Software & Tools

PerlTeX: Defining LaTeX macros using Perl

Scott Pakin

Abstract

Although writing documents with LaTeX is straightforward, *programming* LaTeX to automate repetitive tasks — especially those involving complex string manipulation — can be quite challenging. Many operations that a novice programmer can express easily in a general-purpose programming language cannot be expressed in LaTeX by any but the most experienced LaTeX users. PerlTeX attempts to bridge the worlds of document preparation (LaTeX) and general-purpose programming (Perl) by enabling an author to define LaTeX macros in terms of ordinary Perl code.

1 Introduction

Although TeX is a Turing machine and can therefore express arbitrary computation, the language is not conducive to programming anything sophisticated. As in an assembly language, arithmetic expressions are written in terms of register modifications (e.g., "`\advance\myvar by 3`") and relational expressions involving conjunction and disjunction are constructed from nested comparison operations (e.g., "`\ifnum\myvar>10 \ifnum\myvar<15`"). Loops are expressed in terms of tail-recursive macro evaluation. The only forms of string manipulation are single-token lookahead (`\futurelet`) and macro argument templates that either match a pattern or abort TeX. Finally, there are scalars but no aggregate data types (although these can sometimes be faked with clever use of macro expansion). While the LaTeX kernel and various packages slightly raise the level of programming abstraction, the typical programmer is rapidly frustrated when attempting to code anything nontrivial.

Perl, in contrast, offers a rich programming environment with most of the features one expects from a modern high-level language. However, Perl has no inherent support for document typesetting. For short or highly repetitive documents, it is reasonable to write a Perl script that outputs a `.tex` file and runs it through latex. However, it is generally inconvenient to include a full-length article in its entirety within a Perl script just so it can invoke some simple function which is easier to express in Perl than in LaTeX. Furthermore, a LaTeX-generating

```
\newcount\n
\newcommand{\astsslow}[1]{%
  \n=#1
  \xdef\asts{}%
  \loop\ifnum\n>0 \xdef\asts{\asts*}\advance\n-1
  \repeat}
```

(a) Slow version from *The TeXbook*

```
\newcount\n
\newcommand{\astsfast}[1]{%
  \n=#1
  \begingroup
  \aftergroup\edef\aftergroup\asts\aftergroup{%
    \loop \ifnum\n>0 \aftergroup*\advance\n-1
    \repeat
  \aftergroup}\endgroup}
```

(b) Fast but non-scalable version from *The TeXbook*

```
\newcommand{\asts}{}
\perlnewcommand{\astsperl}[1]
  {'\renewcommand{\asts}{' . '*' x $_[0] . '}'}
```

(c) Fast PerlTeX version

Figure 1: Macro to define `\asts` as a sequence of N asterisks

Perl script supports only one-way communication: Perl can pass information to LaTeX but not the other way around.

In this article, we present PerlTeX, a package that consists of a Perl script (`perltex.pl`) and a LaTeX 2_ε style file (`perltex.sty`). The user simply installs `perltex.pl` in an executable directory and `perltex.sty` in a LaTeX 2_ε style-file directory, incorporates "`\usepackage{perltex}`" into any documents which need PerlTeX's features, and compiles such documents using `perltex.pl` instead of the ordinary latex command. Together, `perltex.pl` and `perltex.sty` give the user the ability to define LaTeX macros in terms of Perl code. Once defined, a PerlTeX macro becomes indistinguishable from any other LaTeX macro. PerlTeX thereby combines LaTeX's typesetting power with Perl's programmability.

1.1 A simple example

A PerlTeX macro definition can be as simple as

```
\perlnewcommand{\hello}{"Hello, world!"}
```

which is essentially equivalent to:

```
\newcommand{\hello}{Hello, world!}
```

```
% Given a list of words, build up a \measurements macro as alternating
% words and word width in points, sorted by order of increasing width.
\perlnewcommand{\splitandmeasure}[1]{
    return
        "\\edef\\measurements{}%\n" .
        join ("",
              map "\\setbox0=\\hbox{$_}%\n" .
                  "\\edef\\measurements{\\measurements\\space $_ \\the\\wd0}%\n",
                  split " ", $_[0]) .
        "\\sortandtabularize{\\measurements}%\n";
}

% Given the \measurements macro produced by \splitandmeasure, output a
% two-column tabular showing each word and its width in points.
\perlnewcommand{\sortandtabularize}[1]{
    %word2width = split " ", $_[0];
    return
        "\\begin{tabular}{|l|r|} \\hline\n" .
        "  \\multicolumn{1}{|c|}{Word} &\n" .
        "  \\multicolumn{1}{c|}{Width} \\\\ \\hline\\hline\n" .
        join ("",
              map ("  $_ & $word2width{$_} \\\\ \\hline\n",
                   sort {$word2width{$a} <=> $word2width{$b}} keys %word2width)) .
        "\\end{tabular}\n";
}
```

Figure 2: A PerlTEX-defined LATEX macro that outputs a table of words sorted by typeset width

(The extra " characters delimit a string constant in Perl.)

