# PerlTEX:
# 펄 코드로 LATEX 매크로 정의하기[*]

Defining LATEX macros in terms of Perl code[†]

Scott Pakin
scott+pt@pakin.org

2007/12/10

**요 약**

PerlTEX은 펄 스크립트 `perltex.pl`와 LATEX 2$\varepsilon$ 스타일 파일 `perltex.sty`로 구성되어 있다. 이들은 함께 사용되어, 사용자가 LATEX 매크로를 펄 코드로 작성할 수 있게 해 준다. 펄 매크로는 한번 정의되기만 하면 다른 LATEX 매크로와 구별되지 않는다. 이를 통해 PerlTEX은 LATEX의 조판 능력과 펄의 프로그래밍 능력을 결합한다.

## 제 1 절 서론

TEX은 전문가 수준의 조판 시스템이다. 하지만, 그 프로그래밍 언어는 가장 단순한 텍스트 교체 외에는 그 무엇에도 쓰기 어렵다. TEX을 위한 매크로 패키지로 가장 널리 쓰인다는 LATEX도 TEX 프로그래밍을 단순화하기 위해 한 일은 거의 없다.

펄(Perl)은 일반 목적 프로그래밍 언어로 그 특기는 텍스트 조작이다. 하지만, 조판이라고 하는 것은 전혀 지원하지 않는다.

PerlTEX의 목표는 이들 두 세계에 다리를 놓는 것이다. 문서를 구성하는 것을 기본적으로는 LATEX 기반으로 하면서 약간의 Perl을 포함시키는 것을 가능하게 해 준다. PerlTEX은 펄 코드를 LATEX 문서에 빈틈없이 끼워넣어서 사용자가 매크로를 정의할 때 TEX이나 LATEX 코드 대신에 펄 코드를 사용할 수 있도록 해 준다.

예를 하나 들어보자. 낱말들의 집합인 문장이 주어졌을 때 그 순서를 뒤집는 매크로를 정의할 필요가 있다고 하자. 아주 단순한 것 같지만, LATEX 사용자 중 그

[*]번역: slomo (http://faq.ktug.or.kr/faq/slomo), 2008년 1월 12일

[†]This document corresponds to PerlTEX v1.7, dated 2007/12/10.

런 매크로를 작성할 수 있을 만큼 TEX 언어에 충분히 숙달된 이는 극히 소수에 불과하다. 하지만, 단어 순서 바꾸기를 펄로 작성하라고 한다면 아주 쉽다. 문자열을 `split` 명령으로 잘라서 어절의 목록을 만들고 `reverse` 명령으로 목록을 뒤집은 후에, `join` 명령으로 다시 합치면 된다. 아래 `\reversewords` 매크로를 PerlTEX 에서 정의하는 방법이 있다.

```
\perlnewcommand{\reversewords}[1]{join " ", reverse split " ", $_[0]}
```

그리고 나서, 문서에 "`\reversewords{Try doing this without Perl!}`" 라고 쓰면 "Perl! without this doing Try" 라는 텍스트가 만들어진다. 간단하지 않은가?

예를 하나 더 들어 보자. 어떻게 하면 LATEX에서 주어진 문자열에서 하위문자열을 추출할 수 있을지 생각해 보라. 이때 문자열과 함께 시작점과 길이가 함께 제공된다. 펄에는 내장된 `substr` 함수가 있고 PerlTEX은 이것을 LATEX에 내보내는 것을 간단하게 해결해 준다.

```
\perlnewcommand{\substr}[3]{substr $_[0], $_[1], $_[2]}
```

`\substr` 매크로는 여타 LATEX 매크로와 똑같이 사용될 수 있다. 그리고 펄의 `substr` 함수처럼 간단하기도 하다.

```
\newcommand{\str}{superlative}
A sample substring of ``\str'' is ``\substr{\str}{2}{4}''.
```

$$\Downarrow$$

A sample substring of "superlative" is "perl".

좀 더 복잡한 예를 제시하기 위해서, 반복적인 행렬을 생성할 때 펄 코드를 이용하면 LATEX 명령을 이용하는 것보다 얼마나 더 쉬워지는지 살펴보자.

```
\perlnewcommand{\hilbertmatrix}[1]{
  my $result = '
\[
\renewcommand{\arraystretch}{1.3}
';
  $result .= '\begin{array}{' . 'c' x $_[0] . "}\n";
  foreach $j (0 .. $_[0]-1) {
    my @row;
    foreach $i (0 .. $_[0]-1) {
      push @row, ($i+$j) ? (sprintf '\frac{1}{%d}', $i+$j+1) : '1';
    }
    $result .= join (' & ', @row) . " \\\\\n";
  }
  $result .= '\end{array}
\]';
  return $result;
}

\hilbertmatrix{20}
```

$$\Downarrow$$

$$\begin{array}{ccccccccccccccc}
1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} \\
\frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} \\
\frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} \\
\frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} \\
\frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19} \\
\frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19} & \frac{1}{20} \\
\frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19} & \frac{1}{20} & \frac{1}{21} \\
\frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19} & \frac{1}{20} & \frac{1}{21} & \frac{1}{22} \\
\frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19} & \frac{1}{20} & \frac{1}{21} & \frac{1}{22} & \frac{1}{23} \\
\frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19} & \frac{1}{20} & \frac{1}{21} & \frac{1}{22} & \frac{1}{23} & \frac{1}{24} \\
\frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19} & \frac{1}{20} & \frac{1}{21} & \frac{1}{22} & \frac{1}{23} & \frac{1}{24} & \frac{1}{25} \\
\frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19} & \frac{1}{20} & \frac{1}{21} & \frac{1}{22} & \frac{1}{23} & \frac{1}{24} & \frac{1}{25} & \frac{1}{26} \\
\frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19} & \frac{1}{20} & \frac{1}{21} & \frac{1}{22} & \frac{1}{23} & \frac{1}{24} & \frac{1}{25} & \frac{1}{26} & \frac{1}{27} \\
\frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19} & \frac{1}{20} & \frac{1}{21} & \frac{1}{22} & \frac{1}{23} & \frac{1}{24} & \frac{1}{25} & \frac{1}{26} & \frac{1}{27} & \frac{1}{28} \\
\frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19} & \frac{1}{20} & \frac{1}{21} & \frac{1}{22} & \frac{1}{23} & \frac{1}{24} & \frac{1}{25} & \frac{1}{26} & \frac{1}{27} & \frac{1}{28} & \frac{1}{29}
\end{array}$$

예제로 살펴본 `\perlnewcommand`, `\perlrenewcommand`와 함께, PerlTEX 은 `\perlnewenvironment`와 `\perlrenewenvironment` 매크로를 제공한다. 환경(environment)에서도 펄 코드를 사용할 수 있는 것이다. 아래 나오는 예는

spreadsheet 환경인데, tabular 환경을 생성하고 미리 정의된 헤더 열을 만들어 준다. 이 예를 PerlTEX 없이 구현하는 것은 훨씬 더 힘들 것이다.

```
\newcounter{ssrow}
\perlnewenvironment{spreadsheet}[1]{
  my $cols = $_[0];
  my $header = "A";
  my $tabular = "\\setcounter{ssrow}{1}\n";
  $tabular .= '\newcommand*{\rownum}{\thessrow\addtocounter{ssrow}{1}}' . "\n";
  $tabular .= '\begin{tabular}{@{}r|*{' . $cols . '}{r}@{}}' . "\n";
  $tabular .= '\\multicolumn{1}{@{}c}{} &' . "\n";
  foreach (1 .. $cols) {
    $tabular .= "\\multicolumn{1}{c";
    $tabular .= '@{}' if $_ == $cols;
    $tabular .= "}{" . $header++ . "}";
    if ($_ == $cols) {
      $tabular .= " \\\\ \\cline{2-" . ($cols+1) . "}"
    }
    else {
      $tabular .= " &";
    }
    $tabular .= "\n";
  }
  return $tabular;
}{
  return "\\end{tabular}\n";
}

\begin{center}
  \begin{spreadsheet}{4}
    \rownum &  1 &  8 & 10 & 15 \\
    \rownum & 12 & 13 &  3 &  6 \\
    \rownum &  7 &  2 & 16 &  9 \\
    \rownum & 14 & 11 &  5 &  4
  \end{spreadsheet}
\end{center}
```

$$\Downarrow$$

|   | A | B | C | D |
|---|---|---|---|---|
| 1 | 1 | 8 | 10 | 15 |
| 2 | 12 | 13 | 3 | 6 |
| 3 | 7 | 2 | 16 | 9 |
| 4 | 14 | 11 | 5 | 4 |

## 제 2 절 사용법

PerlTEX을 사용하기 위해서 필요한 것은 두 가지이다. 첫째, 문서의 전문(preamble)에 반드시 “\usepackage{perltex}”이라는 행이 포함되어야 한다는 것이다. 이 행이 있어야 \perlnewcommand, \perlrenewcommand, \perlnewenvironment, \perlrenewenvironment 매크로를 이용할 수 있다. 둘째, LATEX 문서를 컴파일할 때 perltex.pl 스크립트를 써야 한다는 것이다.

### 2.1 펄 매크로 정의하기와 재정의하기

\perlnewcommand
\perlrenewcommand
\perlnewenvironment
\perlrenewenvironment
\perldo

perltex.sty에는 다섯가지 매크로가 정의되어 있다: \perlnewcommand, \perlrenewcommand, \perlnewenvironment, \perlrenewenvironment, \perldo. 이들 중 앞의 넷이 동작하는 방식은 정확하게 각각의 LATEX 2$_\varepsilon$ 대응 매크로와 같다. 즉, 각각 \newcommand, \renewcommand \newenvironment, \renewenvironment 에 대응한다. 단지 매크로의 본체가 펄 코드로 되어 있고 이를 통해 동적으로 LATEX 코드를 생성한다는 점이 다를 뿐이다. perltex.sty은 선택 인자(optional argument)도 지원하고 명령의 별표 형태(starred form)도 지원한다(예를들어, \perlnewcommand*, \perlrenewcommand*, \perlnewenvironment*, \perlrenewenvironment*). \perldo는 펄 코드 구역을 즉시 실행시키기 위한 것으로 매크로나 환경을 (재)정의하지 않는다.

