# TUGboat

Volume 35, Number 1 / 2014

## TEX Users Group

*TUGboat* (ISSN 0896-3207) is published by the TEX Users Group.

### Memberships and Subscriptions

2014 dues for individual members are as follows:

- Regular members: $105.
- Special rate: $75.

The special rate is available to students, seniors, and citizens of countries with modest economies, as detailed on our web site. Also, anyone joining or renewing before March 31 received a $20 discount.

Membership in the TEX Users Group is for the calendar year, and includes all issues of *TUGboat* for the year in which membership begins or is renewed, as well as software distributions and other benefits. Individual membership is open only to named individuals, and carries with it such rights and responsibilities as voting in TUG elections. For membership information, visit the TUG web site.

Also, (non-voting) *TUGboat* subscriptions are available to organizations and others wishing to receive *TUGboat* in a name other than that of an individual. The subscription rate is $105 per year, including air mail delivery.

### Institutional Membership

Institutional membership is a means of showing continuing interest in and support for both TEX and the TEX Users Group, as well as providing a discounted group rate and other benefits. For further information, see `http://tug.org/instmem.html` or contact the TUG office.

### Trademarks

Many trademarked names appear in the pages of *TUGboat*. If there is any question about whether a name is or is not a trademark, prudence dictates that it should be treated as if it is. The following list of trademarks which commonly appear in *TUGboat* should not be considered complete.

TEX is a trademark of American Mathematical Society.
METAFONT is a trademark of Addison-Wesley Inc.
PostScript is a trademark of Adobe Systems, Inc.

[printing date: April 2014]

Printed in U.S.A.

### Board of Directors

**Addresses**
TEX Users Group
P. O. Box 2311
Portland, OR 97208-2311
U.S.A.

**Telephone**
+1 503 223-9994

**Fax**
+1 815 301-3568

**Web**
`http://tug.org/`
`http://tug.org/TUGboat/`

**Electronic Mail**
(Internet)

General correspondence, membership, subscriptions:
`office@tug.org`

Submissions to *TUGboat*, letters to the Editor:
`TUGboat@tug.org`

Technical support for TEX users:
`support@tug.org`

Contact the Board of Directors:
`board@tug.org`

Trying to do everything inside TeX may be a good sport and educational activity but very often far from being the best solution for real needs. :-)

Michal Jaegermann
`comp.text.tex`, 26 October 1996

# TUGboat

COMMUNICATIONS OF THE TEX USERS GROUP

Editor Barbara Beeton

Volume 35, Number 1 • 2014
Portland • Oregon • U.S.A.

**_TUGboat_ editorial information**
This regular issue (Vol. 35, No. 1) is the first issue of the 2014 volume year.

*TUGboat* is distributed as a benefit of membership to all current TUG members. It is also available to non-members in printed form through the TUG store (`http://tug.org/store`), and online at the *TUGboat* web site, `http://tug.org/TUGboat`. Online publication to non-members is delayed up to one year after print publication, to give members the benefit of early access.

Submissions to *TUGboat* are reviewed by volunteers and checked by the Editor before publication. However, the authors are still assumed to be the experts. Questions regarding content or accuracy should therefore be directed to the authors, with an information copy to the Editor.

**Submitting items for publication**
Proposals and requests for *TUGboat* articles are gratefully accepted. Please submit contributions by electronic mail to `TUGboat@tug.org`.

The second issue for this year is expected to be the TUG 2014 proceedings (`http://tug.org/tug2014`). The deadline for receipt of final papers for that issue is August 11. The third issue deadline is October 3.

The *TUGboat* style files, for use with `plain` TeX and LaTeX, are available from CTAN and the *TUGboat* web site. We also accept submissions using ConTeXt. Deadlines, tips for authors, and other information:
`http://tug.org/TUGboat/location.html`

Effective with the 2005 volume year, submission of a new manuscript implies permission to publish the article, if accepted, on the *TUGboat* web site, as well as in print. Thus, the physical address you provide in the manuscript will also be available online. If you have any reservations about posting online, please notify the editors at the time of submission and we will be happy to make special arrangements.

**_TUGboat_ advertising**
For advertising rates and information, including consultant listings, contact the TUG office, or see:
`http://tug.org/TUGboat/advertising.html`

## Ab Epistulis

Steve Peter

Not long ago, I saw a post on one of the discussion lists asking how to convert a TeX document into Word, "because that's what publishers want." I've recently been working with a group at work to revise our production guidelines for TeX manuscripts, and I began to ponder the question in more depth. Why do publishers want Word (if indeed they do), and what can we do as a community to change that?

For math-heavy books, whether math, physics, or economics, we not infrequently work with TeX files throughout the production process, whether the book is ultimately provided in camera-ready copy by the author, or is produced by a TeX-enabled compositor. Without a doubt, the biggest pain point in the process is with copyediting (for all books) and indexing (for books where the author does not supply camera-ready copy). As the publishing industry moved to outsource these two processes, a vast army of freelance and independent contractors arose, but very few saw fit to gain expertise in TeX. In fact, expertise per se isn't even required, just enough knowledge to be able to work directly in the files without breaking too much.

In essence, it isn't necessarily that publishers are demanding Word, it's the freelance community that is requiring it, and the publishers lack a pool of TeX-savvy talent to draw from to be able to break that dependency. It seems to me that this represents an opportunity to expand our community.

For years, we've been growing by word of mouth among colleagues in the academic disciplines. One mathematician tells another about TeX's abilities in handling all sorts of complex equations; a historian tells another how BibTeX or JabRef can handle the complexities of managing a bibliographic database (no math here, just academic writing); and so on.

Now, we need to engage another sub-community of the (academic) publishing world: the freelancers. Copyeditors and indexers need to know that they can gain a competitive advantage by learning at least enough TeX to be able to work directly in source files. (Speaking from personal experience, more and more of my own freelance work has gone over to TeX-based copyediting, away from TeX programming.)

How will this engagement happen? The key is going to be education, especially in a casual way. If you do encounter a freelancer curious about TeX, show them the simple stuff to dispel the fear and uncertainty. We don't need to turn freelance copyeditors or indexers into hardcore TeX experts who shun any trace of commercial software. We do need to show them just enough to be able to do their specialized jobs as part of a TeX-based workflow.

If we can enable a painless workflow, the publishers will come.

◇ Steve Peter
Princeton University Press
`president (at) tug dot org`
`http://tug.org/TUGboat/Pres`

## Editorial comments

Barbara Beeton

### Updike prize for student type design

An annual prize for student type designers has been announced by the Daniel Berkeley Updike Collection at the Providence Public Library. The basis of the Collection is the archives of the eponymous 20th century printer and proprietor of the Merrymount Press in Boston.

The prize requires that the student make at least one visit to the Updike Collection within 18 months of their application, and must be enrolled in an undergraduate or graduate program during the time of their visit.

The first prize includes $250 and complimentary admission to the 2015 TypeCon, organized by SOTA, the Society of Typographic Aficionados.

Additional information, and the application form, can be obtained from this web site:
`http://www.provlib.org/updikeprize`

### Talk by Matthew Carter

An exhibition celebrating the 200th anniversary of the death of typographer Giambattista Bodoni, and the launch of the Updike Prize was held on February 27. The speaker was Matthew Carter; his subject was "Genuine Imitations: A Type Designer's View of Revivals".

Carter took three of his typeface designs as the material for his talk: Snell Roundhand, Mantinia, and the design he created for Yale University.

The first two designs were based on non-typographic sources; the first was inspired by the work of an English writing master, Charles Snell, and the second, by stone carving in a more-or-less traditional Roman style.

In 1965, when Linotype was converting its fonts from metal to images on film, Carter was invited to design a new script font. It's possible to do many things with film that can't be done with metal. For example, character widths aren't limited to what can fit in a rather narrow rectangle cast as part of a unitary "line of type". (That process doesn't even permit the kerns that can be carved into the corners of hand-set type.)

Charles Snell, a 17th century English writing master, published manuals that included detailed diagrams and instructions for writing script with a quill pen. The flowing *f*s and the tails on the *g* and *y* are representative of the style. An image of Snell's lowercase script can be seen at `http://www.paulshawletterdesign.com/wp-content/uploads/2011/05/Charles-Snell-1714.jpg`, and Carter's re-imagined version at `http://luc.devroye.org/MatthewCarter-RoundhandBT-afterCharlesSnell.gif`.

Andrea Mantegna, a 15th century Italian artist and student of Roman archaeology, was the inspiration for the Mantinia font. The foundation post of Mantegna's house in Mantua (shown in one of Carter's slides) prominently displays the carved name "Mantinia" (the Latin form of Mantegna) as well as other inscriptions in the style typical of Roman monuments and gravestones. All the letters are uppercase. Whether to save space, balance the shape of the inscriptional lines, or for some other reason, two or three adjacent letters are sometimes combined in unusual ways. Some smaller letters occur as well — inscriptional "small caps" — but, rather than being aligned at the baseline with the larger letters, they are aligned at the top.

A more recent use of this style of carving can be found on the panels above the windows of the Boston Public Library (see Fig. 1). These variations gave Carter a model on which to base the many unusual and playful ligatures and top-aligned "lowercase" found in the finished Mantinia font.