To better motivate the use of PerlTEX, consider the first programming challenge in the "Dirty Tricks" appendix of *The TEXbook* [3]: construct a macro that accepts an integer N and defines another macro, `\asts`, to be a sequence of N asterisks. Figure 1(a) presents a LATEX wrapper, `\astsslow`, for the initial *TEXbook* solution. Besides relying on a set of TEX primitives which are unlikely to be familiar to a LATEX user, the code is slow; `\astsslow{10000}` takes over 6 seconds to run on the author's 2.8 GHz Xeon-based workstation.

Figure 1(b) presents a LATEX version of the "fast" solution from *The TEXbook*. `\astsfast` is highly unintuitive; it exploits artifacts of macro expansion and execution that occur when used in the context of the TEX `\aftergroup` primitive. Furthermore, it squanders space on TEX's input and save stacks, limiting the number of asterisks to fewer than 300 when run using the default latex program that ships with teTEX v1.02.

In contrast to *The TEXbook*'s solutions, the PerlTEX solution is fast, scalable, and should be comparatively easy to understand by anyone with basic Perl-programming and LATEX macro-writing skills. Figure 1(c) presents an `\astsperl` macro that takes an argument and returns a `\renewcommand` string which LATEX subsequently evaluates. `\astsperl{10000}` takes less than a second to run on the same 2.8 GHz Xeon system as did the previous macros and uses no TEX primitives, only ordinary LATEX and Perl commands.

1.2 A more complex example

One of PerlTEX's capabilities which is not available with a Perl script that outputs a `.tex` file is the ability to pass data bidirectionally between LATEX and Perl. Suppose, for example, that you wanted to write a macro that accepts a string of text, splits it into its constituent space-separated words, and outputs a table of those words sorted by their typeset width. Neither LATEX nor Perl can easily do this on its own. LATEX can measure word width but cannot easily split a string into words or sort a list; Perl cannot easily determine how wide a word will be when typeset but does have primitives for splitting and sorting strings.

A PerlTEX macro to do the job, named `\splitandmeasure`, is presented in Figure 2. It

```
\edef\measurements{}%
\setbox0=\hbox{How}%
\edef\measurements{\measurements\space How \the\wd0}%
\setbox0=\hbox{now}%
\edef\measurements{\measurements\space now \the\wd0}%
\setbox0=\hbox{brown}%
\edef\measurements{\measurements\space brown \the\wd0}%
\setbox0=\hbox{cow?}%
\edef\measurements{\measurements\space cow? \the\wd0}%
\sortandtabularize{\measurements}%
```

(a) Result of the call to `\splitandmeasure{How now brown cow?}`

```
How 19.44447pt now 17.50003pt brown 26.97227pt
cow? 21.11113pt
```

(b) Final contents of `\measurements` after evaluating the code in Figure 3(a)

```
\begin{tabular}{|l|r|} \hline
  \multicolumn{1}{|c|}{Word} &
  \multicolumn{1}{c|}{Width} \\ \hline\hline
  now & 17.50003pt \\ \hline
  How & 19.44447pt \\ \hline
  cow? & 21.11113pt \\ \hline
  brown & 26.97227pt \\ \hline
\end{tabular}
```

(c) Result of the call to
`\sortandtabularize{\measurements}`

Word	Width
now	17.50003pt
How	19.44447pt
cow?	21.11113pt
brown	26.97227pt

(d) Final typeset table

Figure 3: Overall PerlTeX processing of
`\splitandmeasure{How now brown cow?}`

accepts a string, splits it into words, and writes LaTeX (or more accurately in this case, TeX) code which builds up a `\measurements` macro consisting of alternating words and word widths. This code is followed by a call to a second PerlTeX (helper) macro, `\sortandtabularize`, which accepts a list of alternating words and word widths (i.e., `\measurements`), sorts the list by word width, and outputs a `tabular` environment for LaTeX to typeset.

Figure 3 illustrates the step-by-step operation of `\splitandmeasure`. Processing begins with LaTeX invoking the `\splitandmeasure` macro, causing Perl to output LaTeX code which measures each word (Figure 3(a)). LaTeX then evaluates that code, producing the definition of `\measurements` shown in Figure 3(b) followed by an invocation of `\sortandtabularize`. Control once again passes to Perl, which sorts `\measurements` by word width and outputs a LaTeX `tabular` environment (Figure 3(c)). LaTeX then evaluates the `tabular`, producing the typeset output shown in Figure 3(d).

Macros such as `\splitandmeasure` which pass control from LaTeX to Perl to LaTeX to Perl and back to LaTeX are comparatively easy to implement with PerlTeX — `\splitandmeasure` consists of a single Perl statement; its helper macro, `\sortandtabularize`, consists of only two Perl statements. However, it would be very difficult to implement comparable functionality without the help of PerlTeX.

