## *TUGboat* online, reimplemented

Karl Berry

**Abstract**

This article discusses updates to the data and code for creating the online *TUGboat* HTML files which are automatically generated: the per-issue tables of contents and the accumulated lists across all issues of authors, categories, and titles. All source files, both data and code, are available from `https://tug.org/TUGboat`, and are released to the public domain.

### 1 Introduction

Since 2005, *TUGboat* has had web pages generated for both per-issue tables of contents and accumulated lists across all issues of authors, categories, and titles. David Walden and I worked on the process together and wrote a detailed article about it [1]; Dave wrote all of the code. More recently, we wanted to add some features which necessitated writing a new implementation. This short note describes the new work.

The basic process remains unchanged. To briefly review from the earlier article:

- For each issue, a source file `tb`⟨*n*⟩`capsule.txt` (*n* being the *TUGboat* issue sequence number), which is essentially written in TEX (it is used to create the contents by difficulty on the inside back cover), is converted to an HTML file named `contents`⟨*vv*-*i*⟩`.html` (for issue number *i* in volume *vv*). These `contents*.html` files are intended to closely mimic the printed table of contents (the back cover) with respect to ordering of items, variation in author's names, category names, etc., with only typos corrected.
- The translation from TEX to HTML is done by the code here, not using TEX4ht or any other tool; the overall HTML structure is written directly by the program. The translation is informed by two files (`lists-translations.txt` and `lists-regexps.txt`), which (simplistically) map TEX input strings to HTML output strings.
- Finally, three files are produced accumulating all items from across all issues: `listauthor.html`, `listkeyword.html`, and `listtitle.html`; each is grouped and sorted accordingly. (These cumulative lists are the primary purpose for developing the program in the first place.) In these files, in contrast to the per-issue contents, many unifications are done (directed by a third external data file, `lists-unifications.txt`), so that articles written under the names, say, "Donald E. Knuth", "Donald Knuth", "Don Knuth", etc., all appear together. Similarly, many variations in category names, and related categories, are merged.
- The translations are applied first, then the regular expressions (regexps), and finally the unifications.

### 2 General implementation approach

Both the old implementation and the new are written in Perl, though they do not share any code. I chose Perl simply because it is the scripting language in which I am most comfortable writing nowadays. There was no need to use a compiled language; the total amount of data is small by modern standards. Readability and maintainability of the code are far more important than efficiency.

I wrote the new implementation as a straightforward, if perhaps old-fashioned, program. I did not see the need to create Perl modules, for example, since the program's job is a given, and the chance of any significant reuse outside the context of *TUGboat* seems small indeed. All the code and data are released to the public domain, so any subroutines, utility functions, fragments, or any other pieces of code or data useful elsewhere can be copied, modified, and redistributed at will.

As mentioned above, the capsule source files are essentially TEX. For example, here is the capsule entry from `tb123capsule.txt` for a recent article:

```
\capsule{}
  {Electronic Documents}%add|Software \& Tools
  {Martin Ruckert}
  {\acro{HINT}: Reflowing \TeX\ output}
  {postponing \TeX\ page rendering to ...}
  {217-223}
  {/TUGboat/!TBIDENT!ruckert-hint.pdf}
```

The meaning of the fields is described in the previous article, but is probably evident enough just from the example. For our present purposes, let's just observe the brace-delimited arguments and general TEX markup. To parse this, the present program uses one non-core Perl module (and only this one): `Text::Balanced` (`metacpan.org/pod/Text::Balanced`), which does basic balanced-delimiter parsing. (The previous implementation did the parsing natively, more or less line-based.)

Perl has several modules to do this job; I chose this one because (a) it had a reasonably simple interface, and (b) it could return the non-balanced text between arguments, which was crucial for our format, since we use formatted comments as directives with additional information for the `lists*` files — as seen above with the `%add|...` extra category. Only three directives have been needed so far: to add and replace categories, and to add authors. They are

crucial for making the accumulated lists include all the useful information.