The third font was the one designed for Yale University. Inspired by Bembo, this font serves all

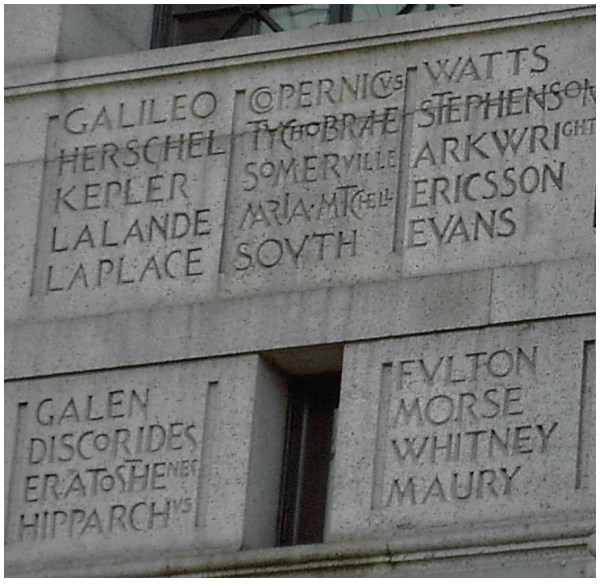


**Figure 1**: A panel from the Boston Public Library showing the names of noted astronomers.
Photo by Jascin L. Finger, curator of the Maria Mitchell House on Nantucket; used with permission. Maria Mitchell was an American astronomer, only the second woman to be recognized as the discoverer of a comet; the first was Caroline Herschel, sister of William Herschel (discoverer of Uranus).

typographic functions of the University, from letterhead to cast bronze letters on building façades to signage on recycling and rubbish baskets on the campus. Images of the Yale font and notes on its history and development can be seen at `http://www.yale.edu/printer/typeface/typeface.html`.

### R.I.P. Mike Parker (1929–2014)

Just a few days before Matthew Carter's talk, Mike Parker died. Parker, as director of type development at Linotype, invited Carter to design the script font that became Snell Roundhand, persuaded Carter to join Linotype as chief designer, and later, in 1981, left Linotype with Carter to found Bitstream.

Parker was largely responsible for bringing font production from the world of metal to film, and from film to digital formats. He was an enthusiastic and influential proponent of Helvetica, and put forth the proposition that Times New Roman had in fact been designed by Starling Burgess, not by Stanley Morison.

A sympathetic obituary appeared in the March 8th edition of *The Economist*, and can be accessed on the web.

### Turing Award for Leslie Lamport

The Association for Computing Machinery will present the 2013 A.M. Turing Award to Leslie Lamport for "advances in reliability and consistency of computing systems". The Turing Award is given for major contributions of lasting importance to computing. The citation can be read here:
`http://techpolicy.acm.org/blog/?p=3641`.

Not a single word recognizes his creation of LaTeX, the accomplishment for which this community knows him best. With this award, Leslie joins Don Knuth, who received the 1974 Turing Award. And all for achievements not related to TeX.

### *TAOCP* volume 1 issued as an ebook

The InformIT arm of Pearson Education (parent of the group that includes Addison-Wesley) has announced that *The Art of Computer Programming* Volume 1 ebook is now available for sale. Their news release further states that

> Only InformIT provides this eBook in three formats — EPUB, MOBI, & PDF — together for one price. So you can buy it once and get it on any device including your PC or eReader of choice.
>
> We will be releasing the other volume eBooks throughout the year and currently hope to have Vol. 2 for you in just a couple of months.

Although Don's web page (`http://www-cs-faculty.stanford.edu/~uno/abcde.html`) explains that the "spiffy new versions" of the *C&T* volumes were "produced entirely with technology that can be expected to last for many generations", there is no hint that they might sometime be available in electronic form.

### Other items worth a look

As a follow-up to last year's interview of Chuck Bigelow, here is an essay in which he shares some earlier history, written for the centennial of Reed College: "Rescued from a Life of Crime" (`http://www.reed.edu/reed_magazine/september2011/articles/features/bigelow/bigelow.html`). Who knew?

The Hamilton Wood Type Museum has been mentioned before in this column. Evicted from its original site, it is now open in a new location in Two Rivers, Wisconsin: `http://woodtype.org`. A documentary, "Typeface", telling the story of this museum, was presented on the Sundance channel in the U.S.; information about the video can be found at `http://typeface.kartemquin.com/about`.

The debate about whether or not there should be a wider space after a period goes on: "Space Invaders, Why you should never, ever use two spaces after a period." in *Slate* (`slate.com/articles/technology/technology/2011/01/space_invaders.html`). Actually, this isn't new, but it's just been called to my attention. As a TeX user, I'm spoiled: I can type two spaces after the periods that end sentences in my (monospace) emacs window, and TeX will do my bidding, whether using the default U.S. style or `\frenchspacing` to treat all spaces alike. Using a monospace font while editing makes sense, since it makes a file easier to read (at least for these old eyes), and permits an author to align or indent to illuminate structure. As I see it, there *are* times when the wider spaces are valuable typographically. Maybe not after every sentence, but if your text says "…etc. W. H. Auden says …", where is the end of that sentence? The *real* culprit (which the author of this screed doesn't even mention) is software that sets multiple spaces *in text* as multiple spaces, not singles. Software is supposed to make life easier for the user, not harder, but some things seem to be going backwards these days.

To end this installment, here's an oldie, but a real goodie: setting 24pt type in an ogee curve (`flickr.com/photos/sos222/12391791743/`). The old guys really did know what they were doing.

◇ Barbara Beeton
`http://tug.org/TUGboat`
`tugboat (at) tug dot org`

# The TEX tuneup of 2014

Donald Knuth

If you ask the Wayback Machine to take you back to the home page

```
http://www-cs-faculty.
        stanford.edu/~knuth/abcde.html
```

of *The TEXbook* and my other books on *Computers & Typesetting*, as that page existed on 16 January 1999, you'll find the following remarks:

> I still take full responsibility for the master sources of TEX, METAFONT, and Computer Modern. Therefore I periodically take a few days off from my current projects and look at all of the accumulated bug reports. This happened most recently in 1992, 1993, 1995, and 1998; following this pattern, I intend to check on purported bugs again in the years 2002, 2007, 2013, 2020, etc. The intervals between such maintenance periods are increasing, because the systems have been converging to an error-free state.

And if you fast-forward nine more years, you can find a *TUGboat* article called "The TEX tuneup of 2008" [4], which describes the changes that were made to TEX and its companion systems based on the comments from users that were received during the years 2003, 2004, 2005, 2006, and 2007. That article ended as follows:

> So now I send best wishes to the whole TEX community, as I leave for vacation to the land of *TAOCP* — until 31 December 2013. Au revoir!

Hello again, dear friends, *allô*! Here is the sequel.

On 31 December 2013, Barbara Beeton duly forwarded to me a well-organized collection of materials covering more than two dozen potentially troublesome topics that had been submitted for consideration during the years 2008, 2009, 2010, 2011, 2012, and 2013. This was the residue of hundreds of items that had been carefully filtered by a team of expert volunteers, who had worked hard to minimize the effort that I would need to devote to this project. (I can't possibly thank all the volunteers individually; but Donald Arseneau, Karl Berry, Peter Breitenlohner, and Bogusław Jackowski deserve particular commendation.)

As in 2008, both TEX and METAFONT have changed slightly and gained new digits in their version numbers. But again, the changes are essentially invisible. I can't resist quoting another paragraph from [4], because it reflects my unwavering philosophy (see [3]):

> The index to *Digital Typography* lists eleven pages where the importance of stability is stressed, and I urge all maintainers of TEX and METAFONT to read them again every few years. Any object of nontrivial complexity is non-optimum, in the sense that it can be improved in some way (while still remaining non-optimum); therefore there's always a reason to change anything that isn't trivial. But one of TEX's principal advantages is the fact that it does not change — except for serious flaws whose correction is unlikely to affect more than a very tiny number of archival documents.

Users can rest assured that I haven't "broken" anything in this round of improvements. Everyone can upgrade at their convenience.

## TEX Version 3.14159265

Let's get down to specifics. The new version of TEX differs from the old only with respect to the "null control sequence" `\csname\endcsname`, which has been a legal construct since version 0.8 (November 1982) although almost nobody uses it. Oleg Bulatov noticed in September 2008 that TEX's `\message` operation has curiously inconsistent behavior: Suppose you say

```
\def\\#1{\message{#1bar}}
\def\surprise{wunder}
\let\foo=!
```

(for example). Then

```
\\\surprise                  gives wunderbar
\\\over                      gives \over bar
\\\foo                       gives \foo bar
\\{\csname 6\endcsname}      gives \6bar
\\{\csname fu\endcsname}     gives \fu bar
```

as messages on your terminal and in your log file. But '`\\{\csname\endcsname}`' unfortunately gives

```
\csname\endcsnamebar
```

because I forgot to insert a space when I coded this part of the *print_cs* routine (see [B], §262). So Oleg has won a check for \$327.68 [1]. Of course I hope that this turns out to be the "historic" final bug in TEX. (It's the 947th; see [3], page 662.)

Henceforth '`\\{\csname\endcsname}`' will give

```
\csname\endcsname bar
```

and everybody will be happy. This corrected behavior does not simply affect TEX's messages; the name of a control sequence can also get into documents, for example via `\write` or `\meaning`. But the change surely won't ruin your archived works.