The rest of this article proceeds as follows. Section 2 highlights some of the design decisions that went into PerlTeX's implementation. We contrast those design decisions to the ones made by similar projects in Section 3. Section 4 describes the mechanisms PerlTeX uses to transfer data betwen LaTeX and Perl. Defining Perl macros in LaTeX was the greatest challenge in implementing PerlTeX and required some fairly sophisticated LaTeX trickery. The solutions that were developed are described in Section 5. By comparison, the Perl side of PerlTeX is comparatively straightforward and is described briefly in Section 6. Section 7 presents some avenues for future enhancements to PerlTeX. Finally, we draw some conclusions in Section 8.

2 Design decisions

There are multiple ways that PerlTeX could have been implemented. The following are the primary alternatives:

- Use the semi-standard "`\write18`" mechanism to invoke the `perl` executable.
- Patch the TeX executable to interface with the Perl interpreter.
- Implement a Perl interpreter in LaTeX.
- Construct macros that enable LaTeX to communicate with an external Perl interpreter.

The final option is the one that was deemed best for PerlTeX. The "`\write18`" approach is a security risk; enabling it (e.g., using the `-shell-escape` command-line option present in some TeX distributions) permits not only PerlTeX but any LaTeX package to execute arbitrary programs on the user's system. Patching TeX is inconvenient for the user, who will need to recompile TeX (plus pdfTeX, ε-

TeX, pdf-ε-TeX, Ω, and any other TeX-based system for which the user wants to add Perl support) then re-dump the LaTeX 2_ε format file for each Perl-enhanced build of TeX. Implementing a Perl interpreter in LaTeX has the advantage of not requiring a separate Perl installation. However, a LaTeX-based Perl interpreter, besides being extremely difficult to implement, would necessarily support only a small subset of Perl, as much of the language cannot be expressed in terms of the mechanisms provided by TeX.

As this article will demonstrate, providing LaTeX-level mechanisms to facilitate communication between LaTeX and an external Perl interpreter enables safe execution of Perl code, ease of installation, compatibility with any underlying TeX implementation, and access to every feature of the Perl language.

3 Related work

PerlTeX is not the first system that attempts to augment LaTeX macro programming with a general-purpose programming language. However, PerlTeX's approach, as outlined in the previous section, makes it unique relative to other, similar systems. Note that many of the following systems support not only LaTeX but other formats as well (e.g., Plain TeX, ConTeXt, and Texinfo); for the purpose of exposition we limit our discussion to LaTeX.

After releasing PerlTeX, the author discovered an existing program written by Alexander Shibakov also called PerlTeX [6]. Unlike the PerlTeX described in this paper, Shibakov's version is implemented as a patch to TeX. That is, the user must recompile TeX (and all its variants) with the PerlTeX patches and re-dump the desired formats. The result is that Perl is more integrated into TeX than is otherwise possible. All code between `\perl` and `\endperl` is executed by Perl. Furthermore, Shibakov's PerlTeX also supports two-way communication between TeX and Perl by enabling code within a `\perl`...`\endperl` block to insert characters and control sequences into the TeX input stream. While Shibakov's PerlTeX works with any TeX format — Plain TeX, LaTeX, ConTeXt, Texinfo, etc. — the PerlTeX described in this paper works only with LaTeX. However, this paper's PerlTeX has the important advantage of not requiring TeX recompilation, which is tedious and may not be possible when using a commercial TeX implementation.

Paraschenko takes a similar approach to Shibakov's with his sTeXme [4], which uses Scheme rather than Perl as the TeX extension language. sTeXme adds a single command to TeX: `\stexme`, which works like `\input` but accepts the name of a Scheme file rather than a TeX or LaTeX file. When the Scheme interpreter evaluates the given file, output procedures such as `newline` and `display` write into the TeX input stream. Two new procedures, `pool-string` and `get-cmd`, provide access to TeX internal state. As with Shibakov's PerlTeX, sTeXme's tight integration with TeX comes at the cost of having to recompile TeX and re-dump all of the format files before the extension language can be used.

TeX2page [7] uses also uses Scheme as a TeX extension language. However, its design is closer to that of (this paper's) PerlTeX than to sTeXme's. TeX2page provides an `\eval` macro which brackets Scheme code. The document is first compiled using the ordinary `latex` executable. As part of that process, `\eval` simply writes its argument to a file. The user then runs `tex2page`, which invokes the Scheme interpreter on the extracted Scheme code and writes the resulting LaTeX code to a file. Finally, the user re-runs `latex` and, on this pass, `\eval` loads the Scheme-produced LaTeX code into the document, where it is typeset normally. Although TeX2page's multi-pass approach supports two-way communication betwen LaTeX and Scheme, it does require an extra run of `tex2page` and an extra run of `latex` for each nesting level. For large documents or heavily nested `\eval` calls, this can be slow and tedious. PerlTeX, in contrast, requires no more `latex` runs than the document would otherwise require.

