet for presentation, the following is one possible template for producing HTML from the definitions given in the previous section:

```
name: foo
<b class="Large" tal:content="self/attributes/footext"></b>

name: mybox
<blockquote class="mybox" tal:content="self"></blockquote>
```

The argument given in the `\foo` command is named `footext` in the class definition and is available in the renderer as `self/attributes/footext`. It is placed

3. The default template engine is SimpleTal which is based on the Python package, Zope Page Templates

inside a `<b>` element which will be displayed as bold in the final display. The `class="Large"` attribute might correspond to a class defined in an accompanying CSS file to render the contents at a larger than normal size.

The contents of the `mybox` environment is placed inside a `blockquote` element with the `class="mybox"` possibly handling further display formatting via CSS.

3.4 Inheritance and Aliasing

Inheriting from the base plasTeX classes as shown in section 3.2.2 is a fast and powerful way to customize your own documents. Te previous example inherited definitions and behavior directly from the `Command` and `Environment` classes. You can also inherit from other plasTeX classes. Choose the class with behavior that most closely resembles your custom macro. For example, suppose that you have an environment in which you display program code (called `Code`) and it is to be treated exactly as a `verbatim` block. You can quickly customize plasTeX with this class:

```
class Code(verbatim):
    pass
```

When the plasTeX parser encounters the `Code` environment, it looks up the definition in the Python classes. The environment subclasses the `verbatim` environment, thereby inheriting all the properties needed for plasTeX to parse the contents. No other code is necessary for plasTeX to recognize and parse your text.

The following lines in your template file exploit the power of template aliasing:

```
name: Code
alias: verbatim
```

When the plasTeX renderer encounters the `Code` environment, it finds the definition in the templates file and renders it as if were a `verbatim` environment. No other code is necessary to render the `Code` environment. Obviously this is a simple example; however, the power of inheritance and aliasing even for complicated macros significantly speeds the development of new macro packages and renderers.

3.5 Creating Themes

Themes provide a powerful method to quickly change the look and feel of your HTML or XML documents. Any template directory (a directory specified in the `XHTMLTEMPLATES` or `DocBookTEMPLATES` environment variables) can contain mutiple themes. To create a theme, create a subdirectory called `Themes` which contains a directory of HTML or XML templates for each theme you want to provide. For example, themes included in the plasTeX distribution for the HTML renderer reside in the following directory structure:

```
XHTMLTEMPLATES/
       Themes/
          default/
          minimal/
          plain/
          python/
```

Each of the theme directories contains at least one file called `default-layout.html`. To create your own themes, use the `default/default-layout.html` file as a guide.

4 Getting Involved

While the core parser and document builder are powerful and stable, there is ample opportunity for users to contribute by testing and writing Python classes to further support popular LaTeX packages, and renderers to new output formats.

The XHTML renderer is intended as the basis of all HTML-based renderers; when writing a new HTML-based renderer, it can be subclassed to exploit the power of inheritance. Even for completely different output formats the code in the XHTML renderer can be used as a guide throughout development.

5 References

The following locations of software distributions and reference material can be useful for finding further information.

plasTeX `plastex.sourceforge.net`

Python http://www.python.org/download/

Python Imaging Library http://www.pythonware.com/products/pil/

HTML Help Workshop http://msdn.microsoft.com/en-us/library/ms669985.aspx

JavaHelp http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javahelp/

JabRef http://jabref.sourceforge.net/

DocBook http://www.docbook.org/

DocBook XSL Stylesheets http://docbook.sourceforge.net/release/xsl/current/doc/index.html

XMLMind Editor http://www.xmlmind.com/xmleditor

S5 Slide Show System http://meyerweb.com/eric/tools/s5/

BrlTex http://brltex.sourceforge.net/

SimpleTal http://www.owlfish.com/software/simpleTAL/

Zope Page Templates http://www.zope.org/Documentation/Books/ZopeBook/2_6Edition/AppendixC.stx