## METAFONT Version 2.7182818

The historic final (I hope) bug in METAFONT was discovered during June 2008 by the longstanding TeX contributor Eberhard Mattes. The error that he brought to light is easier to describe than the TeX error discussed above, but it was much more subtle to detect: Whenever previous versions of METAFONT have transformed **pencircle** into an axially symmetric pen whose polygon has no point on the $x$-axis, the algorithm in §536 of [D] has "leaked memory," by forgetting to reclaim seven words that had been allocated for the omitted point. This happened, for instance, with one of the pens in exercise 16.2 of [C], and in my original `TRAP` test [2] for METAFONT; so I should have discovered the problem long ago. Eberhard noticed that the METAFONT program

```
pen p;
forever:  showstats;
  p := pencircle scaled 1.4; endfor
```

would abort with METAFONT's capacity exceeded—although it did take quite awhile to overflow 3 million words of memory on my current home system—and he also figured out how to cure the problem. For this he amply deserves his new reward in [1].

## Computer Modern

No changes have been made to the Computer Modern fonts of 2008, although I did delete a few bytes of redundant source code and alter two names.

John Bowman noticed a tiny bump that appears near the top right serif when an italic '$K$' is greatly magnified, and Jacko discovered the underlying reason: Part of the stroke of this slanted letter is drawn with a circular pen, but it joins up with outlines that are slanted (hence not true circles). The same tiny bumps can therefore by observed also in various other italic and slanted letters, such as $A$, $V$, $W$, $X$, $Y$, when enlarged.

But those bumps are even less visible than the mispositioned bulbs that I discussed in [4]. And in fact I've even become somewhat fond of such little glitches, now that I've been learning to appreciate the Japanese concept of *wabi-sabi*.

Thus I've decided that the Computer Modern fonts are to be forever frozen in their present form, especially now that the definitive description in the latest printing of [E] has become available.

## TeXware and METAFONTware

I made minor updates to the master `web` files for five other programs, namely `gftopk`, `pltotf`, `tftopl`, `vftovp`, and `vptovf`, in order to make them more robust in the presence of weird input files. (These changes had in fact already been made in recent editions of TeX Live; now they are in some sense "official.") Here is a current list of all the `web` files for which I have traditionally been responsible:

| name | current version | date |
|---|---|---|
| `dvitype.web` | 3.6 | December 1995 |
| `gftodvi.web` | 3.0 | October 1989 |
| `gftopk.web` | 2.4 | January 2014 |
| `gftype.web` | 3.1 | March 1991 |
| `mf.web` | 2.7182818 | January 2014 |
| `mft.web` | 2.0 | October 1989 |
| `pltotf.web` | 3.6 | January 2014 |
| `pooltype.web` | 3.0 | September 1989 |
| `tangle.web` | 4.5 | December 2002 |
| `tex.web` | 3.14159265 | January 2014 |
| `tftopl.web` | 3.3 | January 2014 |
| `vftovp.web` | 1.4 | January 2014 |
| `vptovf.web` | 1.6 | January 2014 |
| `weave.web` | 4.4 | January 1992 |

## Typographic errors and other blunders

So far I've only been discussing potential anomalies in the software. But of course people have also reported problematic aspects of the documentation—which may actually be the hardest thing to get right. Even *The TeXbook* [A], which has been under intense scrutiny for more than thirty years, was not free of hitherto-unperceived defects.

Altogether I made corrections to each of [A], [B], [C], [D], and [E], enough to represent $23.68 in eleven new reward checks. The most significant of these changes can be seen from the home page cited above, if you click to get the PDF errata file and scan for corrections dated in 2014.

## The master sources

The backbone of the TeX system, for the past 25 years or so, has been a collection of 178 files, mostly with names of the forms `*.web`, `*.tex`, and `*.mf`. These files contain almost exactly 7 megabytes altogether; and the new changes have altered about 3500 of those bytes. Thus it appears that the TeX system was 99.95% correct in 2008, if it is 100% correct today.

The master files, together with a bunch of errata files that document past history, can be downloaded from the ftp server `cs.stanford.edu`, which accepts '`anonymous`' as a login name. They're collected together in a single compressed file

`pub/tex/tex14.tar.gz`,

which you can compare if you like to the older files `pub/tex/tex08.tar.gz`, `pub/tex/tex03.tar.gz`. The latest versions of individual files can of course also be found in the CTAN archive.

As I did in [4], I'll mention here the names of all files that have changed in some way during the latest go-round:

`tex/texbook.tex` % source file for [A]
`tex/tex.web` % master file for TeX in Pascal
`tex/trip.fot` % torture test terminal output
`tex/tripin.log` % torture test first log file
`tex/trip.log` % torture test second log file
`tex/trip.typ` % torture test output of DVItype
`texware/pltotf.web` % master file for PLTOTF
`texware/tftopl.web` % master file for TFTOPL
`mf/mfbook.tex` % source file for [C]
`mf/mf.web` % master file for METAFONT in Pascal
`mf/trap.fot` % torture test terminal output
`mf/trapin.log` % torture test first log file
`mf/trap.log` % torture test second log file
`mf/trap.typ` % torture test output of DVItype
`mfware/gftopk.web` % master file for GFTOPK
`cm/romanu.mf` % master file for Computer Modern Roman uppercase
`cm/symbol.mf` % master file for Computer Modern Roman symbols
`etc/vftovp.web` % master file for VFTOVP
`etc/vptovf.web` % master file for VPTOVF
`lib/manmac.tex` % macros for [A] and [C]
`errata/errata.nine` % changes to [A] between 1992 and 1996
`errata/errata.tex` % changes to [A]–[E] since 2001
`errata/tex82.bug` % changes to `tex.web`
`errata/errorlog.tex` % one-per-line annotated summaries of those changes
`errata/mf84.bug` % changes to `mf.web`

(Notice that the basic macro files for plain vanilla TeX and plain vanilla METAFONT, `lib/plain.tex` and `lib/plain.mf`, remain unchanged.)

## Questions and answers

Barbara also asked me to answer three questions, which she said "keep coming up in various forums," so that she could point people to the answers if those questions come up again.

(1) How long did it take to typeset *The TeXbook* in the 80s, and how long does it take today?

This question is a bit strange, because anybody who tries to apply TeX to the file `texbook.tex` immediately gets the message '`~\.{This manual is copyrighted and should not be TeXed`', repeated endlessly. Therefore the running time to typeset *The TeXbook* has always been infinite.

On the other hand, I myself have to generate new printings every now and then; and I have a favorite way to get around the booby trap by first typing '`19`' and then typing some other special codes. (I also realize that unscrupulous people might even try to change `texbook.tex`, although that is strictly forbidden. The source code is intended to be *examined*, if desired, but not *executed* or modified except by its author.)

Unfortunately I don't think I ever noted down the running time in the 80s, so I can't give a definitive answer to the question. My recollection is that the entire book took maybe 20 minutes on Stanford's PDP10 mainframe (shared with other users). There was a noticeable slowdown on certain pages — such as page 218, when prime numbers are computed the hard way.

My colleague David Fuchs used *The TeXbook* as a benchmark in 1986, when he was developing MicroTeX (the first version of TeX to run on an IBM PC). A few days ago I asked him if he could remember its speed. He replied that, like me, he had no firm memory of those days, except that MicroTeX could do several pages per minute; and he guessed that it had taken roughly an hour to complete the whole *TeXbook*. His estimate seems right, because *The TeXbook* has nearly 500 pages.

Today, on my home computer (a 3.6 GHz Xeon with 10 MB cache), TeX transforms `texbook.tex` to `texbook.dvi` in 0.3 seconds.

(2) If you were designing TeX today, would you still use `\over` and friends, rather than something like `\frac{...}{...}`, when the latter would avoid the necessity of `\mathchoice` and `\mathpalette`?

This question, from `tex.stackexchange.com`, also quoted from page 151 of [A]:

> `\mathchoice` is somewhat expensive in terms of time and space, and you should use it only when you're willing to pay the price.

And well, I guess that quote implies my answer. For I was clearly willing to pay the price in 1982, so I'm certainly willing to pay zero today!

I suppose there are some people in the world who prefer expressions like 'sum$(2,3)$' to '$2+3$'; but I'm certainly not among them. Ever since TeX was born, I've been enormously pleased by the ability to write '`2\over3`' or '`n\choose k`' or '`p\atop q`' or $\cdots$, instead of being forced to write something like '`frac{2}{3}`' that would have distracted my attention from the task at hand.

The questioner seems to want to place burdens on all users, rather than on the backs of a few macro-developers.

(3) Why is the default rule thickness 0.4 points?

One of the very first things I did when designing TeX was to choose several publications that represented the highest standards of excellence in mathematical typesetting, and to "reverse engineer" them by making careful measurements of those fine works. (See [3], page 620.) The thickness of rules in *The Art of Computer Programming* was definitive for me. I also knew that Belfast Universities Press was using that value in its typesetting of mathematical journals in 1977.

This question, however, is related to the one sore point with respect to which I wish that I could turn back the clock and redesign TeX from scratch: The actual default rule thickness in TeX is not *exactly* 0.4 printer's points; it is exactly 26214 *scaled* points, where there are 65536 scaled points to every printer's point. Thus the default rule thickness is actually 0.399993896484375 points.

I made the foolish mistake of using binary fractions internally, while providing approximate decimal equivalents in the user interface. I should have defined a scaled point to be 1/100000 of a printer's point, thereby making internal and external representations coincide. This anomaly, which is discussed further in [5], is the only real regret that I have today about TeX's original design.

## Conclusion

The TeX family of programs seems to be healthy as it continues to approach perfection. Volunteers have been stalwart contributors to this success in optimum ways. Stay tuned for The TeX Tuneup of 2021!

## References

[1] The Bank of San Serriffe, account balances. See `http://www-cs-faculty.stanford.edu/~knuth/boss.html` (accessed January 2014).