PerlTEX으로 정의된 매크로나 환경은 펄 서브루틴으로 변환된다. 서브루틴의 이름은 매크로/환경의 이름을 따르면서 그 앞에 “latex_”이라는 접두사가 붙는다. 예를 들어, PerlTEX으로 정의된 LATEX 매크로의 이름이 \myMacro라면 내부적으로 만들어지는 펄 서브루틴의 이름은 latex_myMacro가 된다. 매크로의 인자(argument)는 서브루틴의 인자로 변환된다. LATEX 매크로의 #1 인자는 펄에서 $_[0]으로 참조될 수 있고 #2는 $_[1]로 참조될 수 있는 식이다.

유효한 펄 코드라면 무엇이든 매크로의 본체에 사용될 수 있다. 하지만, PerlTEX은 펄 코드를 안전 모래상자(secure sandbox)에서 실행시킨다는 점을 기억하라. 이 말의 뜻은, 잠재적으로 위험성을 가지고 있는 펄 연산, 그러니까 unlink, rmdir, system 같은 것을 사용하면 런타임 에러가 난다는 뜻이다. (어쨌든, 이 안전 점검 기능을 끄는 것도 가능하다. 설명은 2.2절 참조.) 안전 모래상자가 있음으로 인해 다른 사람이 만든 PerlTEX 문서를, 그들이 내 컴퓨터에 무슨 짓을 하는 것 아닌가 걱정하지 않고, 안전하게 빌드할 수 있다.

하나의 모래상자가 전체 latex 실행 동안 사용된다. 이 말은 \perlnewcommand 에 의해 정의된 여러 매크로들이 서로 서로 호출할 수 있다는 뜻이다. 또, 전역

변수가 여러 매크로 호출에 걸쳐 유지된다는 뜻이기도 하다.

```
\perlnewcommand{\setX}[1]{$x = $_[0]; return ""}
\perlnewcommand{\getX}{'$x$ was set to ' . $x . '.'}
\setX{123}
\getX
\setX{456}
\getX
\perldo{$x = 789}
\getX
```

$$\Downarrow$$

$x$ was set to 123. $x$ was set to 456. $x$ was set to 789.

매크로의 인자는 펄로 넘겨지기 전에 LaTeX에 의해 확장(expansion)될 수 있다. 아래 매크로 정의를 보자. 인자가 `\begin{verbatim*}`...`\end{verbatim*}` 사이에 싸여있다.

```
\perlnewcommand{\verbit}[1]{
  "\\begin{verbatim*}\n$_[0]\n\\end{verbatim*}\n"
}
```

따라서, "`\verbit{\TeX}`"을 호출하면 "`\TeX`"의 *확장*, 그러니까 "`T\kern -.1667em\lower .5ex\hbox {E}\kern -.125emX\spacefactor \@m`"이 조판될 것이고 이건 좀 기대했던 것과는 다르다. 해결책은 `\noexpand`를 사용하는 것이다: `\verbit{\noexpand\TeX}` ⇒ `\TeX`. "Robust" macros as well as `\begin` and `\end` are implicitly preceded by `\noexpand`.

## 2.2 perltex.pl 실행하기

이어지는 페이지들은 `perltex.pl` 프로그램의 설명서를 그대로 찍은 것이다. 설명서의 핵심 부부만 발췌해서 보고 싶으면 `perltex.pl`을 실행시킬 때 `--help` 옵션을 주면 된다. 펄의 다양한 pod2⟨*something*⟩ 도구를 사용해서 전체 프로그램 문서를 다양한 형식으로 생성할 수 있다. LaTeX, HTML, 일반 텍스트, 유닉스 man-page 형식이 지원된다. 예를 들어, 다음 명령을 이용하면 `perltex.pl`에서 유닉스 맨 페이지를 만들어 낼 수 있다.

```
pod2man --center=" " --release=" " perltex.pl > perltex.1
```

## NAME

perltex — LaTeX 매크로를 펄 코드로 정의할 수 있게 해준다

## SYNOPSIS

perltex [–**help**] [–**latex**=*program*] [–[**no**]**safe**] [–**permit**=*feature*] [–**makesty**] [*latex options*]

## DESCRIPTION

LaTeX은, 그 기초가 되는 TeX 조판 시스템을 통해, 아름답게 조판된 문서를 만들어내지만, 그 매크로 언어는 프로그램하기 어렵다. 특히, 복잡한 문자열 조작을 위한 지원은 크게 부족하다. 펄은 널리 사용되는 일반 목적 프로그래밍 언어로 문자열 조작에 강하다. 하지만, 펄에는 조판 능력 같은 것은 없다.

분명히 펄의 프로그램 능력은 LaTeX의 조판 능력을 보완할 수 있을 것이다. **perltex**은 이 두 시스템이 공생할 수 있도록 해주는 도구이다. 사용자가 해야 할 일은 LaTeX 문서를 컴파일할 때 **latex** 대신에 **perltex**을 사용하는 것뿐이다. (**perltex**은 실제로는 **latex**의 wrapper이기 때문에 **latex**의 기능이 사라지거나 하지는 않는다.) 문서의 전문(preamble)에 `\usepackage{perltex}`를 포함시키기만 하면 `\perlnewcommand`과 `\perlrenewcommand` 매크로를 사용할 수 있게 된다. 이것들의 작동 방식은 LaTeX의 `\newcommand`, `\renewcommand`와 동일하다. 다만 매크로의 몸체에 LaTeX 코드 대신에 펄 코드를 가진다는 점만 다르다.

## OPTIONS

**perltex**이 받아들이는 명령행 옵션은 다음과 같다:

**--help**

기본적인 사용법에 관한 정보를 보여준다.

**--latex=*program***

**latex** 대신 사용할 프로그램을 지정한다. 예를 들어, `--latex=pdflatex` 옵션을 주면 해당 문서를 조판할 때 **pdflatex**을 사용한다.

**--[no]safe**

모래상자를 켜거나 끈다. 기본적으로 **--safe** 옵션이 사용되고, **perltex**은 `\perlnewcommand` 또는 `\perlrenewcommand` 매크로에 정의된 펄 코드를 보호된 환경에서 실행시킨다. 파일 접근이나 외부 프로그램 실행 같은 "위험한" 동작은 제한된다. **--nosafe** 옵션을 주면 해당 LaTeX 문서는 "백지 위임장"을 받는다. 어떤 펄 코드라도 실행시킬 수 있고, 따라서 사용자의 파일들에 위해를 가할 수도 있다. 더 자세한 정보는 *Safe* 참조.

**--permit=*feature***

특정한 펄 연산이 수행되는 것을 허가한다. **--permit** 옵션은 명령행에서 여러번 사용될 수 있고, **perltex** 모래상자를 더 정밀하게 제어할 수 있다. 더 상세한 정보는 *Opcode* 참조.

**--makesty**

이 옵션을 주면 *noperltex.sty*라는 이름의 LaTeX 스타일 파일이 생성된다. 그리고 해당 문서의 `\usepackage{perltex}` 행을 `\usepackage{noperltex}`으로 바꾸면 PerlTeX 없이 동일한 출력을 얻을 수 있게 된다. 이것은 PerlTeX을 설치하지 않은 사람들에게 문서를 배포할 때 유용하다. 단점은 *noperltex.sty* 스타일 파일은 그것을 만들 때 사용한 바로 그 문서에만 소용이 있다는 것이다. 만약에 해당 문서의 LaTeX 매크로 정의나 매크로 호출 부분을 바꾸었다면 **perltex**을 **--makesty** 옵션을 주어서 다시 돌려야 한다.

이 옵션들에 뒤이어서 사용한 옵션은 **latex** (또는 `--latex` 옵션으로 지정한) 프로그램에 컴파일할 *.tex* 파일의 이름과 함께 전달된다.

## EXAMPLES

가장 단순한 사용법은 **perltex**을 **latex**처럼 사용하는 것이다:

```
perltex myfile.tex
```

컴파일러로 **latex** 대신에 **pdflatex**을 사용하려면, **--latex** 옵션을 주면 된다:

```
perltex --latex=pdflatex myfile.tex
```

LaTeX이 "`trapped by operation mask`" 에러를 내고 지금 컴파일하려는 *.tex*이 악의적인 펄 코드를 실행하지 않는다고 확신하는 경우에는 (예를 들어, 스스로 작성한 문서인 경우) **perltex**의 안전 메커니즘을 **--nosafe** 옵션으로 끌 수 있다.

```
perltex --nosafe myfile.tex
```

다음 명령은 해당 문서에 **perltex**의 기본 권한인 **:browse**에 파일을 읽고 `time` 명령을 호출할 수 있는 권한만 추가한다.

```
perltex --permit=:browse --permit=:filesys_open
  --permit=time myfile.tex
```

## ENVIRONMENT

**perltex**은 다음 환경 변수들을 유효한 것으로 받아들인다:

**PERLTEX**

LaTeX 컴파일러의 파일명을 지정한다. LaTeX 컴파일러는 기본값으로 "`latex`"으로 되어있다. `PERLTEX` 환경 변수는 이 값을 덮어쓴다. 그리고, 최종 결정은 명령행 옵션의 **--latex**에 의해 이루어진다.

## FILES

컴파일하는 파일이 *jobname.tex*이라고 했을 때, **perltex**은 다음과 같은 파일들을 사용한다:

***jobname.lgpl***

펄에 의해 작성된 로그 파일; 펄 매크로를 디버깅할 때 도움이 된다.

***jobname.topl***

이 파일에 담긴 정보는 LaTeX에서 펄로 전달된 것이다

***jobname.frpl***

이 파일에 담긴 정보는 펄에서 LaTeX으로 전달된 것이다

***jobname.tfpl***

이 "flag" 파일이 있으면 *jobname.topl*에 유효한 데이터가 들어있다는 뜻이다.

***jobname.ffpl***

이 "flag" 파일이 있으면 *jobname.frpl*에 유효한 데이터가 들어있다는 뜻이다.

***jobname.dfpl***

이 "flag" 파일이 있으면 *jobname.ffpl*가 삭제되었다는 뜻이다.

***noperltex-#.tex***

각 파일은 *noperltex.sty*에 의해 생성된 것으로 PerlTEX 매크로 호출을 위한 것이다.

## NOTES

**perltex**의 모래상자의 기본 설정은 *Opcode*에서 "`:browse`"라고 불리는 것이다.

## SEE ALSO

latex(1), pdflatex(1), perl(1), Safe(3pm), Opcode(3pm)

## AUTHOR

Scott Pakin, *scott+pt@pakin.org*

## 제 3 절 구현

단순히 PerlTEX을 사용하는 데에만 관심이 있다면 3 절은 건너뛰어도 된다. 이 절에서는 PerlTEX 소스 코드 전체를 소개하고 있다. 이 절에 기본적으로 관심을 가질 사람은 PerlTEX을 확장하려고 하거나 펄이 아닌 다른 언어를 사용할 수 있도록 수정하려는 사람일 것이다.