Each capsule turns into a Perl hash (associative array), and each issue is another hash, including pointers to all its capsules, and so on. In general, the amount of data is so small that memory usage was a non-issue.

Perhaps it would be a better general approach to completely reformat the TEX source into a non-TEX format (YAML, for instance) and then parse that; and perhaps some future *TUGboat* worker will feel inspired to do that. It would not be especially hard to have the current implementation output such a conversion as a starting point. I merely chose to keep the process more or less as it has been.

## 3 Cleaning up capsule sources and output

I did take the opportunity to clean up the capsule source files, e.g., using more consistent macro abbreviations, adding missing accents to authors' names, correcting typos, etc. The balanced-brace parsing regime meant that unbalanced braces got found, of which there were several.

I also added consistency checks in the code, so that, for example, a new category name that we happen to invent for a future issue will get reported; such cases should (probably) be unified with an existing category. Many unifications of existing categories and authors were also added.

Another part of the cleanup was to ensure that the page number of each item is unique; when two items start on the same page, internally we use decimals (100 and 100.5, say) to order them. This is done with a macro `\offset`. Naturally such decimals are not shown in either the TEX or HTML output. They are necessary in order to have a unique key in all our various hashes, and for sorting.

Speaking of sorting, I wanted the new output for the accumulated lists to be stably sorted, so that the results of any code or data changes could be easily diffed against the previous output. So now the sorting for a given entry is reliably by volume, then issue, then page (and first by title for the title list); having unique internal page values was also a prerequisite for the stable sort.

Another minor point about the HTML output is the anchor names: we intentionally reduce all anchor identifiers (author names, titles, etc.) to 7-bit ASCII — indeed, only letters, numbers, periods, and commas. For example, Herbert Voß's items in *TUGboat* are available at `tug.org/TUGboat/Contents/listauthor.html#Voss,Herbert`. (Sorry, Herbert.) Similarly, if an anchor starts with a non-letter, it is prefixed by `t_`. Although HTML permits general Unicode in anchor names nowadays, this has not always been the case, and regardless, for ease of copying, use in email, etc., this seemed the most useful approach.

## 4 Data files `lists-*.txt`

The external data files `lists-unifications.txt` and `lists-translations.txt` that play a part in all these conversions are described in the earlier article. The third file mentioned above, `lists-regexps.txt`, is new in this implementation. Here are some example entries from each.

### 4.1 lists-unifications.txt

Examples from `lists-unifications.txt`:

```
Dreamboat
    Expanding Horizons
    Future Issues
...
Max D&iacute;az
    M. D&iacute;az
```

The left-justified line shows the name as it should be shown, and following indented lines show alternate names as they are found. We unify categories and names, as shown here.

The Díaz example also shows that we do unifications after the translation to HTML, so both the canonical name and the alternates are expressed that way, not in TEX. Thus, the exact form of the translation matters (whether í translates to `&iacute;` or `&#xed;` or `&#x00ED;` or a literal UTF-8 `í` or ...) and has to match with the `lists-translations.txt` entries. Examples from there are next.

### 4.2 lists-translations.txt

Examples from `lists-translations.txt`:

```
\'{\i}||&iacute;||i
\TUG{}||TUG
```

Each line is two or three strings, separated by a `||` delimiter. The first element is what's in the TEX source; the second is the HTML to output, and the third is the plain text conversion for sorting and anchors. If the third element is absent (as in the `\TUG` line above), the second element is used for both plain text and HTML.

### 4.3 lists-regexps.txt

For `lists-regexps.txt`, the general form is similar to `lists-translations.txt`, with a left-hand side and right-hand side separated by the same `||` delimiter. But here, the lhs is a regular expression, and the rhs is a replacement expression:

```
\\emph\{(.*?)\}||"<i>$1</i>"
\{\\it\s*(.*?)\}||"<i>$1</i>"
```

All that punctuation may look daunting, but if taken a bit at a time, it is mostly standard regular expression syntax. The above two entries handle the usual LaTeX `\emph{...}` and plain `{\it ...}` italic font switching (with no attempt to handle nested `\emph`, as it is not needed). Both syntaxes for font switching are prevalent throughout the capsule sources.