The idea behind PyTeX [1] is to use Python, not LaTeX, as the document's top-level language. With PyTeX, the user's Python code passes strings to a TeX dæmon [2] to evaluate. PyTeX supports only one-way communication (i.e., Python to LaTeX but not LaTeX to Python). PerlTeX, in contrast, supports two-way communication, which is necessary when writing code in a general-purpose language that requires access to typesetting information such as string widths, page counts, or register contents.

Amrita [5] presents an integration framework based on re-entrant *here* documents which supports communication among a variety of languages such as Perl, Python, LaTeX, Ruby, and POV-Ray. Each language can generate code to be executed by any other language. The result of each execution (which itself may recursively generate code for additional languages) is code to be executed by the parent language. While Amrita is a highly capable system, its power necessarily introduces an extra level of complexity to the user. Relative to the generality of Amrita, PerlTeX's niche is that it enables users to

Table 1: Files used for communication between Perl and LaTeX

Filename	Meaning	Purpose
`\jobname.topl`	"to" (Perl)	LaTeX to Perl communication *also*: signal Perl that `\jobname.frpl` has been read
`\jobname.frpl`	"from" (Perl)	Perl to LaTeX communication
`\jobname.tfpl`	"to flag"	signal Perl that `\jobname.topl` is ready to be read
`\jobname.ffpl`	"from flag"	signal LaTeX that `\jobname.frpl` is ready to be read
`\jobname.dfpl`	"done-with-from-flag flag"	signal LaTeX that Perl is ready for the next transaction

add a few Perl macros to an existing LaTeX document with minimal hassle and without having to buy into a more comprehensive software framework.

4 Communication between LaTeX and Perl

PerlTeX has two main components: a Perl script (`perltex.pl`) and a LaTeX 2ε style file (`perltex.sty`). PerlTeX is invoked by running the command perltex.pl, just as one would run latex. `perltex.pl` itself is fairly simple; essentially, it installs a "server" which executes incoming Perl code and outputs the LaTeX result. More information is provided in Section 6.

`perltex.sty` provides the `\perlnewcommand`, `\perlrenewcommand`, `\perlnewenvironment`, and `\perlrenewenvironment` macros which are analogous to their non-`perl` namesakes but are defined with Perl code instead of LaTeX code in the macro body. When a PerlTeX macro is defined, `perltex.sty` instructs `perltex.pl` to define a corresponding Perl subroutine with the given body. Then, when the macro is invoked, `perltex.sty` instructs `perltex.pl` to execute the subroutine. A similar process is performed when defining PerlTeX environments but involving two behind-the-scenes macros, one for the "begin" code and one for the "end" code.

Almost by necessity, communication between LaTeX and Perl is implemented via the filesystem. TeX provides primitives for creating new files, opening existing files, reading and writing files, and closing files, but no other mechanisms that can be used to communicate with entities outside of TeX (excluding `\write18`, which has security implications, as mentioned in Section 2). TeX returns a failure code when trying to open a nonexistent file; this condition can safely be tested from within TeX.

The primary challenge in transferring data via the filesystem is detecting when a file is no longer being written to. This challenge needs to be addressed both on the Perl side of the transfer and on the LaTeX side. The solution that PerlTeX takes is to

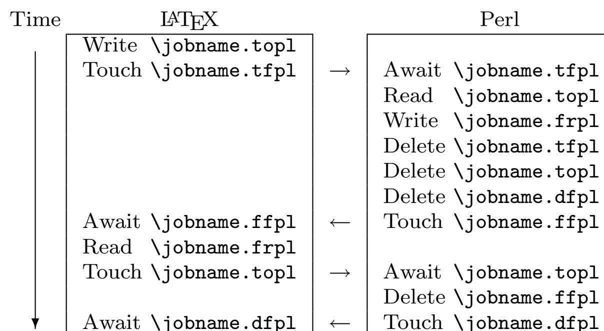

Figure 4: LaTeX/Perl communication protocol

employ some auxiliary "flag" files that signal when an associated file is complete. Table 1 describes the complete set of files used for communication between Perl and LaTeX.

The communication protocol proper, which is illustrated in Figure 4, is necessarily complex because it needs to work around two important limitations of the TeX system:

1. TeX lacks a mechanism for deleting files.
2. The latex executable — at least the version shipped with the teTeX TeX distribution — is prone to crash when opening a file for input while an external process is in the midst of deleting that file. (Recall that testing if a file exists means opening the file for input and checking for success.)

If it were not for those limitations, the protocol would require only one flag file and half as many steps.