제 3 절은 크게 두 부분으로 나뉜다. 제 3.1 절은 `perltex.sty`의 소스 코드, 즉 LATEX의 측면에서 PerlTEX을 소개하고, 제 3.2 절은 `perltex.pl`의 소스 코드, 즉 펄의 측면에서 PerlTEX을 소개한다. 전체적으로 PerlTEX은 코드의 양이 적은 편이다. `perltex.sty`는 226 행의 LATEX 코드로 이루어져 있고 `perltex.pl`는 겨우 302 행의 펄 코드로 이루어져 있다. `perltex.pl`는 아주 직선적인 방식의 펄 코드이기 때문에 펄 프로그래밍을 충분히 다루는 사람이라면 누구나 어렵지 않게 이해할 수 있을 것이다. 반면에 `perltex.sty`는 트릭을 쓴 LATEX 코드를 좀 포함하고 있고 LaTeX 프로그래밍을 이미 직접해 본 사람이 아니라면 아마도 이해할 수 없을 것이다. 다행스럽게도 독자들을 위해서 코드에 풍부한 주석을 달았으니 의기충만한 구루도 뭔가 배울 것이 있을 것이다.

`perltex.sty`과 `perltex.pl` 소스 코드에 대한 문서화 뒤에는 PerlTEX을 포팅하여 펄이 아닌 다른 백 엔드 언어를 사용하는 방법을 위한 관련된 몇가지 힌트가 제시될 것이다 (3.3 절).

### 3.1 `perltex.sty`

Although I've written a number of LATEX packages, `perltex.sty` was the most challenging to date. The key things I needed to learn how to do include the following:

1. storing brace-matched—but otherwise not valid LATEX—code for later use
2. iterating over a macro's arguments

Storing non-LATEX code in a variable involves beginning a group in an argumentless macro, fiddling with category codes, using `\afterassignment` to specify a continuation function, and storing the subsequent brace-delimited tokens in the input stream into a token register. The continuation function, which also takes no arguments, ends the group begun in the first function and proceeds using the correctly `\catcode`d token register. This technique appears in `\plmac@haveargs` and `\plmac@havecode` and in a simpler form (i.e., without the need for storing the argument) in `\plmac@write@perl` and `\plmac@write@perl@i`.

Iterating over a macro's arguments is hindered by TEX's requirement that "`#`" be followed by a number or another "`#`". The technique I discovered (which is used by the Texinfo source code) is first to `\let` a variable be `\relax`, thereby making it unexpandable, then to define a macro that uses that variable followed by a loop variable, and finally to expand the loop variable and `\let` the `\relax`ed variable be "`#`" right before invoking the macro. This technique appears in `\plmac@havecode`.

I hope you find reading the `perltex.sty` source code instructive. Writing it certainly was.

#### 3.1.1 Package initialization

PerlTEX defines six macros that are used for communication between Perl and LATEX. `\plmac@tag` is a string of characters that should never occur within one of the user's macro names, macro arguments, or macro bodies. `perltex.pl` therefore defines `\plmac@tag` as a long string of random uppercase letters. `\plmac@tofile` is the name of a file used for communication from LATEX to Perl. `\plmac@fromfile` is the name of a file used for communication from Perl to LATEX. `\plmac@toflag` signals that `\plmac@tofile` can be read safely. `\plmac@fromflag` signals that `\plmac@fromfile` can be read safely. `\plmac@doneflag` signals that `\plmac@fromflag` has been deleted. Table 1 lists all of these variables along with the value assigned to each by `perltex.pl`.

표 1: Variables used for communication between Perl and LATEX

| Variable | Purpose | `perltex.pl` assignment |
|---|---|---|
| `\plmac@tag` | `\plmac@tofile` field separator | (20 random letters) |
| `\plmac@tofile` | LATEX → Perl communication | `\jobname.topl` |
| `\plmac@fromfile` | Perl → LATEX communication | `\jobname.frpl` |
| `\plmac@toflag` | `\plmac@tofile` synchronization | `\jobname.tfpl` |
| `\plmac@fromflag` | `\plmac@fromfile` synchronization | `\jobname.ffpl` |
| `\plmac@doneflag` | `\plmac@fromflag` synchronization | `\jobname.dfpl` |

`\ifplmac@have@perltex`
`\plmac@have@perltextrue`
`\plmac@have@perltexfalse`

The following block of code checks the existence of each of the variables listed in Table 1 plus `\plmac@pipe`, a Unix named pipe used for to improve performance. If any variable is not defined, `perltex.sty` gives an error message and—

as we shall see on page 26—defines dummy versions of `\perl[re]newcommand` and `\perl[re]newenvironment`.

```
\newif\ifplmac@have@perltex
\plmac@have@perltextrue
\@ifundefined{plmac@tag}{\plmac@have@perltexfalse}{}
\@ifundefined{plmac@tofile}{\plmac@have@perltexfalse}{}
\@ifundefined{plmac@fromfile}{\plmac@have@perltexfalse}{}
\@ifundefined{plmac@toflag}{\plmac@have@perltexfalse}{}
\@ifundefined{plmac@fromflag}{\plmac@have@perltexfalse}{}
\@ifundefined{plmac@doneflag}{\plmac@have@perltexfalse}{}
\@ifundefined{plmac@pipe}{\plmac@have@perltexfalse}{}
\ifplmac@have@perltex
\else
  \PackageError{perltex}{Document must be compiled using perltex}
    {Instead of compiling your document directly with latex, you need
     to\MessageBreak use the perltex script.  \space perltex sets up
     a variety of macros needed by\MessageBreak the perltex
     package as well as a listener process needed for\MessageBreak
     communication between LaTeX and Perl.}
\fi
```

### 3.1.2 Defining Perl macros

PerlTEX defines five macros intended to be called by the author. Section 3.1.2 details the implementation of two of them: `\perlnewcommand` and `\perlrenewcommand`. (Section 3.1.3 details the implementation of the next two, `\perlnewenvironment` and `\perlrenewenvironment`; and, Section 3.1.4 details the implementation of the final macro, `\perldo`.) The goal is for these two macros to behave *exactly* like `\newcommand` and `\renewcommand`, respectively, except that the author macros they in turn define have Perl bodies instead of LATEX bodies.

The sequence of the operations defined in this section is as follows:

1. The user invokes `\perl[re]newcommand`, which stores `\[re]newcommand` in `\plmac@command`. The `\perl[re]newcommand` macro then invokes `\plmac@newcommand@i` with a first argument of "`*`" for `\perl[re]newcommand*` or "`!`" for ordinary `\perl[re]newcommand`.

2. `\plmac@newcommand@i` defines `\plmac@starchar` as "`*`" if it was passed a "`*`" or ⟨*empty*⟩ if it was passed a "`!`". It then stores the name of the user's macro in `\plmac@macname`, a `\write`able version of the name in `\plmac@cleaned@macname`, and the macro's previous definition (needed by `\perlrenewcommand`) in `\plmac@oldbody`. Finally, `\plmac@newcommand@i` invokes `\plmac@newcommand@ii`.

3. `\plmac@newcommand@ii` stores the number of arguments to the user's macro (which may be zero) in `\plmac@numargs`. It then invokes `\plmac@newcommand@iii@opt` if the first argument is supposed to be optional or `\plmac@newcommand@iii@no@opt` if all arguments are supposed to be required.

4. `\plmac@newcommand@iii@opt` defines `\plmac@defarg` as the default value of the optional argument. `\plmac@newcommand@iii@no@opt` defines it as ⟨*empty*⟩. Both functions then call `\plmac@haveargs`.

5. `\plmac@haveargs` stores the user's macro body (written in Perl) verbatim in `\plmac@perlcode`. `\plmac@haveargs` then invokes `\plmac@havecode`.

6. By the time `\plmac@havecode` is invoked all of the information needed to define the user's macro is available. Before defining a LaTeX macro, however, `\plmac@havecode` invokes `\plmac@write@perl` to tell `perltex.pl` to define a Perl subroutine with a name based on `\plmac@cleaned@macname` and the code contained in `\plmac@perlcode`. Figure 1 illustrates the data that `\plmac@write@perl` passes to `perltex.pl`.

| |
|---|
| `DEF` |
| `\plmac@tag` |
| `\plmac@cleaned@macname` |
| `\plmac@tag` |
| `\plmac@perlcode` |

그림 1: Data written to `\plmac@tofile` to define a Perl subroutine

7. `\plmac@havecode` invokes `\newcommand` or `\renewcommand`, as appropriate, defining the user's macro as a call to `\plmac@write@perl`. An invocation of the user's LaTeX macro causes `\plmac@write@perl` to pass the information shown in Figure 2 to `perltex.pl`.

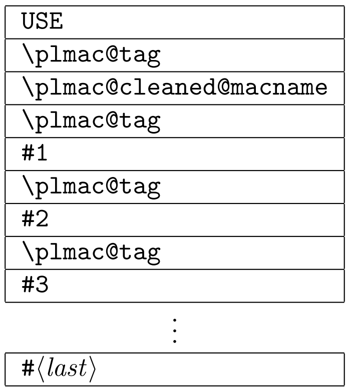


그림 2: Data written to `\plmac@tofile` to invoke a Perl subroutine

8. Whenever `\plmac@write@perl` is invoked it writes its argument verbatim to `\plmac@tofile`; `perltex.pl` evaluates the code and writes `\plmac@fromfile`; finally, `\plmac@write@perl` `\input`s `\plmac@fromfile`.

An example might help distinguish the myriad macros used internally by `perltex.sty`. Consider the following call made by the user's document:

```
\perlnewcommand*{\example}[3][frobozz]{join("---", @_)}
```

Table 2 shows how `perltex.sty` parses that command into its constituent components and which components are bound to which `perltex.sty` macros.

