Programming with PerlTEX

Andrew Mertz, William Slough
Department of Mathematics and Computer Science
Eastern Illinois University
Charleston, IL 61920
aemertz (at) eiu dot edu, waslough (at) eiu dot edu

Abstract

PerlTEX couples two well-known worlds — the Perl programming language and the LATEX typesetting system. The resulting system provides users with a way to augment LATEX macros with Perl code, thereby adding programming capabilities to LATEX that would otherwise be difficult to express. In this paper, we illustrate the use of PerlTEX with a variety of examples and explain the associated Perl code. Although Perl may perhaps be best known for its string manipulation capabilities, we demonstrate how PerlTEX indirectly provides support for "programming" graphics through the use of additional packages such as Ti*k*Z.

1 Introduction

The typesetting capabilities of TEX and LATEX are well known. Each has the ability to define macros, adding significant flexibility and convenience. However, to achieve some effects in TEX requires a level of expertise many users lack.

Perl [8] is a programming language with particular strengths in string processing and scripting. Since it borrows concepts from other languages such as C and SED, its syntax is likely to be reasonably familiar to many.

PerlTEX [4] provides a way to incorporate the expressiveness of Perl directly within a LATEX document. In doing so, the computing capabilities of Perl are coupled with the document preparation abilities present in LATEX. Combining these two systems has an important outcome: for those who already know or are willing to learn some of Perl's rudiments, a number of typesetting tasks become more convenient to express.

2 A first example

In Chapter 20 of *The TEXbook* [3], a TEX macro which generates the first N prime numbers is described, where N is a specified parameter of the macro. This discussion and macro earn Knuth's double dangerous-bend symbols, a warning to readers that esoteric topics are under discussion.

It is probably fair to say that the design of such a macro using the primitives of TEX is a task best left to experts. A simpler approach uses PerlTEX. Figure 1 shows the details.

To use this command, we request the prime numbers within a specified interval. For example,

```
\perlnewcommand{\listPrimes}[2] {
  use Math::Prime::XS "primes";
  return join(" ", primes($_[0], $_[1]));
}
```

Figure 1: Definition of a command, `\listPrimes`, to generate prime numbers. The two arguments of this command specify the desired range of values to be generated.

```
\listPrimes{10}{30}
```

generates the typeset sequence

11 13 17 19 23 29

consisting of the prime numbers between 10 and 30.

Let's take a closer look at how this is accomplished. To begin, we use `\perlnewcommand`, the PerlTEX analog of the `\newcommand` of LATEX. We thus define a new command, `\listPrimes`, with two arguments.

In contrast to the `\newcommand` of LATEX, Perl code is placed within the definition portion of a `\perlnewcommand`. For the current example,

```
use Math::Prime::XS "primes";
```

imports a Perl module which contains a function, named `primes`, which is perfectly suited to the task at hand. Given a pair of values which specify the desired range, this function returns a list of the primes in that range.

The arguments of `\listPrimes` are accessed with the Perl notation `$_[0]` and `$_[1]`. Thus,

```
primes($_[0], $_[1])
```

yields a *list* of the desired primes. To complete the definition of `\listPrimes`, a single *string* must be created from the collection of primes just obtained.

This is easily achieved with Perl's `join`. Here, we use a single space to separate adjacent primes.

In this example, the use of an existing Perl function, `primes`, avoids "reinventing the wheel". Since there are many such functions available, this is one direct benefit of PerlTEX. A wealth of Perl functions can be located by consulting CPAN, the comprehensive Perl archive network, found at `www.cpan.org`.

3 Variations on a theme

Rather than produce one sequence of primes, as in our first example, now suppose a tabular array of primes is desired. We will define a new command, `\tablePrimes`, with three arguments: the first two specify the desired range of primes, as before, and the third argument indicates the number of columns to be used. For example, the command

```
\tablePrimes{1}{20}{3}
```

will produce a table consisting of the primes between 1 and 20, typeset in three columns.