The `\jobname.frpl` file contains ordinary LaTeX code that simply gets `\input` into the document. `\jobname.topl`, in contrast, contains not only Perl code but also some metadata that helps offload some string manipulation from LaTeX to Perl. Consider passing the LaTeX string

```
In C it's \texttt{printf("Hello!")}.
```

as an argument to a function declared with `\perlnewcommand`. Because the string contains both

DEF
⟨*unique tag*⟩
⟨*macro name*⟩
⟨*unique tag*⟩
⟨*Perl code*⟩

(a) Define

USE
⟨*unique tag*⟩
⟨*macro name*⟩
⟨*unique tag*⟩
#1
⟨*unique tag*⟩
#2
⟨*unique tag*⟩
#3
⋮
#⟨*last*⟩

(b) Invoke

Figure 5: Data written to `\jobname.topl` to define or invoke a Perl subroutine

single and double quote characters, every occurrence of at least one type of quote will need to be backslash-escaped for Perl. Rather than do this on the LaTeX side, `perltex.sty` sends the string as-is to `perltex.pl`, which automatically quotes the string while reading it from `\jobname.topl`. The implication is that `perltex.sty` cannot pass raw Perl code to `perltex.pl` to evaluate.

Hence, `\jobname.topl` needs contain some metadata telling `perltex.pl` what to do with the rest of `\jobname.topl`'s contents. This metadata is of one of two types. When `\perlnewcommand` or any of the other PerlTeX macros is invoked, `perltex.sty` sends `perltex.pl` the information shown in Figure 5(a). Then, when a macro defined by one of PerlTeX's `\perl`... macros is called, `perltex.sty` sends `perltex.pl` the information shown in Figure 5(b). In Figure 5, ⟨*unique tag*⟩ refers to a sequence of 20 letters that `perltex.pl` generates randomly at initialization time and passes to `perltex.sty` via the latex command line. The ⟨*unique tag*⟩ is used as a separator, so `perltex.pl` knows where one piece of information ends and the next one begins. ⟨*macro name*⟩ is the name of the macro to be defined or used. `perltex.pl` defines a Perl subroutine named ⟨*macro name*⟩ but with the leading backslash replaced with "`latex_`". The subroutine body contains ⟨*Perl code*⟩ verbatim. When a PerlTeX-defined macro is invoked, `perltex.sty` passes `perltex.pl` the name of the macro plus all of the arguments as expanded LaTeX code.

Figures 6 and 7 present a more concrete expression of a LaTeX/Perl file transfer. Figure 6(a) shows the contents of the `\jobname.topl` file that LaTeX writes as part of the `\perlnewcommand` invocation presented previously in Figure 1(c); Figure 6(b) shows the contents of the `\jobname.frpl` file that Perl writes in response. Figure 7(a) shows the contents of the `\jobname.topl` file that LaTeX writes while executing "`\astsperl{10}`" and Figure 7(b) shows the `\jobname.frpl` file that Perl writes in response to that.

```
DEF
TKOUVLRCDIVSVSIZVHFI
\astsperl
TKOUVLRCDIVSVSIZVHFI
'\renewcommand{\asts}{' . '*' x $_[0] . '}'
```

(a) Macro definition (`\jobname.topl`)

```
\endinput
```

(b) Result of macro definition (`\jobname.frpl`)

Figure 6: LaTeX/Perl communication associated with the code in Figure 1(c)

```
USE
TKOUVLRCDIVSVSIZVHFI
\astsperl
TKOUVLRCDIVSVSIZVHFI
10
```

(a) Macro invocation (`\jobname.topl`)

```
\renewcommand{\asts}{**********}\endinput
```

(b) Result of macro invocation (`\jobname.frpl`)

Figure 7: LaTeX/Perl communication associated with an invocation of "`\asts{10}`"

Expansion is a tricky issue in PerlTeX's design and, in fact, is handled differently in PerlTeX v1.1 than in earlier versions of PerlTeX. The challenge is that Perl cannot evaluate LaTeX code; it requires all subroutine parameters to be ASCII strings. Consider this invocation of some PerlTeX macro `\mymacro`:

```
\mymacro{Hello from Perl\noexpand\TeX!}
```

How should `\mymacro`'s argument be passed to Perl? (1) Unexpanded, as

```
Hello from Perl\noexpand\TeX!
```

or (2) partly expanded, as

```
Hello from Perl\TeX!
```

or (3) fully expanded, as

```
Hello from PerlT\kern -.1667em\lower .5ex
\hbox {E}\kern -.125emX\@!
```

?

The first alternative makes PerlTEX macros behave differently from LATEX macros, which generally execute their arguments. The other two alternatives lead to unexpected behavior in cases like `\mymacro{\def\foo{world}Hello, \foo!}`, which cause latex to abort with an `Undefined control sequence` error as it tries to expand the not-yet-defined `\foo` control word which immediately follows the non-expandable `\def` control word. Execution is not an option because an invocation like `\mymacro{\mbox{Oops}}` would need to pass a box to Perl, which cannot practically be done.