표 2: Macro assignments corresponding to an sample `\perlnewcommand*`

| Macro | Sample definition | |
|---|---|---|
| `\plmac@command` | `\newcommand` | |
| `\plmac@starchar` | `*` | |
| `\plmac@macname` | `\example` | |
| `\plmac@cleaned@macname` | `\example` | (catcode 11) |
| `\plmac@oldbody` | `\relax` | (presumably) |
| `\plmac@numargs` | `3` | |
| `\plmac@defarg` | `frobozz` | |
| `\plmac@perlcode` | `join("---", @_)` | (catcode 11) |

`\perlnewcommand`
`\perlrenewcommand`
`\plmac@command`
`\plmac@next`

`\perlnewcommand` and `\perlrenewcommand` are the first two commands exported to the user by `perltex.sty`. `\perlnewcommand` is analogous to `\newcommand` except that the macro body consists of Perl code instead of LaTeX code. Likewise, `\perlrenewcommand` is analogous to `\renewcommand` except that the macro body consists of Perl code instead of LaTeX code. `\perlnewcommand` and `\perlrenewcommand` merely define `\plmac@command` and `\plmac@next` and invoke `\plmac@newcommand@i`.

```
\def\perlnewcommand{%
  \let\plmac@command=\newcommand
  \let\plmac@next=\relax
  \@ifnextchar*{\plmac@newcommand@i}{\plmac@newcommand@i!}%
}
```

```
\def\perlrenewcommand{%
  \let\plmac@next=\relax
  \let\plmac@command=\renewcommand
  \@ifnextchar*{\plmac@newcommand@i}{\plmac@newcommand@i!}%
}
```

`\plmac@newcommand@i`
`\plmac@starchar`
`\plmac@macname`
`\plmac@oldbody`
`\plmac@cleaned@macname`

If the user invoked `\perl`[re]`newcommand*` then `\plmac@newcommand@i` is passed a "`*`" and, in turn, defines `\plmac@starchar` as "`*`". If the user invoked `\perl`[re]`newcommand` (no "`*`") then `\plmac@newcommand@i` is passed a "`!`" and, in turn, defines `\plmac@starchar` as ⟨*empty*⟩. In either case, `\plmac@newcommand@i` defines `\plmac@macname` as the name of the user's macro, `\plmac@cleaned@macname` as a `\write`able (i.e., category code 11) version of `\plmac@macname`, and `\plmac@oldbody` and the previous definition of the user's macro. (`\plmac@oldbody` is needed by `\perlrenewcommand`.) It then invokes `\plmac@newcommand@ii`.

```
\def\plmac@newcommand@i#1#2{%
  \ifx#1*%
    \def\plmac@starchar{*}%
  \else
    \def\plmac@starchar{}%
  \fi
  \def\plmac@macname{#2}%
  \let\plmac@oldbody=#2\relax
  \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
    \expandafter\string\plmac@macname}%
  \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%]
}
```

\plmac@newcommand@ii
\plmac@numargs

\plmac@newcommand@i invokes \plmac@newcommand@ii with the number of arguments to the user's macro in brackets. \plmac@newcommand@ii stores that number in \plmac@numargs and invokes \plmac@newcommand@iii@opt if the first argument is to be optional or \plmac@newcommand@iii@no@opt if all arguments are to be mandatory.

```
\def\plmac@newcommand@ii[#1]{%
  \def\plmac@numargs{#1}%
  \@ifnextchar[{\plmac@newcommand@iii@opt}
               {\plmac@newcommand@iii@no@opt}%]
}
```

\plmac@newcommand@iii@opt
\plmac@newcommand@iii@no@opt
\plmac@defarg

Only one of these two macros is executed per invocation of \perl[re]newcommand, depending on whether or not the first argument of the user's macro is an optional argument. \plmac@newcommand@iii@opt is invoked if the argument is optional. It defines \plmac@defarg to the default value of the optional argument. \plmac@newcommand@iii@no@opt is invoked if all arguments are mandatory. It defines \plmac@defarg as \relax. Both \plmac@newcommand@iii@opt and \plmac@newcommand@iii@no@opt then invoke \plmac@haveargs.

```
\def\plmac@newcommand@iii@opt[#1]{%
  \def\plmac@defarg{#1}%
  \plmac@haveargs
}
\def\plmac@newcommand@iii@no@opt{%
  \let\plmac@defarg=\relax
  \plmac@haveargs
}
```

\plmac@perlcode
\plmac@haveargs

Now things start to get tricky. We have all of the arguments we need to define the user's command so all that's left is to grab the macro body. But there's a catch: Valid Perl code is unlikely to be valid LaTeX code. We therefore have to read the macro body in a \verb-like mode. Furthermore, we actually need to *store* the macro body in a variable, as we don't need it right away.

The approach we take in \plmac@haveargs is as follows. First, we give all "special" characters category code 12 ("other"). We then indicate that the carriage return character (control-M) marks the end of a line and that curly braces retain their normal meaning. With the aforementioned category-code definitions, we now have to store the next curly-brace-delimited fragment of text, end the current group to reset all category codes to their previous value, and continue processing the user's macro definition. How do we do that? The answer is to assign the upcoming text fragment to a token register (`\plmac@perlcode`) while an `\afterassignment` is in effect. The `\afterassignment` causes control to transfer to `\plmac@havecode` right after `\plmac@perlcode` receives the macro body with all of the "special" characters made impotent.

```
\newtoks\plmac@perlcode
```

```
\def\plmac@haveargs{%
  \begingroup
    \let\do\@makeother\dospecials
    \catcode`\^^M=\active
    \newlinechar`\^^M
    \endlinechar=`\^^M
    \catcode`\{=1
    \catcode`\}=2
    \afterassignment\plmac@havecode
    \global\plmac@perlcode
}
```

Control is transfered to `\plmac@havecode` from `\plmac@haveargs` right after the user's macro body is assigned to `\plmac@perlcode`. We now have everything we need to define the user's macro. The goal is to define it as "`\plmac@write@perl{`⟨*contents of Figure 2*⟩`}`". This is easier said than done because the number of arguments in the user's macro is not known statically, yet we need to iterate over however many arguments there are. Because of this complexity, we will explain `\plmac@perlcode` piece-by-piece.

`\plmac@sep` Define a character to separate each of the items presented in Figures 1 and 2. Perl will need to strip this off each argument. For convenience in porting to languages with less powerful string manipulation than Perl's, we define `\plmac@sep` as a carriage-return character of category code 11 ("letter").

```
{\catcode`\^^M=11\gdef\plmac@sep{^^M}}
```

`\plmac@argnum` Define a loop variable that will iterate from `1` to the number of arguments in the user's function, i.e., `\plmac@numargs`.

```
\newcount\plmac@argnum
```

\plmac@havecode Now comes the final piece of what started as a call to `\perl`[re]`newcommand`. First, to reset all category codes back to normal, `\plmac@havecode` ends the group that was begun in `\plmac@haveargs`.

```
\def\plmac@havecode{%
  \endgroup
```

\plmac@define@sub We invoke `\plmac@write@perl` to define a Perl subroutine named after `\plmac@cleaned@macname`. `\plmac@define@sub` sends Perl the information shown in Figure 1 on page 13.

```
  \edef\plmac@define@sub{%
    \noexpand\plmac@write@perl{DEF\plmac@sep
      \plmac@tag\plmac@sep
      \plmac@cleaned@macname\plmac@sep
      \plmac@tag\plmac@sep
      \the\plmac@perlcode
    }%
  }%
  \plmac@define@sub
```

\plmac@body The rest of `\plmac@havecode` is preparation for defining the user's macro. (LaTeX 2$_\varepsilon$'s `\newcommand` or `\renewcommand` will do the actual work, though.) `\plmac@body` will eventually contain the complete (LaTeX) body of the user's macro. Here, we initialize it to the first three items listed in Figure 2 on page 14 (with intervening `\plmac@sep`s).

```
  \edef\plmac@body{%
    USE\plmac@sep
    \plmac@tag\plmac@sep
    \plmac@cleaned@macname
  }%
```

\plmac@hash Now, for each argument `#1`, `#2`, …, `#\plmac@numargs` we append a `\plmac@tag` plus the argument to `\plmac@body` (as always, with a `\plmac@sep` after each item). This requires more trickery, as TeX requires a macro-parameter character ("`#`") to be followed by a literal number, not a variable. The approach we take, which I first discovered in the Texinfo source code (although it's used by LaTeX and probably other TeX-based systems as well), is to `\let`-bind `\plmac@hash` to `\relax`. This makes `\plmac@hash` unexpandable, and because it's not a "`#`", TeX doesn't complain. After `\plmac@body` has been extended to include `\plmac@hash1`, `\plmac@hash2`, ..., `\plmac@hash\plmac@numargs`, we then `\let`-bind `\plmac@hash` to `##`, which TeX lets us do because we're within a macro definition (`\plmac@havecode`). `\plmac@body` will then contain `#1`, `#2`, ..., `#\plmac@numargs`, as desired.

```
\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=##%
```

`\plmac@define@command` We're ready to execute a `\[re]newcommand`. Because we need to expand many of our variables, we `\edef` `\plmac@define@command` to the appropriate `\[re]newcommand` call, which we will soon execute. The user's macro must first be `\let`-bound to `\relax` to prevent it from expanding. Then, we handle two cases: either all arguments are mandatory (and `\plmac@defarg` is `\relax`) or the user's macro has an optional argument (with default value `\plmac@defarg`).

```
\expandafter\let\plmac@macname=\relax
\ifx\plmac@defarg\relax
  \edef\plmac@define@command{%
    \noexpand\plmac@command\plmac@starchar{\plmac@macname}%
    [\plmac@numargs]{%
      \noexpand\plmac@write@perl{\plmac@body}%
    }%
}%
\else
  \edef\plmac@define@command{%
    \noexpand\plmac@command\plmac@starchar{\plmac@macname}%
    [\plmac@numargs][\plmac@defarg]{%
```

```
      \noexpand\plmac@write@perl{\plmac@body}%
    }%
  }%
  \fi
```

The final steps are to restore the previous definition of the user's macro—we had set it to `\relax` above to make the name unexpandable—then redefine it by invoking `\plmac@define@command`. Why do we need to restore the previous definition if we're just going to redefine it? Because `\newcommand` needs to produce an error if the macro was previously defined and `\renewcommand` needs to produce an error if the macro was *not* previously defined.

`\plmac@havecode` concludes by invoking `\plmac@next`, which is a no-op for `\perlnewcommand` and `\perlrenewcommand` but processes the end-environment code for `\perlnewenvironment` and `\perlrenewenvironment`.

```
  \expandafter\let\plmac@macname=\plmac@oldbody
  \plmac@define@command
  \plmac@next
}
```

### 3.1.3 Defining Perl environments

Section 3.1.2 detailed the implementation of `\perlnewcommand` and `\perlrenewcommand`. Section 3.1.3 does likewise for `\perlnewenvironment` and `\perlrenewenvironment`, which are the Perl-bodied analogues of `\newenvironment` and `\renewenvironment`. This section is significantly shorter than the previous because `\perlnewenvironment` and `\perlrenewenvironment` are largely built atop the macros already defined in Section 3.1.2.