We will show two different definitions for this command. The first solution uses a direct approach, illustrating how the looping and conditional control structures of Perl can be used to generate the required LaTeX code. In the second solution the power of regular expressions is used to achieve the same result, avoiding the need for explicit looping and testing.

Before taking up the Perl details, let's consider the desired LaTeX code to be generated. For the three-column example above, the following needs to be generated:

```
\begin{tabular}{*{3}{r}}
2 & 3 & 5\\
7 & 11 & 13\\
17 & 19 &
\end{tabular}
```

Of course, this is just a tabular environment consisting of the primes to appear within the table. It is helpful to think of this as one string, subdivided into three parts: the beginning of the environment, the environment content, and the end of the environment. The first definition of `\tablePrimes`, shown in Figure 2, reflects this view.

Consider the final `return` statement in this definition. Using Perl's concatenation or dot operator, three string components are formed to yield the desired tabular environment. In the first component, `$_[2]` is used to obtain the value of the third parameter, the number of columns. Each backslash which is to appear in the generated LaTeX code must also be escaped in Perl to avoid its usual meaning. So, for example, `\\begin` appears in order to obtain `\begin`. Without escaping the backslash, Perl would interpret `\b` as a backspace. The use of `\n` in this `return` statement ensures that each component begins on a new line.

```
\perlnewcommand{\tablePrimes}[3] {
 use Math::Prime::XS "primes";
 my $count = 0;
 my $primes = "";
 foreach my $item (primes($_[0], $_[1])) {
   $primes .= $item;
   $count++;
   if ($count == $_[2]) {
     $primes .= "\\\\ \n";
     $count = 0;
   }
   else {
     $primes .= " & ";
   }
 }
 return "\\begin{tabular}{*{$_[2]}{r}}\n" .
        $primes . "\n" .
        "\\end{tabular} \n";
}
```

Figure 2: Definition of a command, `\tablePrimes`, to generate a table of prime numbers.

At this point in the definition, the Perl variable `$primes` contains the string of all primes needed for the table, with `&` column separators and `\\` row separators inserted as appropriate. Everything in the definition prior to the `return` statement is present to generate this string.

This portion of the definition is straightforward, though a few comments might be helpful. The keyword `my` is used when Perl variables are introduced to indicate they are *local*, which is generally a good idea to prevent unintended interactions.

The variable `$primes` begins as an empty string and grows to include all of the needed values to appear in the table. Perl's compound operator `.=` is used to append a new value to this string. We use the `foreach` construct to iterate over all of the primes generated, appending each in turn to the `$primes` string. Column or row separators are appended to this string by keeping count of which column has just been added to the string. As before, some care is needed regarding the escape character. For example, `\\\\` is used to generate `\\`.

The second definition for `\tablePrimes` takes a different viewpoint of the generation of `$strings`. A two-step process is used to generate the value for the tabular environment. As a first step, a `&`-separated string of primes is constructed:

```
    my $primes = join("&",
                      primes($_[0], $_[1]));
```

This generates all primes, incorrectly assuming that the tabular will consist of one very long row. The second step corrects this assumption by replacing every kth column separator with a row separator. This can be achieved using regular expressions:

```
    $primes =~ s/((\d+&){$k}\d+)&/
                $1\\\\ \n/g;
```

Though this might be viewed as somewhat cryptic, the use of regular expressions is one of the widely quoted strengths of Perl and can be used, as here, to concisely describe a pattern substitution. Putting these ideas together yields the definition shown in Figure 3.

```
\perlnewcommand{\tablePrimes}[3] {
 use Math::Prime::XS "primes";

 # Number of ampersands needed per line
 my $k = $_[2] - 1;

 # Build a string of &-separated primes
 my $primes = join("&",
                   primes($_[0], $_[1]));

 # Insert newlines for each row
 $primes =~ s/((\d+&){$k}\d+)&/$1\\\\ \n/g;

 # Put the pieces together
 return "\\begin{tabular}{*{$_[2]}{r}}\n" .
        $primes . "\n" .
        "\\end{tabular} \n";
}
```

Figure 3: Alternate definition of `\tablePrimes`. A regular expression is used to subdivide the primes into rows.