[2] Donald E. Knuth, *A torture test for METAFONT*. Stanford Computer Science Report 1095 (Stanford, California: Stanford University Computer Science Department, January 1986), 78 pages.

[3] Donald E. Knuth, *Digital Typography* (Stanford, California: Center for the Study of Language and Information, 1999), xvi + 685 pages. CSLI Lecture Notes, no. 78. The second printing (2012) contains numerous corrections.

[4] Donald Knuth, "The TeX tuneup of 2008," *TUGboat* **29** (2008), 233–238. `http://tug.org/TUGboat/tb29-2/tb92knut.pdf`.

[5] Donald E. Knuth, "An earthshaking announcement." *TUGboat* **31** (2010), 121–124. `http://tug.org/TUGboat/tb33-3/tb105knut.pdf`.

[A] Donald E. Knuth, *The TeXbook* (Reading, Mass.: Addison–Wesley, 1984), x + 483 pages. Also published as *Computers & Typesetting*, Volume A. Currently in its 34th printing (paperback) and 19th printing (hardcover).

[B] Donald E. Knuth, *Computers & Typesetting*, Volume B, *TeX: The Program* (Reading, Mass.: Addison–Wesley, 1986), xvi + 594 pages. Currently in its 9th printing (hardcover).

[C] Donald E. Knuth, *The METAFONTbook* (Reading, Mass.: Addison–Wesley, 1986), xii + 361 pages. Also published as *Computers & Typesetting*, Volume C. Currently in its 12th printing (paperback) and 8th printing (hardcover).

[D] Donald E. Knuth, *Computers & Typesetting*, Volume D, *METAFONT: The Program* (Reading, Mass.: Addison–Wesley, 1986), xvi + 560 pages. Currently in its 6th printing (hardcover).

[E] Donald E. Knuth, *Computers & Typesetting*, Volume E, *Computer Modern Typefaces* (Reading, Mass.: Addison–Wesley, 1986), xvi + 588 pages. Currently in its 7th printing (hardcover).

◇ Donald Knuth
`http://www-cs-faculty.stanford.edu/~knuth`

# Making Lists: A Journey into Unknown Grammar

James R. Hunt

## Abstract

Textbooks on technical writing, and academic, corporate and other style guides, often prescribe rules for lists that result in basic grammatical errors. Itemised and enumerated lists are grammatically different, and errors arise when the rules for one type of list are used in constructing the other type. Examples of correct and incorrect usage are given, and ways of avoiding errors are described. The strict application of grammatical rules to list construction reveals some interesting limitations of the list form.

## 1 Introduction

We are all familiar with numbered and bulleted lists, and use them often. After ordinary paragraphs, lists are perhaps the commonest devices used by technical and scientific writers for arranging text on a page. It is an unfortunate fact that textbooks on technical writing, corporate style guides, and even style guides promulgated by learned societies, often prescribe rules for lists that result in grammatical errors. Some of the errors produced by these rules, such as prescribing sentences without correct terminating punctuation, are basic indeed. The purpose of this article is to examine some of these errors, and try to find ways to avoid them.

The approach adopted here is an axiomatic one: a number of assumptions about the desirable properties of a technical text are made, and the consequences of those assumptions are examined.

### 1.1 Preliminaries and Basic Assumptions

First, some preliminaries and plausible basic rules or axioms.

#### 1.1.1 Writing in a Formal Register

Technical works are usually written in a *formal register*. It will be assumed here that a formal document is one that is written in grammatically correct, complete sentences, and is correctly punctuated. The actual quality and style of the language used is not under consideration.

#### 1.1.2 Improving Comprehension

A technical work must be written in complete, grammatically correct sentences. These sentences can be typographically arranged on a page or screen in any way that helps the reader to understand the material. We can, in the interests of clarity and comprehension, decorate the text of the work in any way that we consider necessary: bold, italic, indenting, white space, bullets, numbers, table rules, illustrations, and so on.

Navigational devices, such as headings and subheadings, page numbers, captions for figures and tables, tables of contents, lists of figures and tables, and headers and footers may be applied to the text as the writer considers necessary. These navigational devices may be, but are not necessarily, made from text elements, and may be, but are not necessarily, complete sentences. No matter what they are, text decorations are designed to assist comprehension of the text, and navigational devices are designed to assist in finding specific material in the text. Neither text decorations nor navigational devices are part of the text itself.

#### 1.1.3 Ellipsis

Ellipsis is the omission of elements recoverable from the context, and can involve punctuation as well as words. Restoring the punctuation and missing words (usually conjunctions like *and*) will produce a complete, grammatically correct sentence.

Elliptical sentences should be constructed carefully, in such a way that the reader of the material can without effort reconstruct the full version, and *not gain any conscious impression that the material is grammatically incorrect.*

Elliptical sentences are often used to reduce the apparent complexity of sentences in technical works. In particular, elliptical sentences are commonly used in list constructions.

#### 1.1.4 There is No Such Thing as a Sentence Fragment

Some writers refer to incomplete or otherwise ungrammatical sentences as *sentence fragments*. Now *sentence fragment* is not really a useful concept: incomplete or ungrammatical sentences are not acceptable in the body of a technical work, which must be written in complete sentences or easily-reconstructed elliptical sentences.

There is no requirement for the words in chapter and section headings and captions to constitute complete sentences, and they usually don't. Such headings are only navigation devices, designed to help readers to find their way around a complicated document. Such navigational devices could be called *sentence fragments*, but since *sentence fragments* are limited to navigation devices, there seems to be no real need for a separate name: we could simply refer to headings and captions.

#### 1.1.5 Sentences Cannot be Nested

In the English language, sentences cannot be *nested*, that is, a complete sentence cannot be placed inside another sentence as a standalone entity. (It is of course possible to construct elaborate sentences that contain parts that would be complete sentences if they were written out separately, but that is not the same as a nested sentence.)

#### 1.1.6 A Basic Rule for Writing

Any writing rule that results in grammatical error is useless, and must be discarded. The application of this rule to the construction of numbered and bulleted lists leads to some surprising conclusions.

## 2 How Do You Know It's a List?

Bullets or numbers do not make a list. Word processors and text formatters can generate tagging symbols or sequential numbering for any type of text item that you can specify, and you can place those symbols anywhere in a text. However, these symbols and numbers may not serve the purpose of assisting in comprehending the ideas being presented.

### 2.1 False Lists

There are lists, and there are *false lists.* A false list is a collection of items that do not really belong together, but have bullets or numbers at the left on their first lines. In the Age of PowerPoint, false lists are very common, because it is so easy to add symbols to text. However, the bullet symbols, pointing hands, marching ants, or numbers quite often add nothing to the presentation: plain, undecorated text would have been more informative.

Bad Example 1 shows a false list, of a kind commonly found in PowerPoint presentations.

Bad Example 1: Vertical False List

*Topics for today*

- *What is the Automatic Manual Writer?*
- *Five easy steps to success*
- *How it works*

False lists are not always set out in a neat vertical alignment, as Bad Example 2 shows.

Bad Example 2: Horizontal False List

*There are three options available, namely, (a) running away; (b) staying and hiding; and (c) staying and being brave.*

If the brackets and letters are left out, the sentence is grammatically correct, and a little easier to read. The bracketed letters add nothing to our understanding, and are merely extraneous decoration.

### 2.2 Identifying a List

The items of a list must have some common property: for example, they must all relate to a single idea or task; and the bullets or numbers must in some way improve the clarity of the material being presented. This idea will be progressively refined here.

## 3 Terminology

LaTeX recognises three types of lists: *enumerated* lists, described in Section 3.2, *itemised* lists, described in Section 4, and *definition* lists.

The definition list is something of an oddity, because it appears to be nothing more than a table in disguise. The definition list is considered further in Section 8.

### 3.1 Other Names for Itemised Lists

Itemised lists are sometimes referred to as *bulleted lists* or *unordered* lists. The term *bulleted list* is common but not all that useful, because bulleted lists do not necessarily have text bullets to mark their items: items may be marked by dashes, graphics, or not marked at all but merely set out on separate lines. The term *unordered list* is often used as a more general term than *bulleted list.*

The term *unordered list* is not particularly useful either, since the items of the list are arranged on the page in some order that the author considered useful. The important point is that the order does not matter, in the sense that the items can be rearranged without changing the meaning of the text, even if the clarity of the presentation is reduced as a consequence.

### 3.2 Enumerated Lists

An *enumerated list* is used to set out sequential instructions, or to list components or cases, or to indicate the order of importance of cases. The items in enumerated lists are marked by numbers or letters, or other obviously sequential symbols.

Setting out sequential instructions is fundamentally different from the other two uses: if the enumerated list comprises components or cases, then the order of the items in the list does not matter, because the items can be rearranged without changing the meaning of the presentation. Sometimes the items in an enumerated list, such as a list of component parts of an assembly, can be changed at random without reducing the clarity of the presentation.

## 4 Itemised Lists

Itemised and enumerated lists appear to be similar, but there is a fundamental difference between the two types, and each has its own rules of construction. Itemised and enumerated lists are often confused by technical writers, in the sense that the rules for one type are often applied to the other.

An *itemised* list is simply a visual device for displaying a single, usually complex, sentence on a page. The bullets and other typographical devices, such as indents and line spacing, are not part of the syntactical structure of the sentence itself: they serve only to assist comprehension. It follows from the basic premise stated in Section 1.1.2 that a sentence written as an itemised list should still be grammatically correct when the visual devices are removed. Consider the itemised list shown in Example 1.