`\perlnewenvironment`
`\perlrenewenvironment`
`\plmac@command`
`\plmac@next`

`\perlnewenvironment` and `\perlrenewenvironment` are the remaining two commands exported to the user by `perltex.sty`. `\perlnewenvironment` is analogous to `\newenvironment` except that the macro body consists of Perl code instead of LaTeX code. Likewise, `\perlrenewenvironment` is analogous to `\renewenvironment` except that the macro body consists of Perl code instead of LaTeX code. `\perlnewenvironment` and `\perlrenewenvironment` merely define `\plmac@command` and `\plmac@next` and invoke `\plmac@newenvironment@i`.

The significance of `\plmac@next` (which was let-bound to `\relax` for `\perl`[re]`newcommand` but is let-bound to `\plmac@end@environment` here) is that a LaTeX environment definition is really two macro definitions: `\`⟨*name*⟩ and `\end`⟨*name*⟩. Because we want to reuse as much code as possible the idea is to define the "begin" code as one macro, then inject—by way of `plmac@next`—a call to `\plmac@end@environment`, which defines the "end" code as a second macro.

```
\def\perlnewenvironment{%
  \let\plmac@command=\newcommand
  \let\plmac@next=\plmac@end@environment
  \@ifnextchar*{\plmac@newenvironment@i}{\plmac@newenvironment@i!}%
}
```

```
\def\perlrenewenvironment{%
  \let\plmac@command=\renewcommand
  \let\plmac@next=\plmac@end@environment
  \@ifnextchar*{\plmac@newenvironment@i}{\plmac@newenvironment@i!}%
}
```

`\plmac@newenvironment@i`
`\plmac@starchar`
`\plmac@envname`
`\plmac@macname`
`\plmac@oldbody`
`\plmac@cleaned@macname`

The `\plmac@newenvironment@i` macro is analogous to `\plmac@newcommand@i`; see the description of `\plmac@newcommand@i` on page 15 to understand the basic structure. The primary difference is that the environment name (`#2`) is just text, not a control sequence. We store this text in `\plmac@envname` to facilitate generating the names of the two macros that constitute an environment definition. Note that there is no `\plmac@newenvironment@ii`; control passes instead to `\plmac@newcommand@ii`.

```
\def\plmac@newenvironment@i#1#2{%
  \ifx#1*%
    \def\plmac@starchar{*}%
  \else
    \def\plmac@starchar{}%
  \fi
  \def\plmac@envname{#2}%
  \expandafter\def\expandafter\plmac@macname\expandafter{\csname#2\endcsname}%
  \expandafter\let\expandafter\plmac@oldbody\plmac@macname\relax
  \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
    \expandafter\string\plmac@macname}%
  \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%]
}
```

`\plmac@end@environment`
`\plmac@next`
`\plmac@macname`
`\plmac@oldbody`
`\plmac@cleaned@macname`

Recall that an environment definition is a shortcut for two macro definitions:

`\`⟨*name*⟩ and `\end`⟨*name*⟩ (where ⟨*name*⟩ was stored in `\plmac@envname` by `\plmac@newenvironment@i`). After defining `\`⟨*name*⟩, `\plmac@havecode` transfers control to `\plmac@end@environment` because `\plmac@next` was let-bound to `\plmac@end@environment` in `\perl`[`re`]`newenvironment`.

`\plmac@end@environment`'s purpose is to define `\end`⟨*name*⟩. This is a little tricky, however, because LaTeX's `\`[`re`]`newcommand` refuses to (re)define a macro whose name begins with "`end`". The solution that `\plmac@end@environment` takes is first to define a `\plmac@end@macro` macro then (in `plmac@next`) let-bind `\end`⟨*name*⟩ to it. Other than that, `\plmac@end@environment` is a combined and simplified version of `\perlnewenvironment`, `\perlrenewenvironment`, and `\plmac@newenvironment@i`.

```
\def\plmac@end@environment{%
  \expandafter\def\expandafter\plmac@next\expandafter{\expandafter
    \let\csname end\plmac@envname\endcsname=\plmac@end@macro
    \let\plmac@next=\relax
  }%
  \def\plmac@macname{\plmac@end@macro}%
  \expandafter\let\expandafter\plmac@oldbody\csname end\plmac@envname\endcsname
  \expandafter\def\expandafter\plmac@cleaned@macname\expandafter{%
    \expandafter\string\plmac@macname}%
  \@ifnextchar[{\plmac@newcommand@ii}{\plmac@newcommand@ii[0]}%]
}
```

### 3.1.4 Executing top-level Perl code

The macros defined in Sections 3.1.2 and 3.1.3 enable an author to inject subroutines into the Perl sandbox. The final PerlTeX macro, `\perldo`, instructs the Perl sandbox to execute a block of code outside of all subroutines. `\perldo`'s implementation is much simpler than that of the other author macros because `\perldo` does not have to process subroutine arguments. Figure 3 illustrates the data that gets written to `plmac@tofile` (indirectly) by `\perldo`.

`\perldo` Execute a block of Perl code and pass the result to LaTeX for further processing. This code is nearly identical to that of Section 3.1.2's `\plmac@haveargs` but ends by invoking `\plmac@have@run@code` instead of `\plmac@havecode`.

```
\def\perldo{%
```

| |
|---|
| RUN |
| \plmac@tag |
| *Ignored* |
| \plmac@tag |
| \plmac@perlcode |

그림 3: Data written to `\plmac@tofile` to execute Perl code

```
  \begingroup
    \let\do\@makeother\dospecials
    \catcode`\^^M=\active
    \newlinechar`\^^M
    \endlinechar=`\^^M
    \catcode`\{=1
    \catcode`\}=2
    \afterassignment\plmac@have@run@code
    \global\plmac@perlcode
}
```

`\plmac@have@run@code` `\plmac@run@code`

Pass a block of code to Perl to execute. `\plmac@have@run@code` is identical to `\plmac@havecode` but specifies the `RUN` tag instead of the `DEF` tag.

```
\def\plmac@have@run@code{%
  \endgroup
  \edef\plmac@run@code{%
    \noexpand\plmac@write@perl{RUN\plmac@sep
      \plmac@tag\plmac@sep
      N/A\plmac@sep
      \plmac@tag\plmac@sep
      \the\plmac@perlcode
    }%
  }%
  \plmac@run@code
}
```

### 3.1.5 Communication between LaTeX and Perl

As shown in the previous section, when a document invokes `\perl`[re]`newcommand` to define a macro, `perltex.sty` defines the macro in terms of a call to `\plmac@write@perl`. In this section, we learn how `\plmac@write@perl` operates.

At the highest level, LaTeX-to-Perl communication is performed via the filesystem. In essence, LaTeX writes a file (`\plmac@tofile`) corresponding to the information in either Figure 1 or Figure 2; Perl reads the file, executes the code within it, and writes a `.tex` file (`\plmac@fromfile`); and, finally, LaTeX reads and executes the new `.tex` file. However, the actual communication protocol is a bit more involved than that. The problem is that Perl needs to know when LaTeX has finished writing Perl code and LaTeX needs to know when Perl has finished writing LaTeX code. The solution involves introducing three extra files—`\plmac@toflag`, `\plmac@fromflag`, and `\plmac@doneflag`—which are used exclusively for LaTeX-to-Perl synchronization.

There's a catch: Although Perl can create and delete files, LaTeX can only create them. Even worse, LaTeX (more specifically, teTeX, which is the TeX distribution under which I developed PerlTeX) cannot reliably poll for a file's *non*existence; if a file is deleted in the middle of an `\immediate\openin`, `latex` aborts with an error message. These restrictions led to the regrettably convoluted protocol illustrated in Figure 4. In the figure, "Touch" means "create a zero-length file"; "Await" means "wait until the file exists"; and, "Read", "Write", and "Delete" are defined as expected. Assuming the filesystem performs these operations in a sequentially consistent order (not necessarily guaranteed on all filesystems, unfortunately), PerlTeX should behave as expected.

| Time | LaTeX | | Perl |
|---|---|---|---|
| | Write `\plmac@tofile` | | |
| | Touch `\plmac@toflag` | → | Await `\plmac@toflag` |
| | | | Read `\plmac@tofile` |
| | | | Write `\plmac@fromfile` |
| | | | Delete `\plmac@toflag` |
| | | | Delete `\plmac@tofile` |
| | | | Delete `\plmac@doneflag` |
| | Await `\plmac@fromflag` | ← | Touch `\plmac@fromflag` |
| | Read `\plmac@fromfile` | | |
| | Touch `\plmac@tofile` | → | Await `\plmac@tofile` |
| | | | Delete `\plmac@fromflag` |
| ↓ | Await `\plmac@doneflag` | ← | Touch `\plmac@doneflag` |

그림 4: LaTeX-to-Perl communication protocol

`\plmac@await@existence`
`\ifplmac@file@exists`
`\plmac@file@existstrue`
`\plmac@file@existsfalse`

The purpose of the `\plmac@await@existence` macro is to repeatedly check the existence of a given file until the file actually exists. For convenience, we use $\LaTeX\,2_\varepsilon$'s `\IfFileExists` macro to check the file and invoke `\plmac@file@existstrue` or `\plmac@file@existsfalse`, as appropriate.

As a performance optimization we `\input` a named pipe. This causes the `latex` process to relinquish the CPU until the `perltex` process writes data (always just "`\endinput`") into the named pipe. On systems that don't support persistent named pipes (e.g., Microsoft Windows), `\plmac@pipe` is an ordinary file containing only "`\endinput`". While reading that file is not guaranteed to relinquish the CPU, it should not hurt the performance or correctness of the communication protocol between LaTeX and Perl.

```
\newif\ifplmac@file@exists
```

```
\newcommand{\plmac@await@existence}[1]{%
  \input\plmac@pipe
  \loop
    \IfFileExists{#1}%
                 {\plmac@file@existstrue}%
                 {\plmac@file@existsfalse}%
    \ifplmac@file@exists
    \else
  \repeat
}
```

`\plmac@outfile` We define a file handle for `\plmac@write@perl@i` to use to create and write `\plmac@tofile` and `\plmac@toflag`.

```
\newwrite\plmac@outfile
```

`\plmac@write@perl` `\plmac@write@perl` begins the LaTeX-to-Perl data exchange, following the protocol illustrated in Figure 4. `\plmac@write@perl` prepares for the next piece of text in the input stream to be read with "special" characters marked as category code 12 ("other"). This prevents LaTeX from complaining if the Perl code contains invalid LaTeX (which it usually will). `\plmac@write@perl` ends by passing control to `\plmac@write@perl@i`, which performs the bulk of the work.

```
\newcommand{\plmac@write@perl}{%
  \begingroup
    \let\do\@makeother\dospecials
    \catcode`\^^M=\active
    \newlinechar`\^^M
    \endlinechar=`\^^M
    \catcode`\{=1
    \catcode`\}=2
    \plmac@write@perl@i
}
```

\plmac@write@perl@i When `\plmac@write@perl@i` begins executing, the category codes are set up so that the macro's argument will be evaluated "verbatim" except for the part consisting of the LaTeX code passed in by the author, which is partially expanded. Thus, everything is in place for `\plmac@write@perl@i` to send its argument to Perl and read back the (LaTeX) result.