4 Layout and processing

The layout of a LaTeX document which uses PerlTeX is straightforward. Within the preamble

```
    \usepackage{perltex}
```

loads the PerlTeX package. Also in the preamble, one or more PerlTeX commands are defined with `\perlnewcommand`. Within the document environment, the PerlTeX commands which were defined can be utilized. Figure 4 shows an example.

Processing a PerlTeX source file requires the services of both Perl and TeX. This is accomplished using a script provided with PerlTeX. For example, the command

```
    perltex foo.tex
```

```
\documentclass{article}
...
\usepackage{perltex}
...

\perlnewcommand{\tablePrimes}[3]{
    definition
}

\begin{document}
   ...
   \tablePrimes{1}{20}{3}
   ...
   \tablePrimes{1}{20}{4}
   ...
\end{document}
```

Figure 4: Sample layout of a LaTeX document intended for PerlTeX. The definition of `\tablePrimes` is omitted here.

processes the source file `foo.tex`. As shown in Figure 5, this initiates the processing by creating a pair of communicating processes, one each for TeX and Perl, ultimately creating the output file.

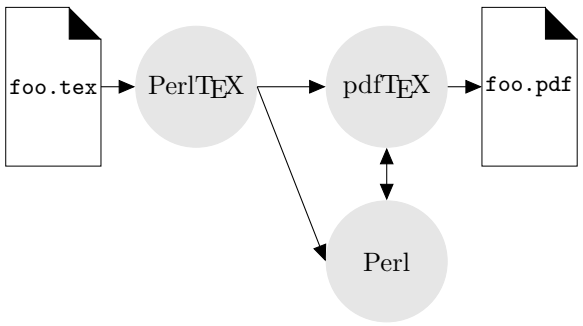

Figure 5: Processing a source file with PerlTeX.

By default, PerlTeX causes the Perl processing to use a secure sandbox, insulating the user from potentially dangerous actions, such as removal of directories or other undesirable system-related actions. If this is not desired, the command

```
    perltex --nosafe foo.tex
```

disables the sandbox. Disabling the sandbox is helpful when importing Perl modules, accessing files or the network, and in many other cases.

Introducing Perl code provides new opportunities for errors. To assist with debugging, PerlTeX creates a log file with the suffix `lgpl` which contains all of the Perl and LaTeX code generated during processing. Any error messages returned by the Perl interpreter also appear in this file.

5 Graphical output

One appealing feature of PerlTEX is its ability to interact with other packages. To see an example of this, consider the triangular array of binomial coefficients — more popularly known as Pascal's triangle. If a fixed modulus m is selected, each entry of the triangle can be reduced, modulo m. Each of the m possible remainder values can be assigned a color, providing a way to visualize the coefficients as a multi-colored graphic. A number of very attractive diagrams of this sort with varying moduli appear in *Chaos and Fractals* [5]. In this section, we use Ti*k*Z [6] to take care of the graphical aspects, while using PerlTEX to generate the numerical values of Pascal's triangle.

As a starting point, Figure 6 defines a PerlTEX command, `\pascalMatrix`, which generates a tabular array of binomial coefficients. Its single argument specifies the size of the triangle. (For an argument n, the rows of the array are numbered 0 through n.) For example, `\pascalMatrix{5}` yields the triangular array:

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

As in previous examples, the `return` statement in this definition is responsible for creating the entire tabular environment. In this case, the variable `$tabularRows` contains all of the rows needed for Pascal's triangle.

Each iteration of the outer loop appends one complete row to `tabularRows`. The inner loop is responsible for generating row $r+1$ given the contents of row r, using a well-known identity for binomial coefficients: $\binom{r+1}{c} = \binom{r}{c-1} + \binom{r}{c}$.