Example 1: Simple Itemised List

*Three colours are available:*

- *red,*
- *green, and*
- *blue.*

This itemised list is only a typographical rearrangement of the following sentence.

*Three colours are available: red, green, and blue.*

### 4.1 Commas and Semicolons

In order to avoid ambiguities arising from the repeated use of the word *and* in more complicated examples, we could adopt a convention that sentences are to be subdivided by semicolons, not commas, and thus write out itemised lists in the form shown in the Example 2.

Example 2: Itemised List with Semicolons

*Three colour combinations are available:*

- *red, white, and blue;*
- *blue, green, and yellow; and*
- *green, orange, and white.*

This is an example of the classic, fully punctuated version of the itemised list.

Since an itemised list comprises only one sentence, the word immediately following an item decoration must not be capitalised unless it is a proper noun of some form. Bad Example 3 illustrates a common but incorrect usage.

Bad Example 3: Incorrect Use of Initial Capitals

*This section gives guidelines for:*

- *Creating lists;*
- *Punctuating lists; and*
- *Creating embedded lists.*

### 4.2 Omitting Punctuation

Itemised lists are often written in unpunctuated, or elliptical, form. Consider the itemised list shown in Example 3. (Some textbook writers insist that an elliptical list like this should have a final full stop. But why would you bother?)

Example 3: Itemised List, Unpunctuated (Elliptical)

*Three colours are available:*

- *red*
- *green*
- *blue*

#### 4.2.1 Ellipsis

The unpunctuated list in Example 3 is derived from the fully punctuated version by the process of *ellipsis*.

Some textbooks on technical writing refer to an itemised list formed by ellipsis as a *list of sentence fragments*, but, as was pointed out in Subsection 1.1.4, the concept of a *sentence fragment* is not useful.

If a grammatically correct version cannot be constructed, then we are dealing with a false list.

Many corporate style guides actually specify that the elliptic form of an itemised list *must* be used in place of the fully punctuated original version. There is of course nothing wrong with this specification, as long as we are aware that the elliptical form is a derivative and not the full version.

#### 4.2.2 Ellipsis and Parallel Construction

Many style guides specify that the items in an itemised list should show *parallel construction*, that is, the items should be syntactically similar. Parallel construction is useful because it makes it easier to understand the material presented, and easier to recover the full form of a list from the elliptical form.

## 5 Enumerated Lists

An itemised list is a visual device for displaying a single sentence. In contrast, an *enumerated list*, in numbered or lettered form, is a visual device for displaying a passage of text comprising a number of sentences. The sentences in the text may, but do not necessarily, collectively describe a sequence of events in time or space. Sometimes the form of a enumerated list is used to indicate the number of components in a collection, or cases under consideration. This is usually clear from the context.

If you intend to construct cross-references to individual list items, then those items should be part of an enumerated list.

The commonest example of an enumerated list is a set of instructions that must be performed in a fixed time order. Consider the set of instructions in Example 4.

Example 4: Instructions in Narrative Form

*The instructions for servicing the device are as follows. Open the top panel of the veeblefetzer. Insert the screwdriver into the slot at the left. Turn the screw clockwise until the pressure is released. Close the top panel.*

We usually present a set of instructions like this as a numbered or lettered list with an introductory sentence, or heading, or both. The heading is only text decoration, and as such need not be either grammatically correct or punctuated. The introductory sentence must of course be a complete sentence, ending with a full stop (or question mark, or exclamation mark, if appropriate). An example of an enumerated list with a heading and an introductory sentence is shown in Example 5.

Example 5: Enumerated List with Heading and Introduction

**Servicing the Device**

*To perform a routine service, carry out the following steps.*

1. Open the top panel of the veeblefetzer.
2. Insert the screwdriver into the slot at the left.
3. Turn the screw clockwise until the pressure is released.
4. Close the top panel.

### 5.1 No Colon Introducing an Enumerated List

Many writers end the introductory sentence before an enumerated list with a colon. *This is a grammatical error*: every sentence in a technical work must be complete, and every sentence must end with either a full stop, a question mark, or an exclamation mark, but *never* with a colon. This remarkably common error arises from a confusion of the layout rules for an enumerated list with those for an itemised list.

### 5.2 Decorating the Numbers

The numbers beside the items in an enumerated list are decorations — visual devices that assist the reader in understanding the material — and have no syntactical meaning at all. Any full stops, brackets, bolding, animation or other devices added to the numbers are, from a syntactical point of view, also decorative. The order of the actions is determined by the order of the sentences, not by the numbers on the page: the numbers serve only to reinforce the sequence in the reader's mind.

The use of full stops or brackets after the numbers or letters could suggest to a reader a syntactical structure that does not actually exist, and this is one reason why such decorations of the basic numbers or letters should perhaps be avoided. This is more easily said than done: most writing software will insert these decorations (for example, full stops after item numbers) automatically.

Sometimes, you may need to make the numbers quite prominent: for example, lists of instructions in a user guide may be embedded in masses of explanatory material.

## 6 Complications and Restrictions

Strict application of the rules produces some interesting results. These results are described in more detail in following sections.

An individual item in an itemised list may have another, subsidiary itemised list attached to it. (See Subsection 6.1.)

Putting an explanatory paragraph after an item in an itemised list is a grammatical error. (See Subsection 6.2.)

An individual item in an enumerated list may have an itemised list attached to it. (See Subsection 6.3.)

An individual item in an itemised list *cannot* have a subsidiary, enumerated list attached to it. (See Subsection 6.4.)

### 6.1 Itemised Lists within an Itemised List

It is possible, and quite common, to insert secondary itemised lists under the individual items in another higher level list. There are even widely accepted rules about the selection of bullet points: round bullets at the first level, dashes at the second level, and so on. Recall that an itemised list comprises only one sentence: it follows that all of the items in the list, considered together, must constitute only one sentence. Attempting to follow this rule could easily result in complicated constructions that lack clarity. Example 6 shows a two-level itemised list that follows the rule and is still clear. (A sentence that can be displayed as a three-level itemised list would be rather complicated, at best.)

Example 6: Itemised Lists within an Itemised List

The colour of the body of the device may be:

- a primary colour, which may be one of:
  - *red*;
  - *yellow*; *or*
  - *blue*; *or*
- a pastel colour, which may be one of:
  - *pink*;
  - *pale yellow; or*
  - *azure; or*
- a neutral colour, which may be one of:
  - *beige*;
  - *bone; or*
  - *ecru.*

### 6.2 Explanatory Paragraphs in Itemised Lists Not Possible

It is common practice to insert explanatory paragraphs after items in itemised lists. *This is a grammatical error*: an itemised list comprises only one sentence, and it is not possible to nest other sentences within that sentence.

### 6.3 Itemised List After Numbered Item

In an enumerated list, it is possible to have two or more sentences after each number, if the writer thinks this necessary — and of course any of those sentences may be displayed as an itemised list if appropriate (that is, bullet points may follow a numbered list item).

Some of those following sentences may be displayed with their own ordering symbols (that is, a subsidiary enumerated list may follow a numbered list item). This is common, and presents no conceptual problems.

### 6.4 No Numbered Lists After a Bullet Item

We can place an itemised list after a numbered item in an enumerated list. Can we, conversely, place an enumerated list after a bulleted item in an itemised list? As was the case with the legendary ski resort full of girls looking for husbands and husbands looking for girls, the situation is not as symmetrical as it may at first appear.

Recall that the bulleted items form a single sentence, and each enumerated item contains one or more complete sentences. Sentences cannot reasonably be nested within other sentences, and so an enumerated list *cannot* be placed after a bulleted item without violating grammatical rules.

It is common in technical works to use bulleted items as a variety of unnumbered heading, with a bulleted item, often in bold type, followed by explanatory paragraphs. On reflection, it is hard to think of any reason why you would want to do this — why would you lay out sets of instructions on one or more pages, and imply that the order in which those sets are presented to the reader does not matter? The best solution to this problem is to avoid it: possibly by rewriting the bulleted items as headings followed by text, with any enumerated lists in the text left to stand alone.

## 7 Pathologies

Technical works often contain strange list constructions: enumerated lists disguised as itemised lists, itemised lists disguised as enumerated lists, and hybrid constructions and monsters. These errors often proceed from a failure to understand the basic difference between enumerated and itemised lists.

### 7.1 Enumerated List Disguised as an Itemised List

The most common error appears to be displaying an enumerated list in the guise of an itemised list.

Bad Example 4, taken from a corporate style guide, appears to be an itemised list, but it actually contains four sentences, one of which is not correctly terminated.

Bad Example 4: Enumerated List Disguised as an Itemised List

*Use the following guidelines for creating appendices*:

- *List the appendices in the table of contents.*
- *Refer to the appendices in the preface.*
- *For each appendix, provide an introductory paragraph.*

The list shown in Bad Example 4 should be presented as an enumerated list with an introductory paragraph, as shown in Example 7.

The numbers in Example 7 do not necessarily specify a sequence of actions: they may merely clarify the number of rules to be followed.

Example 7: Improved Version of Bad Example 4

*Use the following guidelines for creating appendices.*

1. *List the appendices in the table of contents.*
2. *Refer to the appendices in the preface.*
3. *For each appendix, provide an introductory paragraph.*

### 7.2 Itemised List Disguised as an Enumerated List

Bad Example 5 appears to be an enumerated list, but the numbers serve no purpose: they do not indicate steps to be taken, or a sequence in which items may be used, or a number of items (how many items are included in "its usual accoutrements"?), or an order of importance.