Because all of `perltex.sty`'s protocol processing is encapsulated within `\plmac@write@perl@i`, this is the only macro that strictly requires `perltex.pl`. Consequently, we wrap the entire macro definition within a check for `perltex.pl`.

```
\ifplmac@have@perltex
  \newcommand{\plmac@write@perl@i}[1]{%
```

The first step is to write argument `#1` to `\plmac@tofile`:

```
      \immediate\openout\plmac@outfile=\plmac@tofile\relax
      \let\protect=\noexpand
      \def\begin{\noexpand\begin}%
      \def\end{\noexpand\end}%
      \immediate\write\plmac@outfile{#1}%
      \immediate\closeout\plmac@outfile
```

(In the future, it might be worth redefining `\def`, `\edef`, `\gdef`, `\xdef`, `\let`, and maybe some other control sequences as "`\noexpand`⟨*control sequence*⟩`\noexpand`" so that `\write` doesn't try to expand an undefined control sequence.)

We're now finished using `#1` so we can end the group begun by `\plmac@write@perl`, thereby resetting each character's category code back to its previous value.

```
    \endgroup
```

Continuing the protocol illustrated in Figure 4, we create a zero-byte `\plmac@toflag` in order to notify `perltex.pl` that it's now safe to read `\plmac@tofile`.

```
    \immediate\openout\plmac@outfile=\plmac@toflag\relax
    \immediate\closeout\plmac@outfile
```

To avoid reading `\plmac@fromfile` before `perltex.pl` has finished writing it we must wait until `perltex.pl` creates `\plmac@fromflag`, which it does only after it has written `\plmac@fromfile`.

```
    \plmac@await@existence\plmac@fromflag
```

At this point, `\plmac@fromfile` should contain valid LaTeX code. However, we defer inputting it until we the very end. Doing so enables recursive and mutually recursive invocations of PerlTeX macros.

Because TeX can't delete files we require an additional LaTeX-to-Perl synchronization step. For convenience, we recycle `\plmac@tofile` as a synchronization file rather than introduce yet another flag file to complement `\plmac@toflag`, `\plmac@fromflag`, and `\plmac@doneflag`.

```
    \immediate\openout\plmac@outfile=\plmac@tofile\relax
    \immediate\closeout\plmac@outfile
    \plmac@await@existence\plmac@doneflag
```

The only thing left to do is to `\input` and evaluate `\plmac@fromfile`, which contains the LaTeX output from the Perl subroutine.

```
    \input\plmac@fromfile\relax
  }
```

`\plmac@write@perl@i` The foregoing code represents the "real" definition of `\plmac@write@perl@i`. For the user's convenience, we define a dummy version of `\plmac@write@perl@i` so that a document which utilizes `perltex.sty` can still compile even if not built using `perltex.pl`. All calls to macros defined with `\perl`[re]`newcommand` and all invocations of environments defined with `\perl`[re]`newenvironment` are replaced with "$\boxed{\text{PerlTeX}}$". A minor complication is that text can't be inserted before the `\begin{document}`. Hence, we initially define `\plmac@write@perl@i` as a do-nothing macro and redefine it as "`\fbox{Perl\TeX}`" at the `\begin{document}`.

```
\else
  \newcommand{\plmac@write@perl@i}[1]{\endgroup}
```

`\plmac@show@placeholder` There's really no point in outputting a framed "PerlTeX" when a macro is defined *and* when it's used. `\plmac@show@placeholder` checks the first character of the protocol header. If it's "D" (`DEF`), nothing is output. Otherwise, it'll be "U" (`USE`) and "PerlTeX" will be output.

```
  \gdef\plmac@show@placeholder#1#2\@empty{%
```

```
    \ifx#1D\relax
      \endgroup
    \else
      \endgroup
      \fbox{Perl\TeX}%
    \fi
  }%
```

```
  \AtBeginDocument{%
    \renewcommand{\plmac@write@perl@i}[1]{%
      \plmac@show@placeholder#1\@empty
    }%
  }
\fi
```

## 3.2 perltex.pl

perltex.pl is a wrapper script for latex (or any other LaTeX compiler). It sets up client-server communication between LaTeX and Perl, with LaTeX as the client and Perl as the server. When a LaTeX document sends a piece of Perl code to perltex.pl (with the help of perltex.sty, as detailed in Section 3.1), perltex.pl executes it within a secure sandbox and transmits the resulting LaTeX code back to the document.

### 3.2.1 Header comments

Because perltex.pl is generated without a DocStrip preamble or postamble we have to manually include the desired text as Perl comments.

```
#! /usr/bin/env perl

###########################################################
# Prepare a LaTeX run for two-way communication with Perl #
# By Scott Pakin <scott+pt@pakin.org>                     #
###########################################################

#-----------------------------------------------------------------
# This is file ‘perltex.pl’,
# generated with the docstrip utility.
```

```
#
# The original source files were:
#
# perltex.dtx  (with options: `perltex')
#
# This is a generated file.
#
# Copyright (C) 2007 Scott Pakin <scott+pt@pakin.org>
#
# This file may be distributed and/or modified under the conditions
# of the LaTeX Project Public License, either version 1.3c of this
# license or (at your option) any later version.  The latest
# version of this license is in:
#
#    http://www.latex-project.org/lppl.txt
#
# and version 1.3c or later is part of all distributions of LaTeX
# version 2006/05/20 or later.
#-------------------------------------------------------------------

```

### 3.2.2 Top-level code evaluation

In previous versions of `perltex.pl`, the `--nosafe` option created and ran code within a sandbox in which all operations are allowed (via `Opcode::full_opset()`). Unfortunately, certain operations still fail to work within such a sandbox. We therefore define a top-level "non-sandbox", `top_level_eval()`, in which to execute code. `top_level_eval()` merely calls `eval()` on its argument. However, it needs to be declared top-level and before anything else because `eval()` runs in the lexical scope of its caller.

```
sub top_level_eval ($)
{
    return eval $_[0];
}
```

### 3.2.3 Perl modules and pragmas

We use `Safe` and `Opcode` to implement the secure sandbox, `Getopt::Long` and `Pod::Usage` to parse the command line, and various other modules and pragmas for miscellaneous things.

```
use Safe;
use Opcode;
use Getopt::Long;
use Pod::Usage;
use File::Basename;
use Fcntl;
use POSIX;
use warnings;
use strict;
```

### 3.2.4 Variable declarations

With `use strict` in effect, we need to declare all of our variables. For clarity, we separate our global-variable declarations into variables corresponding to command-line options and other global variables.

#### Variables corresponding to command-line arguments

`$latexprog` `$runsafely` `@permittedops` `$usepipe`

`$latexprog` is the name of the LaTeX executable (e.g., "`latex`"). If `$runsafely` is `1` (the default), then the user's Perl code runs in a secure sandbox; if it's `0`, then arbitrary Perl code is allowed to run. `@permittedops` is a list of features made available to the user's Perl code. Valid values are described in Perl's `Opcode` manual page. `perltex.pl`'s default is a list containing only `:browse`. `$usepipe` is `1` if `perltex.pl` should attempt to use a named pipe for communicating with `latex` or `0` if an ordinary file should be used instead.

```
my $latexprog;
my $runsafely = 1;
my @permittedops;
my $usepipe = 1;
```

#### Filename variables

`$progname` `$jobname` `$toperl` `$fromperl` `$toflag` `$fromflag` `$doneflag` `$logfile` `$pipe`

`$progname` is the run-time name of the `perltex.pl` program. `$jobname` is the base name of the user's `.tex` file, which defaults to the TEX default of `texput`. `$toperl` defines the filename used for LATEX-to-Perl communication. `$fromperl` defines the filename used for Perl-to-LATEX communication. `$toflag` is the name of a file that will exist only after LATEX creates `$tofile`. `$fromflag` is the name of a file that will exist only after Perl creates `$fromfile`. `$doneflag` is the name of a file that will exist only after Perl deletes `$fromflag`. `$logfile` is the name of a log file to which `perltex.pl` writes verbose execution information. `$pipe` is the name of a Unix named pipe (or ordinary file on operating systems that lack support for persistent named pipes or in the case that `$usepipe` is set to `0`) used to convince the `latex` process to yield control of the CPU.

```
my $progname = basename $0;
my $jobname = "texput";
my $toperl;
my $fromperl;
my $toflag;
my $fromflag;
my $doneflag;
my $logfile;
my $pipe;
```

### Other global variables

`@latexcmdline` `$styfile` `@macroexpansions` `$sandbox` `$sandbox_eval` `$latexpid`

`@latexcmdline` is the command line to pass to the LATEX executable. `$styfile` is the string `noperltex.sty` if `perltex.pl` is run with `--makesty`, otherwise undefined. `@macroexpansions` is a list of PerlTEX macro expansions in the order they were encountered. It is used for creating a `noperltex.sty` file when `--makesty` is specified. `$sandbox` is a secure sandbox in which to run code that appeared in the LATEX document. `$sandbox_eval` is a subroutine that evalutes a string within `$sandbox` (normally) or outside of all sandboxes (if `--nosafe` is specified). `$latexpid` is the process ID of the `latex` process.

```
my @latexcmdline;
my $styfile;
my @macroexpansions;
my $sandbox = new Safe;
my $sandbox_eval;
my $latexpid;
```

$pipestring  `$pipestring` is a constant string to write to the `$pipe` named pipe (or file) at each LaTeX synchronization point. Its particular definition is really a bug workaround for XeTeX. The current version of XeTeX reads the first few bytes of a file to determine the character encoding (UTF-8 or UTF-16, big-endian or little-endian) then attempts to rewind the file pointer. Because pipes can't be rewound, the effect is that the first two bytes of `$pipe` are discarded and the rest are input. Hence, the "`\endinput`" used in prior versions of PerlTeX inserted a spurious "`ndinput`" into the author's document. We therefore define `$pipestring` such that it will not interfere with the document even if the first few bytes are discarded.

```
my $pipestring = "\%\%\%\%\% Generated by $progname\n\\endinput\n";
```

### 3.2.5 Command-line conversion

In this section, `perltex.pl` parses its own command line and prepares a command line to pass to `latex`.