Perl supports the syntax of the familiar `for` loop of the C programming language, thus allowing a common looping mechanism to be used within TEX.

Figure 7 gives a graphical view of Pascal's triangle. In this diagram, a modulus of two has been used, giving two possible remainders, shown as white and black. This diagram can be specified using Ti*k*Z as a sequence of `\fill` statements, as shown in Figure 8. Each `\fill` statement is responsible for producing one small square of the diagram and specifies its color, position, and size.

Figure 9 is a revision of `\pascalMatrix`. Using this definition, the necessary `\fill` statements to generate a graphical form of Pascal's triangle can

```
\perlnewcommand{\pascalMatrix}[1] {
  my $tabularRows = "";
  my @row = (1);
  for (my $r = 0; $r <= $_[0]; $r++) {
    # Output the current row
    $tabularRows .= join ("&", @row) .
                    " \\\\\n";
    # Generate the next row
    my @nextRow = (1);
    for (my $c = 1; $c <= $r; $c++) {
      push @nextRow,
           @row[$c - 1] + @row[$c];
    }
    push @nextRow, 1;
    @row = @nextRow;
  }
  return
    "\\begin{tabular}{*{$_[0]}{c}c}\n" .
    $tabularRows .
    "\\end{tabular}\n";
}
```

Figure 6: Generating entries of Pascal's triangle.

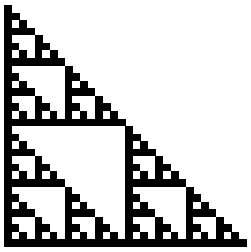

Figure 7: Graphical view of Pascal's triangle.

```
\begin{tikzpicture}[scale=0.5]
  \fill[black] (0,0) rectangle +(1,1);
  \fill[black] (0,-1) rectangle +(1,1);
  \fill[black] (1,-1) rectangle +(1,1);
  \fill[black] (0,-2) rectangle +(1,1);
  \fill[white] (1,-2) rectangle +(1,1);
  \fill[black] (2,-2) rectangle +(1,1);
  \fill[black] (0,-3) rectangle +(1,1);
  \fill[black] (1,-3) rectangle +(1,1);
  \fill[black] (2,-3) rectangle +(1,1);
  \fill[black] (3,-3) rectangle +(1,1);
\end{tikzpicture}
```

Figure 8: Ti*k*Z code for four rows of Pascal's triangle.

```
\perlnewcommand{\pascalGraphic}[1]{
  my $result = "";
  my @row = (1);
  my @colors = ("white", "black");
  for (my $r = 0; $r <= $_[0]; $r++) {
    # Output the current row
    for (my $c = 0; $c <= $r; $c++) {
      $result .= sprintf("\\fill[%s] (%d, %d) rectangle +(1, 1);\n",
                         $colors[$row[$c]], $c, -$r);
    }
    # Generate the next row
    my @nextRow = (1);
    for (my $c = 1; $c <= $r; $c++) {
      push @nextRow, (@row[$c - 1] + @row[$c]) % 2;
    }
    push @nextRow, 1;
    @row = @nextRow;
  }
  return $result;
}
```

Figure 9: Generating a graphical view of Pascal's triangle, modulo two.

```
TimeCDT,TemperatureF,Dew PointF,Humidity,Sea Level PressureIn,VisibilityMPH,
Wind Direction,Wind SpeedMPH<BR>
12:53 AM,73.0,70.0,90,30.05,10.0,SSW,4.6<BR>
1:53 AM,73.0,69.1,87,30.04,9.0,SW,3.5<BR>
2:53 AM,72.0,68.0,87,30.04,10.0,West,3.5<BR>
 additional lines omitted
<!-- 0.122:1 -->
```

Figure 10: A brief excerpt of weather information obtained from the Weather Underground.

be obtained. For example,

```
\begin{tikzpicture}[scale=0.1]
  \pascalGraphic{31}
\end{tikzpicture}
```

generates the 32-row graphic of Figure 7.