Bad Example 5: Itemised List Disguised as an Enumerated List

*The items enclosed in the Vampire Protection Kit are as follows.*

1. *An efficient pistol with its usual accoutrements.*
2. *Silver bullets.*
3. *An ivory crucifix.*
4. *Powdered flowers of garlic.*
5. *A wooden stake.*
6. *Professor Blomberg's new serum.*

The information in Bad Example 5 is better presented in an itemised list, as shown in Example 8.

Example 8: Improved Version of Bad Example 5

*The items enclosed in the Vampire Protection Kit are as follows:*

- *an efficient pistol with its usual accoutrements;*
- *silver bullets;*
- *an ivory crucifix;*
- *powdered flowers of garlic;*
- *a wooden stake; and*
- *Professor Blomberg's new serum.*

### 7.3 Sometimes, a Table is a Better Idea

It is possible to put too much material into a list, and sometimes the material would be clearer if it were set out in a table. Bad Example 6, which displays a list with too much material, was taken from a popular work on linguistics [2].

Bad Example 6: An Overloaded List

[The] eight main varieties of speech in China [...]

- *Cantonese* (Yúe) Spoken in the south, mainly Guangdong, southern Guangxi, Macau, Hong Kong. (46 million)
- *Gan* Spoken in Shanxi and south-west Hebei. (21 million)
- *Hakka* Widespread, especially between Fujian and Guangxi. (26 million)
- *Mandarin* A wide range of dialects in the northern, central, and western regions. North Mandarin, as found in Beijing, is the basis of the modern standard language. (720 million)
- *Northern Min* Spoken in north-west Fujian. (10 million)
- *Southern Min* Spoken in the south-east, mainly in parts of Zhejiang, Fujian, Hainan Island, and Taiwan. (26 million)
- *Wu* Spoken in parts of Anhui, Zhejiang, and Jiangsu. (77 million)
- *Xian (Hunan)* Spoken in the south-central region, in Hunan. (36 million)

The material in Bad Example 6 would be better displayed in a three-column table — possibly with itemised lists in some of the table cells.

## 8 Definition Lists

The *definition list*, mentioned in Section 3, is a different kind of list, because the elements of the list are neither itemised nor enumerated. Instead, an item in a definition list comprises two parts: a term and an explanation of the term. These two parts of the item are usually distinguished typographically, and may, but do not necessarily, appear on the same line.

Most word processors do not offer the definition list as an option on their drop-down menus, because a two-column table without a caption, rules, or headers (that is, an *informal table*) does much the same job.

The definition list may be used for setting out glossary items, as shown in Example 9. (This material was derived from [1].)

### 8.1 Uses of the Definition List

Definition lists may be used to set out definitions and construct glossaries. Other possible uses of definition lists are not so obvious: for example, a definition list could be used as a way of setting out otherwise awkward one-bullet lists, which are banned by some style guides.

Example 9: A Definition List

**Klingon** This is perhaps the most fully realised science fiction language. Klingon has a complete grammar and vocabulary, and countless nerds have learned it like high-school French or German.

**Qwghlmian** From Neal Stephenson's *Cryptonomicon* novel and *Baroque Cycle* trilogy, this fictional language is allegedly spoken on obscure British islands. The language has sixteen consonants and no vowels, and is thus ideal for representing binary information — and nearly impossible to pronounce.

**R'lyehian** This other-worldly, barely speakable language is part of the Cthulhu mythos (introduced in the classic Lovecraft short story *The Call of Cthulhu*).

**Sindarin** While Tolkien created several languages for his various *Lord of the Rings* books, Sindarin, the language of the elves, is not only his most beautiful but also his most fully realised invented language.

### 8.2 No Definition List in Common Word Processors

Constructing a definition list with the wraparound layout shown in Example 9 is simple enough in LaTeX, but common word processors do not offer menu items relating to definition lists. Two-column informal tables will usually be a good approximation.

### 8.3 History of the Definition List

The definition list is defined in HTML, in the DITA (Darwin Information Typing Architecture: an XML data model for authoring) specification of 2005, in the DocBook specification of 1990, and in the LaTeX specification of 1986; it appears to have originated in the now almost-forgotten Scribe text formatter, ca. 1978. The aim of the author of Scribe was to provide a simple way of coding command descriptions in programming manuals.

Scribe could not handle tables, but LaTeX could, to a limited extent. The only tables that LaTeX could handle at first were less than one page in length, and those tables had to be *floats*, where the position of the table in the final print version of the document was determined by an algorithm that positioned the table so that it did not extend over a page break. While that one-page limitation existed, the definition list was still useful, because it could be used to lay out tabular material that occupied more than one page.

## 9 Conclusions

Applying a few simple rules to itemised and enumerated lists leads to some unexpected conclusions. In particular, itemised lists are much more limited in scope than they at first appear. In many instances, enumerated lists are much more useful.

Definition lists are not used by technical writers as much as they could be.

## References

[1] John Baichtel. Top ten geekiest constructed languages. 2009. `http://www.wired.com/geekdad/2009/08/top-ten-geekiest-constructe-languages/`.

[2] David Crystal. *How Language Works.* Avery, New York, 2007.

[3] Microsoft Corporation. *Microsoft Manual of Style for Technical Publications, Third Edition.* Microsoft Press, Sebastopol, Calif., 2004.

◇ James R. Hunt
P. O. Box 580
Mt Gravatt
Queensland 4122
Australia
`writerlyjames (at) gmail dot com`

## *In memoriam* Jean-Pierre Drucbert

Jean-Michel Hufflen

Jean-Pierre Drucbert passed away in March 2009 . . . quietly . . . as he lived . . . He was discreet yet efficient. . . in his own way. . . I personally worked in the same building[1] as him, for two years, from 1989 to 1991. At that time, I got by using LaTeX, but had only a little experience, compared with my present ability. So several times I hesitated to start in implementing some functions and asked him for some advice. At the time, he told me: 'That's not obvious.' Or: 'That's difficult.' But a little while after that, he waved discreetly to me to join him. . . and he implemented the commands.

I had met him after hearing about his translation of Leslie Lamport's original LaTeX book, incorporating much additional information. As far as I know, this work[2] was one of the first complete LaTeX manuals in French, if not THE first. It has only been distributed privately; I personally thought that was a pity. Several times I suggested to Jean-Pierre that he should submit it to a publisher. But he was not interested in the limelight, it was not in his character. He was solitary, as if he bore some secret and deep-rooted pain, some dead-end despair. But he was always ready to help other people within his activities' scope. He left many packages[3] as a remarkable testimony of that.

◇ Jean-Michel Hufflen
FEMTO-ST & University of Franche-Comté,
16 route de Gray,
25030 Besançon Cedex,
France

[1] At CERT ('***C**entre d'**É**tudes et de **R**echerches de **T**oulouse*', that is, 'Toulouse study and research centre').

[2] Jean-Pierre DRUCBERT : *Utilisation de LaTeX et BibTeX sur Multics au CERT*. August 1988. CERT, IT-service group.

[3] See `http://ctan.org/author/drucbert` for a complete list including 'original' implementations and adaptations to French of existing ones.

# Letters

## Does not suffice to run `latex` a finite number of times to get cross-references right

Jaime Gaspar

**Abstract.** We present a LaTeX file such that a cross-reference is wrong no matter how many times we run `latex`.

— — ∗ — —

It is well-known that we need to run `latex` several times to get cross-references right. This raises a natural question for mathematicians: for any LaTeX file, does it suffice to run `latex` a finite number of times? We show that the answer is negative, by a counterexample. The LaTeX file

```
\documentclass{article}
\usepackage{forloop}
\begin{document}
 \newcounter{n}
 \forloop{n}{0}
          {\value{n} < \pageref{l}}{~\newpage}
 Last-page label here\label{l}.
 Label value: \pageref{l}.
\end{document}
```

is such that the cross-reference `\pageref{l}` is wrong no matter how many times we run `latex`. This file uses a little diabolical trick: a label `l` is created in the last page (line 7) and there are created (resorting to a for loop) `\pageref{l}` many new pages (lines 5 and 6), causing the document to have `\pageref{l}` $+ 1$ pages, so the cross-reference `\pageref{l}` to the last page is wrong. (An even more diabolical counterexample that avoids a for loop is shown at `http://tex.stackexchange.com/questions/30674`.)

**Acknowledgement.** At the time of writing: INRIA Paris-Rocquencourt, $\pi r^2$, Univ Paris Diderot, Sorbonne Paris Cité, F-78153 Le Chesnay, France; financially supported by the French Fondation Sciences Mathématiques de Paris.

◇ Jaime Gaspar
Universitat Rovira i Virgili
Department of Computer Engineering and Mathematics
Av. Països Catalans 26
E-43007 Tarragona, Catalonia
`jaime.gaspar (at) urv dot cat`;
Centro de Matemática e Aplicações (CMA), FCT, UNL

## Fetamont: An extended logo typeface

Linus Romer

### Abstract

The logo font, known from logos like METAFONT or METAPOST, has been very limited in its collection of glyphs. The new typeface *Fetamont* extends the logo typeface in two ways:

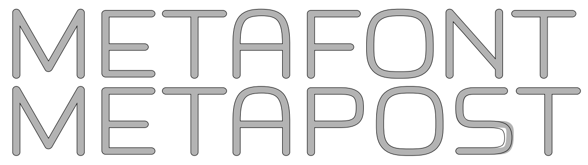
- Fetamont consists of 256 glyphs, such that the T1 (a.k.a. EC, a.k.a. Cork) encoding table is complete now.
- Fetamont has additional faces like "light ultra-condensed" or "script".