**Parsing `perltex.pl`'s command line** We first set `$latexprog` to be the contents of the environment variable `PERLTEX` or the value "`latex`" if `PERLTEX` is not specified. We then use `Getopt::Long` to parse the command line, leaving any parameters we don't recognize in the argument vector (`@ARGV`) because these are presumably `latex` options.

```
$latexprog = $ENV{"PERLTEX"} || "latex";
Getopt::Long::Configure("require_order", "pass_through");
GetOptions("help"        => sub {pod2usage(-verbose => 1)},
           "latex=s"     => \$latexprog,
           "safe!"       => \$runsafely,
```

The following two options are undocumented because the defaults should always suffice. We're not yet removing these options, however, in case they turn out to be useful for diagnostic purposes.

```
           "pipe!"       => \$usepipe,
           "synctext=s"  => \$pipestring,
```

```
           "makesty"     => sub {$styfile = "noperltex.sty"},
           "permit=s"    => \@permittedops) || pod2usage(2);
```

### Preparing a LaTeX command line

$firstcmd $option We start by searching @ARGV for the first string that does not start with "-" or "\". This string, which represents a filename, is used to set $jobname.

```
@latexcmdline = @ARGV;
my $firstcmd = 0;
for ($firstcmd=0; $firstcmd<=$#latexcmdline; $firstcmd++) {
    my $option = $latexcmdline[$firstcmd];
    next if substr($option, 0, 1) eq "-";
    if (substr ($option, 0, 1) ne "\\") {
        $jobname = basename $option, ".tex" ;
        $latexcmdline[$firstcmd] = "\\input $option";
    }
    last;
}
push @latexcmdline, "" if $#latexcmdline==-1;
```

$separator To avoid conflicts with the code and parameters passed to Perl from LaTeX (see Figure 1 on page 13 and Figure 2 on page 14) we define a separator string, $separator, containing 20 random uppercase letters.

```
my $separator = "";
foreach (1 .. 20) {
    $separator .= chr(ord("A") + rand(26));
}
```

Now that we have the name of the LaTeX job ($jobname) we can assign $toperl, $fromperl, $toflag, $fromflag, $doneflag, $logfile, and $pipe in terms of $jobname plus a suitable extension.

```
$toperl = $jobname . ".topl";
$fromperl = $jobname . ".frpl";
$toflag = $jobname . ".tfpl";
$fromflag = $jobname . ".ffpl";
$doneflag = $jobname . ".dfpl";
$logfile = $jobname . ".lgpl";
$pipe = $jobname . ".pipe";
```

We now replace the filename of the .tex file passed to perltex.pl with a \definition of the separator character, \definitions of the various files, and the original file with \input prepended if necessary.

```
$latexcmdline[$firstcmd] =
    sprintf '\makeatletter' . '\def%s{%s}' x 7 . '\makeatother%s',
    '\plmac@tag', $separator,
    '\plmac@tofile', $toperl,
    '\plmac@fromfile', $fromperl,
    '\plmac@toflag', $toflag,
    '\plmac@fromflag', $fromflag,
    '\plmac@doneflag', $doneflag,
    '\plmac@pipe', $pipe,
    $latexcmdline[$firstcmd];
```

### 3.2.6 Launching LaTeX

We start by deleting the `$toperl`, `$fromperl`, `$toflag`, `$fromflag`, `$doneflag`, and `$pipe` files, in case any of these were left over from a previous (aborted) run. We also create a log file (`$logfile`), a named pipe (`$pipe`)—or a file containing only `\endinput` if we can't create a named pipe—and, if `$styfile` is defined, a LaTeX $2_\varepsilon$ style file. As `@latexcmdline` contains the complete command line to pass to `latex` we need only `fork` a new process and have the child process overlay itself with `latex`. `perltex.pl` continues running as the parent.

Note that here and elsewhere in `perltex.pl`, `unlink` is called repeatedly until the file is actually deleted. This works around a race condition that occurs in some filesystems in which file deletions are executed somewhat lazily.

```
foreach my $file ($toperl, $fromperl, $toflag, $fromflag, $doneflag, $pipe) {
    unlink $file while -e $file;
}
open (LOGFILE, ">$logfile") || die "open(\"$logfile\"): $!\n";
if (defined $styfile) {
    open (STYFILE, ">$styfile") || die "open(\"$styfile\"): $!\n";
}
if (!$usepipe || !eval {mkfifo($pipe, 0600)}) {
    sysopen PIPE, $pipe, O_WRONLY|O_CREAT, 0755;
    print PIPE $pipestring;
    close PIPE;
    $usepipe = 0;
}
defined ($latexpid = fork) || die "fork: $!\n";
unshift @latexcmdline, $latexprog;
if (!$latexpid) {
    exec {$latexcmdline[0]} @latexcmdline;
    die "exec('@latexcmdline'): $!\n";
}
```

### 3.2.7 Preparing a sandbox

`perltex.pl` uses Perl's `Safe` and `Opcode` modules to declare a secure sandbox (`$sandbox`) in which to run Perl code passed to it from LaTeX. When the sandbox compiles and executes Perl code, it permits only operations that are deemed safe. For example, the Perl code is allowed by default to assign variables, call functions, and execute loops. However, it is not normally allowed to delete files, kill processes, or invoke other programs. If `perltex.pl` is run with the `--nosafe` option we bypass the sandbox entirely and execute Perl code using an ordinary `eval()` statement.

```
if ($runsafely) {
    @permittedops=(":browse") if $#permittedops==-1;
    $sandbox->permit_only (@permittedops);
    $sandbox_eval = sub {$sandbox->reval($_[0])};
}
else {
    $sandbox_eval = \&top_level_eval;
}
```

### 3.2.8 Communicating with LaTeX

The following code constitutes `perltex.pl`'s main loop. Until `latex` exits, the loop repeatedly reads Perl code from LaTeX, evaluates it, and returns the result as per the protocol described in Figure 4 on page 24.

```
while (1) {
```

`$awaitexists` We define a local subroutine `$awaitexists` which waits for a given file to exist. If `latex` exits while `$awaitexists` is waiting, then `perltex.pl` cleans up and exits, too.

```perl
my $awaitexists = sub {
  while (!-e $_[0]) {
      sleep 0;
      if (waitpid($latexpid, &WNOHANG)==-1) {
          foreach my $file ($toperl, $fromperl, $toflag,
                            $fromflag, $doneflag, $pipe) {
              unlink $file while -e $file;
          }
          undef $latexpid;
          exit 0;
      }
  }
};
```

`$entirefile`

Wait for `$toflag` to exist. When it does, this implies that `$toperl` must exist as well. We read the entire contents of `$toperl` into the `$entirefile` variable and process it. Figures 1 and 2 illustrate the contents of `$toperl`.

```perl
$awaitexists->($toflag);
my $entirefile;
{
    local $/ = undef;
    open (TOPERL, "<$toperl") || die "open($toperl): $!\n";
    $entirefile = <TOPERL>;
    close TOPERL;
}
```

`$optag`
`$macroname`
`@otherstuff`

We split the contents of `$entirefile` into an operation tag (either `DEF`, `USE`, or `RUN`), the macro name, and everything else (`@otherstuff`). If `$optag` is `DEF` then `@otherstuff` will contain the Perl code to define. If `$optag` is `USE` then `@otherstuff` will be a list of subroutine arguments. If `$optag` is `RUN` then `@otherstuff` will be a block of Perl code to run.

```perl
my ($optag, $macroname, @otherstuff) =
    map {chomp; $_} split "$separator\n", $entirefile;
```

We clean up the macro name by deleting all leading non-letters, replacing all subsequent non-alphanumerics with “`_`”, and prepending “`latex_`” to the macro name.

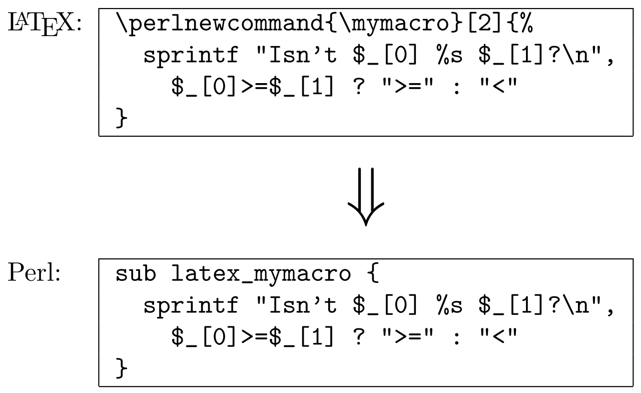
```
  $macroname =~ s/^[^A-Za-z]+//;
  $macroname =~ s/\W/_/g;
  $macroname = "latex_" . $macroname;
```

If we're calling a subroutine, then we make the arguments more palatable to Perl by single-quoting them and replacing every occurrence of "\" with "\\" and every occurrence of "'" with "\'".

```
  if ($optag eq "USE") {
    foreach (@otherstuff) {
        s/\\/\\\\/g;
        s/\'/\\\'/g;
        $_ = "'$_'";
    }
  }
```

`$perlcode` There are three possible values that can be assigned to `$perlcode`. If `$optag` is `DEF`, then `$perlcode` is made to contain a definition of the user's subroutine, named `$macroname`. If `$optag` is `USE`, then `$perlcode` becomes an invocation of `$macroname` which gets passed all of the macro arguments. Finally, if `$optag` is `RUN`, then `$perlcode` is the unmodified Perl code passed to us from `perltex.sty`. Figure 5 presents an example of how the following code converts a PerlTEX macro definition into a Perl subroutine definition and Figure 6 presents an example of how the following code converts a PerlTEX macro invocation into a Perl subroutine invocation.

```
  my $perlcode;
```



LaTeX:

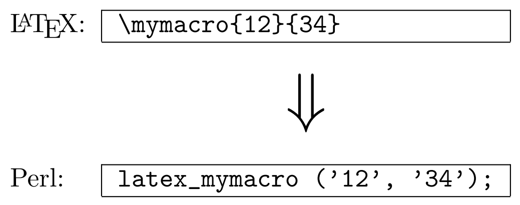
```
\perlnewcommand{\mymacro}[2]{%
  sprintf "Isn't $_[0] %s $_[1]?\n",
    $_[0]>=$_[1] ? ">=" : "<"
}
```