As might be expected, the definitions for the two Pascal triangle commands are very similar. In the graphical version, a single string consisting of the sequence of `\fill` statements is built up in `$result`. Each of these `\fill` statements is obtained by a formatted print statement, appended to `$result`. Also, since values within any one row of the triangle are stored modulo two, Perl's `%` operator is used. Both `sprintf` and the `%` operator are language features shared with C.

6 LaTeX documents and the Internet

The LWP* Perl library [1] can be used to access data on the web. With the assistance of PerlTeX, this allows information from web sites to be retrieved and incorporated within a LaTeX document.

For example, suppose we wish to access weather data and display it in either tabular or graphical form within a LaTeX document. This type of processing is made possible by LWP and Perl's support for regular expressions.

The Weather Underground† is one of many sites which provides access to historical weather data. Given an airport code, year, month, and day, it is possible to retrieve many details about the weather recorded at the requested location and date. Figure 10 shows an excerpt of the results of a web request for June 28, 2007, at the Coles County Memorial Airport, with airport code KMTO. What is important to know about this request is that

```
KMTO/2007/6/28
```

appears as a substring of a lengthy URL. Figure 11 shows how this raw weather data is to be formatted as a tabular.

* LWP is an acronym for 'Library for WWW in Perl'.

† `www.wunderground.com`

Time	Temp	Dew Point	Humidity	SL Pressure	Vis	Wind Dir	Wind Speed
12:53 AM	73.0	70.0	90	30.05	10.0	SSW	4.6
1:53 AM	73.0	69.1	87	30.04	9.0	SW	3.5
2:53 AM	72.0	68.0	87	30.04	10.0	West	3.5

Figure 11: An excerpt of formatted weather data.

```
\perlnewcommand{\getWeather}[4] {
  use LWP::Simple;

  # Form the URL from the airport code, year, month, and day
  $id = join"/", @_;
  $URL =  A URL incorporating $id;

  # If we have already looked up this day, do nothing
  return "" if exists $data{$id};

  # Otherwise fetch and store the data
  $data{$id} = get$URL;

  # Return nothing as this command only fetches data but
  # does not cause anything to appear in the document.
  return "";
}
```

Figure 12: A command to fetch the weather data from a web site. The four parameters specify airport code, year, month, and day.

The data is retrieved and formatted with two PerlTEX commands: one retrieves the data from the Web site, the other one performs some text manipulations and formats the data as a tabular. A variety of text substitutions are needed, for example, to account for HTML tags which are present in the retrieved data.

The first of these commands, `\getWeather`, is shown in Figure 12. This command introduces variables `$id` and `$data`, but does not declare them to be local, thus making them accessible from the second command, `\formatWeather`. The command

```
\getWeather{KMTO}{2007}{6}{28}
```

creates the appropriate URL, accesses the web site to retrieve the data, and saves the result in the Perl hash variable `$data`. This command differs from the others presented in that the return value is of no interest: rather, it is the side-effect of storing the weather data in `$data` that is desired.

Figure 13 shows the details of `\formatWeather`. Weather data is obtained from the web site by invoking `latex_getWeather`, the Perl function created from the definition of `\getWeather`. At this point, various textual substitutions are made to the data. For example, the comma-separated values are adjusted to become ampersand-separated values, in anticipation of inclusion in a tabular environment. This command illustrates some of the string-processing conveniences of Perl.

In this example, information was retrieved from the Internet. However, data can be obtained from files, databases, and other sources just as easily.

7 Graphical animations

The `animate` package [2] provides a convenient way to create PDF files with animated content consisting of a sequence of frames, optionally controlled with VCR-style buttons. We provide a brief example of the use of this package with a special focus on the role PerlTEX can play.

One of the environments provided by this package, `animateinline`, creates animations from the typeset material within its body. This might consist of a sequence of Ti*k*Z commands which generate frames of the animation.

In many cases, each frame of an animation can be viewed under the control of some parameter t. For example, an animation of a Bézier curve can be constructed as a sequence of frames controlled by t, where t varies from 0 to 1.