The `.*?` construct in the left hand side may be unfamiliar; this is merely a convenience meaning a "non-greedy" match — any characters up until the first following right brace (the `\}` means a literal right brace character). A `.*` without the `?` would match until the *last* following right brace.


On second glance, what also may seem unusual is the rhs being enclosed in double quotes, `"..."`, specifying a Perl string constant. Why? Because, ultimately, this is going to turn into a Perl substitution, `s/`⟨*lhs*⟩`/`⟨*rhs*⟩`/g` (all substitutions specified here are done globally), but initially the lhs and rhs have to be read into variables — in other words, string values. But we don't want to evaluate these strings when they are read from the `lists-regexps` file; the `$1` in the rhs needs to refer to what is matched by the `(...)` group on the lhs when the substitution is executed. This turns out to be a programming exercise in layers of evaluation.


The simplest way I found to do it was to use a Perl feature I had never before needed, or even heard of, in my 30-odd years of using Perl since it first appeared: including the `/ee` modifier on the substitution, as well as the `/g`. I won't try to explain it here; for the curious, there is a discussion at `stackoverflow.com/q/392644`, in addition to the Perl manual itself (`perlre`).

## 5 Performance and profiling

Although I said above that efficiency was not an issue, that is not quite true. Especially during development and debugging, doing a test run must not take too long, since it gets done over and over. My extremely naive initial version took over 45 seconds (on my development machine, which is plenty fast) to process the ≈120 *TUGboat* issues — much too frustrating to be borne.

The Perl module `Devel::NYTProf` turned out to be by far the best profiling tool available. (By the way, NYT stands for *New York Times*; a programmer there did the initial development.) Unlike other Perl profiling tools, it shows timing data per individual source line, not just functions or blocks.

Using that, it turned out that almost all the time was being consumed dealing with the substitutions from the `lists-*` files, since the strings were being read at runtime, instead of being literal in the source code. The easy step of "precompiling" the regular expressions after reading the files, with `qr//`, resulted in the total runtime dropping an order of magnitude, to under 4 seconds. So development could proceed without any major surgery on the code or data structures.

## 6 Conclusion

The ad hoc conversion approach described here is viable only because we have complete control over the not-very-complicated input, and desirable mainly because we want complete control over the output. I did not want to struggle with any tool to get the HTML I wanted, namely to be reasonably formatted and otherwise comprehensible, and not making significant use of external resources or JavaScript.

Although changes and new needs are inevitable, I hope the program and data will be sufficiently robust for years to come.

## References

[1] K. Berry and D. Walden. TUGboat online. *TUGboat* 32(1):23–26, 2011. `http://www.tug.org/TUGboat/tb32-1/tb100berry.pdf`

◇ Karl Berry
`https://tug.org/TUGboat`

Editor's note: Until last year, the comprehensive BibTeX database for TUGboat, `tugboat.bib`, which Nelson Beebe maintains on the servers at the University of Utah was derived programmatically from the TUGboat contents files. Earlier this year, Nelson modified the procedures to instead use the HTML files (generated as described here) for the issues. This method should not only be easier to maintain, but also contain additional information.

New labeling conventions have been applied to newly created entries; these follow Nelson's BibNet Project. Labels of older entries are frozen in the old form (e.g., `Knuth:TB2-3-5`), so as not to invalidate user documents that cite such labels.

These files are on both the Utah web server, at `www.math.utah.edu/pub/tex/bib`, and the parallel FTP server. In addition to `tugboat.bib`, these associated files are present: `.def`, `.dvi`, `.html`, `.ltx`, `.pdf`, `.ps.gz`, `.ps.xz`, `.sok`, and `.twx`. (To fetch the whole collection under `ftp`, issue the command "`mget tugboat.*`".)

The collection is also on CTAN in the area `info/biblio`, along with `bib` files on additional topics. Some of these, including `tugboat.bib`, are also in TeX Live. — Barbara Beeton