$\Downarrow$

Perl:

```
sub latex_mymacro {
  sprintf "Isn't $_[0] %s $_[1]?\n",
    $_[0]>=$_[1] ? ">=" : "<"
}
```

그림 5: Conversion from LaTeX to Perl (subroutine definition)



```
\mymacro{12}{34}
```

```
latex_mymacro ('12', '34');
```

그림 6: Conversion from LaTeX to Perl (subroutine invocation)

```
if ($optag eq "DEF") {
    $perlcode =
        sprintf "sub %s {%s}\n",
        $macroname, $otherstuff[0];
}
elsif ($optag eq "USE") {
    $perlcode = sprintf "%s (%s);\n", $macroname, join(", ", @otherstuff);
}
elsif ($optag eq "RUN") {
    $perlcode = $otherstuff[0];
}
else {
    die "${progname}: Internal error -- unexpected operation tag \"$optag\"\n";
}
```

Log what we're about to evaluate.

```
print LOGFILE "#" x 31, " PERL CODE ", "#" x 32, "\n";
print LOGFILE $perlcode, "\n";
```

`$result` `$msg` We're now ready to execute the user's code using the `$sandbox_eval` function. If a warning occurs we write it as a Perl comment to the log file. If an error occurs (i.e., `$@` is defined) we replace the result (`$result`) with a call to LaTeX $2_\varepsilon$'s `\PackageError` macro to return a suitable error message. We produce one error message for sandbox policy violations (detected by the error message, `$@`, containing the string "`trapped by`") and a different error message for all other errors caused by executing the user's code. For clarity of reading both warning and error messages, we elide the string "`at (eval` ⟨*number*⟩`) line` ⟨*number*⟩". Once `$result` is defined—as either the resulting LaTeX code or as a `\PackageError`—we store it in `@macroexpansions` in preparation for writing it to `noperltex.sty` (when perltex.pl is run with --makesty).

```
undef $_;
my $result;
{
    my $warningmsg;
    local $SIG{__WARN__} =
        sub {chomp ($warningmsg=$_[0]); return 0};
    $result = $sandbox_eval->($perlcode);
    if (defined $warningmsg) {
        $warningmsg =~ s/at \(eval \d+\) line \d+\W+//;
        print LOGFILE "# ===> $warningmsg\n\n";
    }
}
$result = "" if !$result || $optag eq "RUN";
if ($@) {
    my $msg = $@;
    $msg =~ s/at \(eval \d+\) line \d+\W+//;
    $msg =~ s/\s+/ /;
    $result = "\\PackageError{perltex}{$msg}";
    my @helpstring;
    if ($msg =~ /\btrapped by\b/) {
        @helpstring =
            ("The preceding error message comes from Perl.  Apparently,",
             "the Perl code you tried to execute attempted to perform an",
             "‘unsafe’ operation.  If you trust the Perl code (e.g., if",
             "you wrote it) then you can invoke perltex with the --nosafe",
             "option to allow arbitrary Perl code to execute.",
             "Alternatively, you can selectively enable Perl features",
             "using perltex's --permit option.  Don't do this if you don't",
             "trust the Perl code, however; malicious Perl code can do a",
             "world of harm to your computer system.");
    }
    else {
        @helpstring =
          ("The preceding error message comes from Perl.  Apparently,",
           "there's a bug in your Perl code.  You'll need to sort that",
           "out in your document and re-run perltex.");
    }
    my $helpstring = join ("\\MessageBreak\n", @helpstring);
    $helpstring =~ s/\.  /.\\space\\space /g;
    $result .= "{$helpstring}";
}
push @macroexpansions, $result if defined $styfile && $optag eq "USE";
```

Log the resulting LaTeX code.

```
    print LOGFILE "%" x 30, " LATEX RESULT ", "%" x 30, "\n";
    print LOGFILE $result, "\n\n";
```

We add `\endinput` to the generated LaTeX code to suppress an extraneous end-of-line character that TeX would otherwise insert.

```
    $result .= '\endinput';
```

Continuing the protocol described in Figure 4 on page 24 we now write `$result` (which contains either the result of executing the user's or a `\PackageError`) to the `$fromperl` file, delete `$toflag`, `$toperl`, and `$doneflag`, and notify LaTeX by touching the `$fromflag` file. As a performance optimization, we also write `\endinput` into `$pipe` to wake up the `latex` process.

```
    open (FROMPERL, ">$fromperl") || die "open($fromperl): $!\n";
    syswrite FROMPERL, $result;
    close FROMPERL;

    unlink $toflag while -e $toflag;
    unlink $toperl while -e $toperl;
    unlink $doneflag while -e $doneflag;

    open (FROMFLAG, ">$fromflag") || die "open($fromflag): $!\n";
    close FROMFLAG;

    if (open (PIPE, ">$pipe")) {
        print PIPE $pipestring;
        close PIPE;
    }
```

We have to perform one final LaTeX-to-Perl synchronization step. Otherwise, a subsequent `\perl[re]newcommand` would see that `$fromflag` already exists and race ahead, finding that `$fromperl` does not contain what it's supposed to.

```
    $awaitexists->($toperl);
    unlink $fromflag while -e $fromflag;
    open (DONEFLAG, ">$doneflag") || die "open($doneflag): $!\n";
    close DONEFLAG;
```

Again, we awaken the `latex` process, which is blocked on `$pipe`.

```
    if (open (PIPE, ">$pipe")) {
        print PIPE $pipestring;
        close PIPE;
    }
}
```

### 3.2.9 Final cleanup

If we exit abnormally we should do our best to kill the child `latex` process so that it doesn't continue running forever, holding onto system resources.

```
END {
    close LOGFILE;
    if (defined $latexpid) {
        kill (9, $latexpid);
        exit 1;
    }

    if (defined $styfile) {
```

This is the big moment for the `--makesty` option. We've accumulated the output from each PerlTEX macro invocation into `@macroexpansions`, and now we need to produce a `noperltex.sty` file. We start by generating a boilerplate header in which we set up the package and load both `perltex.sty` and `filecontents.sty`.

```
        print STYFILE <<"STYFILEHEADER1";
\\NeedsTeXFormat{LaTeX2e}[1999/12/01]
\\ProvidesPackage{noperltex}
    [2007/09/29 v1.4 Perl-free version of PerlTeX specific to $jobname.tex]
STYFILEHEADER1
        ;
        print STYFILE <<'STYFILEHEADER2';
\RequirePackage{filecontents}

% Suppress the "Document must be compiled using perltex" error from perltex.
\let\noperltex@PackageError=\PackageError
```

```
\renewcommand{\PackageError}[3]{}
\RequirePackage{perltex}
\let\PackageError=\noperltex@PackageError

```

\plmac@macro@invocation@num
\plmac@show@placeholder

`noperltex.sty` works by redefining the `\plmac@show@placeholder` macro, which normally outputs a framed "PerlTEX" when `perltex.pl` isn't running, changing it to input `noperltex-`⟨*number*⟩`.tex` instead (where ⟨*number*⟩ is the contents of the `\plmac@macro@invocation@num` counter). Each `noperltex-`⟨*number*⟩`.tex` file contains the output from a single invocation of a PerlTEX-defined macro.

```
% Modify \plmac@show@placeholder to input the next noperltex-*.tex file
% each time a PerlTeX-defined macro is invoked.
\newcount\plmac@macro@invocation@num
\gdef\plmac@show@placeholder#1#2\@empty{%
  \ifx#1U\relax
    \endgroup
    \advance\plmac@macro@invocation@num by 1\relax
    \global\plmac@macro@invocation@num=\plmac@macro@invocation@num
    \input{noperltex-\the\plmac@macro@invocation@num.tex}%
  \else
    \endgroup
  \fi
}
STYFILEHEADER2
        ;
```

Finally, we need to have `noperltex.sty` generate each of the `noperltex-`⟨*number*⟩`.tex` files. For each element of `@macroexpansions` we use one `filecontents` environment to write the macro expansion verbatim to a file.

```
        foreach my $e (0 .. $#macroexpansions) {
            print STYFILE "\n";
            printf STYFILE "%% Invocation #%d\n", 1+$e;
                printf STYFILE "\\begin{filecontents}{noperltex-%d.tex}\n", 1+$e;
            print STYFILE $macroexpansions[$e], "\\endinput\n";
            print STYFILE "\\end{filecontents}\n";
        }
        print STYFILE "\\endinput\n";
        close STYFILE;
    }

    exit 0;
}

__END__
```

#### 3.2.10 `perltex.pl` POD documentation

`perltex.pl` includes documentation in Perl's POD (Plain Old Documentation) format. This is used both to produce manual pages and to provide usage information when `perltex.pl` is invoked with the `--help` option. The POD documentation is not listed here as part of the documented `perltex.pl` source code because it contains essentially the same information as that shown in Section 2.2. If you're curious what the POD source looks like then see the generated `perltex.pl` file.

### 3.3 Porting to other languages

Perl is a natural choice for a LaTeX macro language because of its excellent support for text manipulation including extended regular expressions, string interpolation, and "here" strings, to name a few nice features. However, Perl's syntax is unusual and its semantics are rife with annoying special cases. Some users will therefore long for a ⟨*some-language-other-than-Perl*⟩TeX. Fortunately, porting PerlTeX to use a different language should be fairly straightforward. `perltex.pl` will need to be rewritten in the target language, of course, but `perltex.sty` modifications will likely be fairly minimal. In all probability, only the following changes will need to be made:

- Rename `perltex.sty` and `perltex.pl` (and choose a package name other than "PerlTeX") as per the PerlTeX license agreement (Section 4).
- In your replacement for `perltex.sty`, replace all occurrences of "`plmac`" with a different string.
- In your replacement for `perltex.pl`, choose different file extensions for the various helper files.

The importance of these changes is that they help ensure version consistency and that they make it possible to run ⟨*some-language-other-than-Perl*⟩TEX alongside PerlTEX, enabling multiple programming languages to be utilized in the same LATEX document.

## 제 4 절 License agreement

## Acknowledgments

Thanks to Andrew Mertz for writing the first draft of the code that produces the PerlTEX-free `noperltex.sty` style file and for testing the final draft; to Andrei Alexandrescu for providing a few bug fixes; and to Nick Andrewes for identifying and helping diagnose a problem running PerlTEX with XƎTEX and to Jonathan Kew for suggesting a workaround. Also, thanks to the many people who have sent me fan mail or submitted bug reports, documentation corrections, or feature requests. (The `\perldo` macro and the `--makesty` option were particularly popular requests.)