For the present example, assume there is a command, `\bcurve`, with a single parameter t which yields a graphical image of a single frame. To achieve

```
\perlnewcommand{\formatWeather}[4] {
  # Ensure the appropriate weather data has been retrieved
  latex_getWeather($_[0], $_[1], $_[2], $_[3]);
  $rows = $data{$id};

  $rows =~ s/(.*\n){2}//; # Strip the first two lines

  # Add the headings
  $rows = "Time,Temp,Dew Point,Humidity,SL Pressure,Vis," .
          "Wind Dir,Wind Speed\n" . $rows;
  $rows =~ tr/,/&/;       # Commas become alignment tabs
  $rows =~ s/\n/\\\\\n/g; # Newlines become the end of a row
  $rows =~ s/<.*>//g;     # Remove any HTML tags

  # Return the table
  return "\\begin{tabular}{...}\n $rows \\end{tabular}\n";
}
```

Figure 13: A command to tabulate the weather data obtained from a web site. Notice how the PerlTEX command `\getWeather` is invoked.

a smooth animation, what is needed is a sequence of these frames, with closely spaced values of t. For the `animateinline` environment, a sequence such as:

```
\bcurve{0}%
\newframe%
\bcurve{0.02}%
\newframe%
 etc.
```

is needed to generate the animation.

This sequence can be generated easily with a PerlTEX command. In doing so, we cover a wide class of animations, namely, those that can be described as a sequence of frames controlled by a single parameter.

Figure 14 shows the details of a command which generates a sequence of frames. This command takes four arguments: the name of the command which generates a single frame, the starting and ending values of t, and the total number of frames desired. For example,

```
\animationLoop{bcurve}{0}{1}{51}
```

generates the sequence of 51 frames `\bcurve{0.0}`, `\bcurve{0.02}`, ..., `\bcurve{1.0}`.

8 Demonstrating algorithms

One way to understand an algorithm is to trace its actions, maintaining a history of the values being computed and stored. For some algorithms, this history can be compactly displayed with a tabular environment.

Using the `beamer` package [7], this type of tabular information can be incrementally revealed by inserting `\pause` commands at selected locations. In the context of demonstrating an algorithm, such pauses can be used to show the effect of one iteration of a loop.

To illustrate, consider Euclid's algorithm for computing the greatest common divisor. At the heart of this algorithm is a loop which repeats the following three actions:

```
r = x % y;  x = y;  y = r;
```

The history for this algorithm is a tabular array with three columns, one for each of the three variables. A partial history might reveal the following table, "at rest" at a `\pause`.

x	y	r
120	70	50
70	50	20
50	20	

Picking up after the `\pause`, the history grows by an amount equal to one iteration of the loop:

x	y	r
120	70	50
70	50	20
50	20	10
20	10	

As shown in Figure 15, these types of tabular environments can be generated with a PerlTEX command. Since the values in the tabular are being computed by Euclid's algorithm, it is easy to generate a wide variety of example histories — and to have confidence that the values in the table are correct!

```
\perlnewcommand{\animationLoop}[4]{
  my $result = "";
  my $delta = ($_[2] - $_[1]) / ($_[3] - 1.0);
  my $x = $_[1];
  for (my $count = 1; $count < $_[3]; $count++) {
    $result .= "\\" . $_[0] . "{$x}%\n" .
               "\\newframe%\n";
    $x += $delta;
  }
  return $result . "\\" . $_[0] . "{$_[2]}%\n";
}
```

Figure 14: A command to generate a sequence of frames in a graphical animation.

```
\perlnewcommand{\euclidAlgorithm}[2]{
  my $x = $_[0];
  my $y = $_[1];
  my $result = "$x & $y & \\pause ";
  while ($y != 0) {
    my $r = $x % $y;
    $result .= "$r \\\\ \\hline \\pause \n";
    $x = $y;
    $y = $r;
    $result .= "$x & $y & ";
    $result .= "\\pause " if $y != 0;
  }
  return
    "\\begin{tabular}{c|c|c} \\hline \n" .
    "\$x\$ & \$y\$ & \$x \\bmod y\$\\\\  \\hline \\pause \n" .
    $result . "\\\\ \\hline \n" .
    "\\end{tabular}";
}
```

Figure 15: A PerlTEX command to generate a tabular history for Euclid's algorithm.