PerlTEX's approach (as of version 1.1) is to partially expand macro arguments but with `\protect` mapped to `\noexpand` and with `\begin` and `\end` marked as non-expandable. In this approach, robust macros (such as many of the ones provided by LATEX) are not expanded while fragile macros (such as many of the ones defined by a user) are expanded. For example, the following sequence will write "LATEX is nice" to the typeset output, which is a fairly intuitive result:

```
\newcommand{\adjective}{nice}
\perlnewcommand{\identity}[1]{$_[0]}
\identity{\LaTeX{} is \adjective.}
```

5 Defining Perl macros from LATEX

From a LATEX programming perspective, there are two primary challenges that need to be overcome in order to implement `\perlnewcommand`, `\perlrenewcommand`, `\perlnewenvironment`, and `\perlrenewenvironment`:

1. How can syntactically incorrect LATEX code be stored and manipulated?
2. How can a LATEX macro iterate over a variable number of macro arguments?

A solution to the former question is required because `\perlnewcommand`, etc. need to write Perl code to a file. Syntactically correct Perl code is unlikely also to be syntactically correct LATEX code. For example, Perl associative arrays are prefixed with the LATEX comment character, "`%`"; Perl scalars are prefixed with "`$`", which introduces math mode in LATEX; and Perl uses "\" to escape special characters in strings and create variable references while LATEX expects a valid control sequence to follow. The difficulty, therefore, is in enabling a LATEX macro to manipulate one of its arguments while neither expanding nor evaluating it.

A solution to the latter question, how to iterate over macro arguments, is required because each macro argument must be passed to Perl (via the `\jobname.topl` file). Just as with `\newcommand`, a macro defined by `\perlnewcommand` accepts a user-defined number of arguments (e.g., `\perlnewcommand{\mymac}[5]{...}`). However, TEX requires that macro arguments be referenced by a literal number (e.g., "`#3`"); variable argument numbers (e.g., "`#\argnum`") result in a TEX error. The challenge is to construct a loop that iterates over a variable number of arguments, writing each argument to a file, yet does not use a variable to reference any arguments.

5.1 Storing non-LATEX code

The final argument to `\perlnewcommand` is a block of Perl code which will almost certainly cause errors if evaluated by LATEX. Storing this Perl code in a macro is similar to outputting non-LATEX code using the `\verb` macro. The difference is that `\verb` does not need to store its argument.

The solution taken by `perltex.sty` works as follows. First, `\perlnewcommand` is defined to read one fewer argument than actually needed; the Perl code is considered the first piece of text *following* `\perlnewcommand`'s argument list. `\perlnewcommand`'s last action is to begin a new variable scope with `\begingroup` and, within that scope, set the TEX category codes for all characters to "other" (i.e., 12) to prevent "`%`", "`$`", "\", and so forth from being treated specially. The only exceptions are that "`{`" and "`}`" retain their original meanings so that TEX brace-counting will indicate when the Perl code has ended. Also, the end-of-line character is made significant because it has meaning within a Perl string.

The next task involves figuring out how to store the Perl code following `\perlnewcommand` and then reset all of the category codes back to their prior values. The trick that `perltex.sty` relies upon is the TEX `\afterassignment` primitive, which specifies a command to execute after the next assignment takes place. The following are the last two lines of `\perlnewcommand`'s implementation:

```
\afterassignment\plmac@havecode
\global\plmac@perlcode
```

In other words, the `\plmac@havecode` macro should be executed after the next assignment. Then, `\perlnewcommand` ends with an assignment to the global token register `\plmac@perlcode`. The right-hand side of the assignment is the block of Perl code, which is already within a pair of curly braces, as required by a token-register assignment. After the assignment takes place, control automatically transfers to the `\plmac@havecode` macro. Before

changing category codes, \perlnewcommand began a new scope with \begingroup; \plmac@havecode resets the category codes by executing the matching \endgroup. The result is that the Perl code is stored unevaluated in the \plmac@perlcode token register, as desired, and LaTeX can continue compiling the user's document.

```
\def\plmac@havecode{%
            ⋮
  \let\plmac@hash=\relax
  \plmac@argnum=\@ne
  \loop
    \ifnum\plmac@numargs<\plmac@argnum
    \else
      \edef\plmac@body{%
        \plmac@body
        \plmac@sep\plmac@tag\plmac@sep
        \plmac@hash\plmac@hash
          \number\plmac@argnum}%
      \advance\plmac@argnum by \@ne
  \repeat
  \let\plmac@hash=##%
            ⋮
}
```

Figure 8: perltex.sty code that iterates over macro arguments

5.2 Iterating over macro arguments

One limitation of TeX's macro-processing facility is that macro arguments must be referred to by a literal argument number. Hence, "#2" is acceptable but \newcommand*{\whicharg}{2} followed inside a macro definition by "#\whicharg" results in an "Illegal parameter number" error. Even worse, the error occurs at macro-definition time; even if a macro containing "#\whicharg" is never invoked it will still cause TeX to report an error and abort.