The `fetamont` package provides LaTeX support for the Fetamont typeface. Both the package and the typeface are distributed on CTAN under the terms of the *LaTeX Project Public License* (LPPL).

The following article presents some facets of the Fetamont typeface, explains important techniques and shows the history of the logo typeface.

## 1 Comparison with existing logos

The following picture shows the METAPOST and the METAFONT logos written in Fetamont (gray) and Taco Hoekwater's Type 1 version of the logo font (outlined).



There are hardly any differences; only the "S" is significantly different, because its shape was changed by D. E. Knuth in 1997 (see section 6). The other faces of Hoekwater's *Logo* are also very similar to their corresponding Fetamont faces. Widths and kernings may rarely differ by one unit (except for the "A" in *Logo 9*, which has a strange width).

A comparison with the METATYPE1 logo from Jackowski, Nowacki, and Strzelczyk (2001) shows virtually no differences as well.[1]

## 2 Comparison with the `mflogo` package

The control sequences defined in the `fetamont` package are analogous to those from the `mflogo` package (Vieth, 1999). `\MF`, `\MP` and `\MT` produce the well-known logos of METAFONT, METAPOST and METATYPE1.

[1] I have never seen the original sources of the "Y" and the "1" but I think that my imitated "Y" and "1" are extremely close to the original.

## 3 The many faces of fetamont

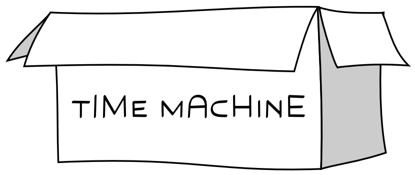
It is clear that thanks to the power of METAFONT the number of possible faces is endless. However, Fetamont comes in a mere 36 predefined faces. The suffixes of every face are schematically listed in the following table:

| Upright | | | | Oblique | | | |
|---|---|---|---|---|---|---|---|
| | r8 | b8 | h8 | | o8 | bo8 | ho8 |
| | r9 | b9 | h9 | | o9 | bo9 | ho9 |
| l10 | r10 | b10 | h10 | lo10 | o10 | bo10 | ho10 |
| Condensed Upright | | | | Condensed Oblique | | | |
| lc10 | c10 | | | lco10 | co10 | | |
| | | bc40 | | | | bco40 | |
| Ultracond. Upright | | | | Ultracond. Oblique | | | |
| lq10 | | | | lqo10 | | | |
| Script Upright | | | | Script Oblique | | | |
| lw10 | w10 | bw10 | hw10 | lwo10 | wo10 | bwo10 | hwo10 |

Anyone wishing to design a new face for Fetamont can do so by just redefining the parameters of `ffmr10.mf` and saving the file under a new name.

### 3.1 Script faces

The Fetamont script faces make use of randomized paths that are drawn by a rotated ellipse pen to make it look more handwritten. They may be used for comics or children's texts:



Since version 1.3, the OpenType versions of the script faces even support the "Randomize" feature.

### 3.2 Condensed faces

The titles in Knuth's books show a variant of the logo font that blends with *Computer Modern Sans Serif Demibold Condensed 40*. So I decided to add this variant as *Fetamont Bold Condensed 40* and let also a light and medium variant benefit from the condensation because it looked so good.

Light Condensed 10

Medium Condensed 10

**Bold Condensed 40**

### 3.3 Ultracondensed Face

I went even a step further and created an ultracondensed face for Fetamont. The credits written on movie posters are often typeset in such ultracondensed faces:

THE MOST IMPORTANT THING IN THE PROGRAMMING LANGUAGE IS THE NAME. A LANGUAGE WILL NOT SUCCEED WITHOUT A GOOD NAME. I HAVE RECENTLY INVENTED A VERY GOOD NAME AND NOW I AM LOOKING FOR A SUITABLE LANGUAGE.

(This is said to be a quotation from D. E. Knuth.)

## 4 Spacing problems with the "S"

The original spacing of the letter "S" fits perfectly for the combination "OST" as in METAPOST. However, in combination with normal letters like "N", the "S" is positioned too much to the right (see the left part of the following picture).

NSN NSN

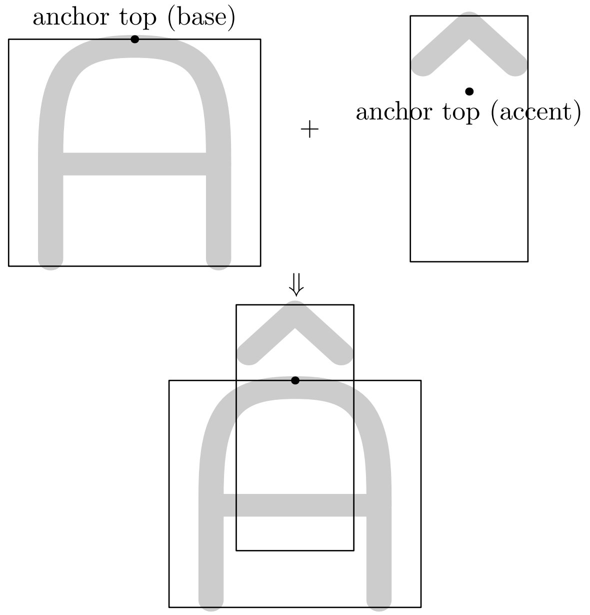
Thus, I shifted the "S" a bit to the left (see the right part of the upper picture) and added corresponding kerning instructions for "OS" and "ST" to keep the spacing of the original logo intact.

## 5 Special techniques

### 5.1 Anchor pairing with METAFONT

In order to draw accented and other combined characters, it is helpful to use *anchors*. The concept of anchors is common in type design outside of the METAFONT world. However, anchors rarely have been seen in METAFONT up to now.

The idea is easy: Put an anchor at a given point in a base glyph and in the accent glyph; then overlay the two glyphs such that the anchors coincide, producing the pre-composed accented character.



Normally several kinds of anchors are needed. E.g. "Ć" and "Ç" need two different anchors. Fetamont stores anchors as ordinary points in arrays. Furthermore, the character picture is stored in another array at the end of each character. For combined characters, METAFONT looks at the different arrays and then overlays the base and the accent character using the macro `addto currentpicture`.

### 5.2 Kerning classes with METAFONT

Like anchor pairing, the concept of kerning classes is widely known but not frequently used in METAFONT. The reason for this is that METAFONT cannot natively write kernings for multiple characters at once. Hence, multiple kerning information has to be cached in arrays.

Let me illustrate the general idea of this caching with a fictional example:

#### 5.2.1 Define kerning classes

It is clear that "OV" needs the same kerning as "DV". But beware, "VO" needs a different kerning than "VD"! So generally there are two kinds of kerning classes:

- The *first kerning class* groups glyphs together that share the same shape to the right, like "D" and "O"
- The *second kerning class* groups glyphs together that share the same shape to the left, like "C" and "O"

The first kerning class is stored in an array as follows:

| | | | | |
|---|---|---|---|---|
| row 0 | 2 | | | |
| row 1 | 3 | 68 (D) | 79 (O) | 214 (Ö) |
| row 2 | 2 | 86 (V) | 87 (W) | |

The 0th column is reserved for the number of columns (0th row) and the number of items in the corresponding row, because there is no straightforward way to determine the length of arrays or subarrays in METAFONT. Each row forms a first kerning class. The characters are stored as codes (for the sake of clarity, they are shown here as letters also).

The storage of the second kerning classes works analogously, in another array:

| | | | | | |
|---|---|---|---|---|---|
| row 0 | 2 | | | | |
| row 1 | 4 | 67 (C) | 79 (O) | 80 (Q) | 214 (Ö) |
| row 2 | 2 | 86 (V) | 87 (W) | | |

#### 5.2.2 Kern the kerning classes

The classes are then kerned with the following commands:

```
addclasskern("D","C",2u#)
addclasskern("D","V",-u#)
addclasskern("V","C",-u#)
```

This kerning information is stored in a large three-dimensional array, which has as many columns as characters:

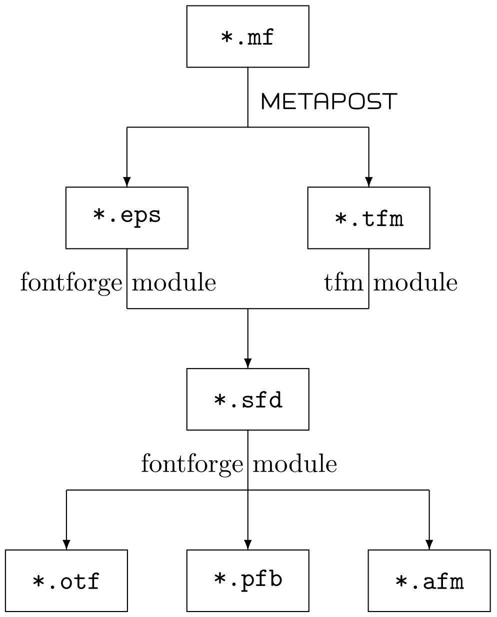
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| row 68 | 6 | $\binom{67}{2u\#}$ | $\binom{79}{2u\#}$ | $\binom{80}{2u\#}$ | $\binom{214}{2u\#}$ | $\binom{86}{-u\#}$ | $\binom{87}{-u\#}$ |
| row 79 | 6 | $\binom{67}{2u\#}$ | $\binom{79}{2u\#}$ | $\binom{80}{2u\#}$ | $\binom{214}{2u\#}$ | $\binom{86}{-u\#}$ | $\binom{87}{-u\#}$ |
| row 86 | 4 | $\binom{67}{-u\#}$ | $\binom{79}{-u\#}$ | $\binom{80}{-u\#}$ | $\binom{214}{-u\#}$ | | |
| row 87 | 4 | $\binom{67}{-u\#}$ | $\binom{79}{-u\#}$ | $\binom{80}{-u\#}$ | $\binom{214}{-u\#}$ | | |
| row 214 | 6 | $\binom{67}{2u\#}$ | $\binom{79}{2u\#}$ | $\binom{80}{2u\#}$ | $\binom{214}{2u\#}$ | $\binom{86}{-u\#}$ | $\binom{87}{-u\#}$ |

#### 5.2.3 Write the kerning information

At the very end, the macro `writeligtable` writes all kerning information from the large three-dimensional array, row-wise, in a METAFONT-friendly way.