9 Shuffling an enumerated list

So far, all of the examples presented have focused on the ability to define new commands with PerlTEX. However, PerlTEX also provides a mechanism to define a new environment: `\perlnewenvironment`. A command of the form

`\perlnewenvironment{foo}{`*start*`}{`*finish*`}`

defines a new environment, named `foo`. PerlTEX replaces a subsequent `\begin{foo}` with the *start* text and an `\end{foo}` with the *finish* text.

To illustrate, suppose it is desired to typeset a shuffled enumerated list. In principle, each time the source text is processed, a different ordering of the list items could result. To accomplish this, we introduce a new environment, `shuffle`, and a new command, `\shuffleItem`. For example, Figure 16 illustrates how this environment can be used to typeset an enumerated list of the four given items, arranged in an arbitrary order.

```
\begin{shuffle}
  \shuffleItem{TUG 2007}
  \shuffleItem{SDSU}
  \shuffleItem{San Diego}
  \shuffleItem{California}
\end{shuffle}
```

Figure 16: An example of the use of the `shuffle` environment.

To achieve this behavior, consider the action required for each item in the `shuffle` environment. As shown in Figure 17, each item which appears gets appended to a variable, `@items`, which maintains a list of all items encountered thus far.

The bulk of the work takes place at the conclusion of the `shuffle` environment. The items that have been accumulating are now rearranged and an enumerated list is constructed from this permuted list. Figure 18 provides the Perl details. The variable `@items` is undefined as a final step, since a

```
\perlnewcommand{\shuffleItem}[1]{
  push @items, $_[0];
  return "";
}
```

Figure 17: A PerlTEX command which stores one item of a `shuffle` environment.

subsequent `shuffle` environment must start with a "clean slate" of items. Permuting the list of items is a simple operation, since there is a Perl library module well-suited to this task.

Although this implementation of shuffled enumerated lists does not allow for nested shuffled lists, it does nevertheless provide an illustration of the ability to define new environments within PerlTEX.

```
\perlnewenvironment{shuffle}
{return "\\begin{enumerate}\n"}
{
  use List::Util "shuffle";

  @items = shuffle(@items);

  my $result = " \\item ".
               join("\n \\item ", @items).
               "\n\\end{enumerate}";
  undef @items;
  return $result;
}
```

Figure 18: A PerlTEX command which stores an item of a `shuffle` environment.

10 Summary

TEX provides powerful ways to format text and to perform general-purpose computations. For many users, however, the techniques required to access the computational features of TEX are cumbersome. By providing a bridge between TEX and Perl, PerlTEX makes these computations more accessible.

Perl's widely acknowledged strengths, including extensive libraries, support for regular expressions, a rich collection of string primitives, and familiar control structures, make PerlTEX a natural candidate for the LATEX user seeking finer control over typesetting tasks.

References

[1] Sean Burke. *Perl & LWP*. O'Reilly, 2002.

[2] Alexander Grahn. *The animate package*. `http://www.ctan.org/tex-archive/macros/latex/contrib/animate`.

[3] Donald E. Knuth. *The TEXbook*. Addison-Wesley, 1986.

[4] Scott Pakin. PerlTEX: Defining LATEX macros using Perl. *TUGboat*, 25(2):150–159, 2004.

[5] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and Fractals*. Springer-Verlag, 2004.

[6] Till Tantau. *TikZ and PGF manual*. `http://sourceforge.net/projects/pgf/`.

[7] Till Tantau. *User's Guide to the Beamer Class*. `http://latex-beamer.sourceforge.net`.

[8] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl, Third Edition*. O'Reilly, 2000.