Fortunately, the aforementioned limitation is not insurmountable but it does require a bit of trickery. The solution is to replace "#" with a control sequence that is let-bound to \relax. TeX does not expand such control sequences. After the macro is defined, the control sequence can then be let-bound to #, making it work as desired.

There are two caveats to this approach. First, # can be used only within a macro definition; hence, the macro definition must itself be within a macro definition in order for the let-binding to succeed. Second, when the macro is executed, # must be followed by a literal argument number. The let-binding trickery merely delays the literal-number check from definition time to execution time — but this is sufficient for the purpose of accessing a variable-numbered macro argument. Careful use of \edef and \noexpand can then make it possible to iterate over macro arguments, as desired.

Figure 8 presents an excerpt of code from perltex.sty which constructs a \plmac@body macro that references in turn each argument from 1 up to \plmac@numargs. In this code, \plmac@hash is the placeholder for the # character and \plmac@argnum is the argument number, which varies from 1 to \plmac@numargs. In each iteration of the loop, \plmac@body is redefined as the concatenation of its old value, a carriage-return character (\plmac@sep), a unique tag as described in Section 4, another carriage-return character, and "##" (doubled because the \edef is nested within another macro) followed immediately by the argument number. Only at the end of the loop, after \plmac@body has its final contents, is \plmac@hash set to an actual # character (written as "##" because it occurs within the definition of \plmac@havecode).

6 Processing Perl code

While perltex.sty contains rather complex LaTeX code, perltex.pl contains fairly straightforward Perl code. perltex.pl's basic structure is as follows:

1. Parse the command line.
2. Create a secure sandbox in which to execute Perl code coming from the document.
3. Spawn a latex process, passing it a variety of macro definitions in addition to the name of the user's LaTeX source file.
4. Repeatedly poll for new Perl code to execute, execute that code in the secure sandbox, and return the (LaTeX) result.

perltex.pl uses the Safe and Opcode modules to create a secure sandbox in which to execute code. The idea behind a sandbox is that it limits the types of code that can be executed. Code deemed too dangerous to run (e.g., an attempt to delete a file or to kill a running process) produces a run-time error. Sandboxing the code passed from LaTeX to perltex.pl enables users to build a PerlTeX document created by a third party without having to worry about it containing malicious or otherwise destructive Perl code. The default set of sandbox permissions is Opcode's ":browse" permissions, which enable the core Perl language features such as arrays, loops, variable assignment, and function definitions, but forbid creating and opening files, spawning child processes, communicating

with other processes, and performing most other input/output functions. A command-line option selectively enables individual functions or groups of functions. (Another command-line option disables sandboxing altogether, although this is not generally recommended.)

After spawning latex (alternatively, pdflatex, elatex, vlatex, or any other LaTeX compiler), `perltex.pl` makes that the *foreground* process, leaving itself in the background. Doing so makes it possible for latex to run interactively (e.g., when encountering an error), which it could not do as easily as a background process.

Finally, `perltex.pl` enters a loop in which it polls the filesystem for incoming Perl code, executes the code, and returns the (LaTeX) result via the filesystem. The LaTeX/Perl communication protocol is as described in Section 4. The loop terminates when the latex process exits.

7 Future work

Although PerlTeX performs its tasks reliably, there are a variety of avenues for future expansion and enhancement, mostly suggested by PerlTeX users. First, while PerlTeX's `\perlnewcommand`, `\perlrenewcommand`, `\perlnewenvironment`, and `\perlrenewenvironment` macros provide a faithful Perl analogue to LaTeX's command- and environment-defining macros, a useful addition would be a way to execute Perl code directly. Such a feature would be useful when writing Perl code that is executed only once, such as program initialization or generation of a particularly unique list, table, or equation.

The performance of the PerlTeX implementation could be improved. Although filesystem-based communication between LaTeX and Perl is portable, file activity — especially over a remote filesystem — can be a performance bottleneck when compiling PerlTeX-intensive documents.

One alternative to using the filesystem is to communicate using standard input and standard output. There are two challenges in implementing this approach. First, TeX lacks a mechanism to explicitly flush standard output. Depending on how latex is implemented, a deadlock can result if LaTeX sends a command to Perl and blocks waiting for the result while Perl never sees the command because the standard-output buffers have not been flushed. Second, maintaining support for user interaction (e.g., to diagnose error conditions) may be complicated if PerlTeX needs to compete with the user for control over standard input and standard output.

A second alternative to filesystem-based communication is to use named pipes, an internal operating-system data structure for interprocess communication. A problem with named pipes is that they are not as portable as files; not every operating system supports named pipes or implements them in the file namespace (i.e., they might be accessed via a different interface, making them inaccessible to TeX). In addition, while Perl can create named pipes, TeX cannot. This restriction may limit their usefulness in the context of PerlTeX.

Finally, a meaningful follow-on to PerlTeX would be an ⟨*anything*⟩TeX system. Most of PerlTeX's magic is in the extension-language-independent `perltex.sty` file. The Perl-specific `perltex.pl` file performs only simple file and string manipulation and should easily be portable to any other programming language. Users could then write LaTeX macros in the language (or languages) with which they are most comfortable.

8 Conclusions

As this article has demonstrated, PerlTeX takes a practical, portable approach to augmenting TeX's typesetting finesse with Perl's power in string manipulation and general-purpose programming. The importance of PerlTeX's design — a Perl "server" that accepts Perl input and produces LaTeX output — is that it enables two-way communication between LaTeX and Perl. As Section 1.2 demonstrated, LaTeX can invoke a Perl subroutine which can produce LaTeX code that itself invokes a Perl subroutine which outputs some final LaTeX code. Support for this dynamic usage model is a clear advantage of PerlTeX over a custom Perl script which generates a static LaTeX document. By exploiting Perl's sandboxing features, users can compile PerlTeX documents written by others without fear of their system being harmed by malicious Perl code.

A key design decision in PerlTeX's implementation was to keep the perl and latex programs largely decoupled. The advantage of decoupling the two programs is that PerlTeX remains compatible with every underlying TeX variant — TeX, pdfTeX, ε-TeX, pdf-ε-TeX, Ω, etc. — and does not require the user to recompile the base TeX executable or re-dump a LaTeX 2_ε format. The disadvantages are that Perl cannot directly access TeX's internals and that TeX can communicate with external applications only via the filesystem (not counting the security-risk-prone `\write18` mechanism or by revoking user control over standard input and standard output). This article has presented a filesystem-based communication protocol that en-

ables LaTeX and Perl to communicate even though the two systems are asymmetric in terms of the types of file operations each supports. Even though TeX cannot, for example, delete a file, the protocol ensures correct behavior, including in the presence of mutually recursive LaTeX and Perl routines such as those utilized in Section 1.2.

Finally, this paper presented solutions to two challenging LaTeX puzzles: how to store and manipulate syntactically incorrect LaTeX code; and, how to iterate over a variable number of macro arguments. The former problem is solved using a token-register assignment at the end of a macro call with `\afterassignment` used to transfer control to a continuation macro. The latter problem is solved using a control sequence bound to `\relax` while defining a macro but bound to `#` afterwards. Neither of those techniques is specific to PerlTeX; advanced LaTeX users can readily employ them in their own macros.

In summary, PerlTeX combines Perl's fortes of string manipulation, regular-expression processing, and general programmability with LaTeX's typesetting capabilities. A few lines of PerlTeX can easily replace their much longer, more complex equivalent coded in ordinary LaTeX. PerlTeX thereby makes sophisticated LaTeX macro programming more accessible to the novice and more convenient for the advanced user.

The PerlTeX distribution is available for download from CTAN at `http://www.ctan.org/tex-archive/macros/latex/contrib/perltex/`.

9 Acknowledgments

The author would like to thank all of the people who have provided feedback, suggestions, and bug reports for PerlTeX including Andrei Alexandrescu, José Pedro Oliveira, Fernando P. Schapachnik, Ivo Welch, James Quirk, Michele Dondi, Hans Fredrik Nordhaug, and everyone else who helped make PerlTeX a success. Also, thanks to James Quirk for critiquing the PerlTeX examples originally used in this paper's Introduction section.

References

[1] Jonathan Fine. PyTeX: Python plus TeX. `http://www.pytex.org/`.

[2] Jonathan Fine. TeX as a callable function. In *Proceedings of the 13th European and 10th Polish TeX Conference (EuroBachoTeX 2002)*, pages 26–35, Bachotek, Poland, April 29–May 3, 2002. Available from `http://www.pytex.org/doc/euro2002.pdf`.

[3] Donald E. Knuth. *The TeXbook*. Addison-Wesley, 1986. ISBN 0-201-13447-0.

[4] Oleg Paraschenko. sTeXme: TeX + Scheme. `http://stexme.sourceforge.net/`.

[5] James J. Quirk. Programming dynamic LaTeX documents. In *Proceedings of the 24th Annual Meeting and Conference of the TeX Users Group (TUG 2003)*, Waikoloa, Hawai'i, July 20–25, 2003. Slides available from `http://www.tug.org/tug2003/bulletin/highlights/slides/2_Monday/4_Quirk/4_quirk.pdf`.

[6] Alexander Shibakov. PerlTeX—a fusion of Perl and TeX via Web2C. `http://www.math.tntech.edu/alex/`.

[7] Dorai Sitaram. TeX2page. `http://www.ccs.neu.edu/home/dorai/tex2page/`.

◇ Scott Pakin
4975 S. Sol
Los Alamos, NM 87544-3794, USA
scott+tb@pakin.org
http://www.pakin.org/~scott