### 5.3 Producing outlines

The METAFONT sources have been converted to outline font formats like Type 1 or OpenType with a Python script. The script calls METAPOST to produce PostScript files for each glyph. These glyphs are imported by the `fontforge` module. Hosny (2011) previously used this technique to produce the outlines of *Punk Nova*. Because the glyph widths are lost in the import, the `tfm` module from the `mftrace` project is also needed (Nienhuys, 2006).



## 6 History of the logo typeface

- 1979/07/12: Knuth (1979a) shows the first versions of the "Computer Modern" typeface and the METAFONT logo. The graphic below is the result of a conversion from the original "manfnt" sources (Knuth, 1979b), which were written in the obsolete METAFONT78.

  **METAFONT**

  Knuth used quite a thick circular pen. This "manfnt" also contains 8 pt, 9 pt and title versions of the logo typeface. However, the title font is just a magnified version of the 10 pt font.
- 1984/05/27: The sources of the logo font are rewritten for the new METAFONT84 (Knuth, 1984a). The pen has become elliptic and thinner.
- 1984/09/09: Second try for the new METAFONT84 (Knuth, 1984b). The characters "E" and "F" have rounded vertices.

  METAFONT
- 1985/09/03: Knuth (1985e) defines some "crazy shapes". He then (Knuth, 1986) uses them to demonstrate randomized typefaces.

  METAFONT METAFONT
- 1985/09/22: The logo typeface gets a slanted variant (Knuth, 1985b), a backslanted skinny bold variant (Knuth, 1985f) and an ultrawide light variant (Knuth, 1985d).

  META FONT

  The shapes of the letters A, E, F, M, N, O, T have reached their final state (Knuth, 1985c and Knuth, 1985a).

  METAFONT
- 1985/10/06: Knuth (1985g) uses a logo variant for titles along with *Computer Modern Sans Serif Demibold Condensed 40* for the title pages in Knuth (1986).

  **METAFONT**
- 1986/01/07: Knuth (1986) specifies a boldface variant of the METAFONT logo. This variant goes well with *Computer Modern Sans Serif Bold*:

  **SansMETA**
- 1989/04/22: Knuth (1989) shows a new special weight to go with Pandora (see Billawala, 1989).

  Pandora METAFONT
- 1989/06/24: Cugley (1989) extends the logo font to cover the whole uppercase alphabet and a couple of punctuation characters in the so-called "mf" font.

  DAMIAN CUGLEY

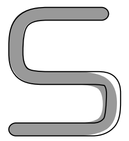
- 1992/01/16: Knuth (1992a) adds a demibold variant. This variant is needed for the title "Excerpts from the Programs for TEX and METAFONT" (Knuth, 1992b), which combines *Computer Modern Bold Extended* with the logo font.

**TEX and METAFONT**

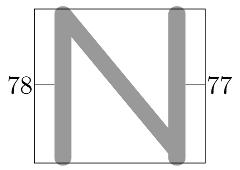
- 1993/03/23: Knuth (1993a) adds the characters "P" and "S" to the logo font such that one can typeset "METAPOST" with it. Hobby (2009) states that both of the following logos are okay:

METAPOST MetaPost

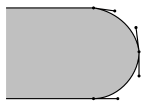
- 1997/04/20: Knuth (1993b) changes the shape of the "S". "It now sort of assumes that a 'T' will follow" (Beeton, 1998).



- 1997/09/30: The American Mathematical Society provides Type 1 versions of the logo font in the following styles: `logo8`, `logo9`, `logo10`, `logobf10`, `logosl10` (AMS, 2009). The "P" and the "S" are not included and the widths of the supplied glyphs are often wrongly rounded.



- 1999/6/03: Taco Hoekwater releases new Type 1 versions of the logo font (Hoekwater, 1999). The conversion has been done in MetaFog, which is only available with TrueTEX. The "S" still has its old shape. The round endings are sometimes suboptimal (e.g. the "F" from *Logo 10*) but all in all, the conversion is very clean.



- 2001/03/04: Taco Hoekwater presents *Elogo*, an extended version of the logo font, which covers the 7-bit TEX text encoding (Hoekwater, 2001). The font has been used in some of Hoekwater's presentation slides (e.g. the following image is extracted from Hoekwater, 2007). The "S" still has the old shape.

DYNAMIC

- 2001/09/26: At EuroTEX 2001 the METATYPE1 logo is shown in the presentation of Jackowski, Nowacki, and Strzelczyk (2001). The newly designed characters "Y" and "1" are different to the "Y" and "1" in *Elogo*.

METATYPE1

Funnily, the EuroTEX 2001 conference logo also embeds an extended logo font (Pepping, 2001).

EUROTEX 2001

- 2011/06/02: For the documentation of another typeface, I wanted to write "mf2pt1" in the logo font, because I was heavily relying on this program back then. Having realised that there were other tools related to METAFONT that could not be written in the logo font, I started extending the logo font.

MF2PT1 MFLua MetaFog

**References**

AMS. "Computer Modern PostScript Fonts". `www.ams.org/amsfonts`, 2009.

Beeton, Barbara. "Editorial comments". *TUGboat* **19**(4), 1998. `tug.org/TUGboat/tb19-4/tb61beet.pdf`.

Billawala, Nazneen N. "Metamarks: Preliminary Studies for a Pandora's Box of Shapes". Technical Report STAN-CS 1256, Stanford University, 1989.

Cugley, Damian. "A character font more than a little reminiscient of the METAFONT logo font by D. E. Knuth". `mirror.ctan.org/fonts/utilities/mff-29/mf.mf`, 1989.

Hobby, John. "The METAPOST page". `ect.bell-labs.com/who/hobby/MetaPost.html`, 2009.

Hoekwater, Taco. `mirror.ctan.org/fonts/mflogo/ps-type1/hoekwater`, 1999.

Hoekwater, Taco. "A new metafont (beta test)". `comments.gmane.org/gmane.comp.tex.context/4259`, 2001.

Hoekwater, Taco. "METAPOST Developments". `www.luatex.org/talks/tug2007-taco-metapost.pdf`, 2007.

Hosny, Khaled. `https://github.com/khaledhosny/punk-otf/blob/master/tools/build.py`, 2011.

Jackowski, Bogusław, J. M. Nowacki, and P. Strzelczyk. "METATYPE1: A METAPOST-based engine for generating Type 1 fonts". `www.ntg.nl/eurotex/JackowskiMT.pdf`, 2001.

Knuth, Donald E. *TEX and METAFONT: New directions in typesetting.* American Mathematical Society, 1979a.

Knuth, Donald E. "Special font for the METAFONT manual". www.saildart.org/MANFNT.MF[MF,DEK]1, 1979b.

Knuth, Donald E. "Letters for the METAFONT logo; first try with the new MF". www.saildart.org/LOGO.MF[FNT,DEK]1, 1984a.

Knuth, Donald E. "Letters for the METAFONT logo; second try with the new MF". www.saildart.org/LOGO.MF[FNT,DEK], 1984b.

Knuth, Donald E. www.saildart.org/METAFO.MF[MF,SYS], 1985a.

Knuth, Donald E. "10-point slanted METAFONT logo". www.saildart.org/LOGL10.MF[MF,SYS]1, 1985b.

Knuth, Donald E. "Driver file for the METAFONT logo". www.saildart.org/NLOGO.MF[MF,SYS], 1985c.

Knuth, Donald E. "Fat version of METAFONT logo". www.saildart.org/FLOGO.MF[MF,SYS], 1985d.

Knuth, Donald E. "Font for examples in Chapter 21 of *The METAFONTbook*". www.saildart.org/RANDOM.MF[FNT,DEK]1, 1985e.

Knuth, Donald E. "Skinny variant of METAFONT logo". www.saildart.org/SKLOGO.MF[MF,SYS], 1985f.

Knuth, Donald E. "Special font for the TEX and METAFONT manuals". www.saildart.org/LOGO.MF[MF,SYS]1, 1985g.

Knuth, Donald E. *The METAFONTbook.* Addison-Wesley, 1986.

Knuth, Donald E. "10-point METAFONT logo, special weight to go with Pandora text". www.saildart.org/LOGN10.MF[NB,DEK], 1989.

Knuth, Donald E. "10-point demibold METAFONT logo". mirror.ctan.org/systems/knuth/local/lib/logod10.mf, 1992a.

Knuth, Donald E. *Literate Programming.* CSLI, 1992b.

Knuth, Donald E. "Routines for the METAFONT logo, as found in *The METAFONTbook*". ftp://cs.stanford.edu/pub/concretemath.errata/fonts/logo.mf, 1993a.

Knuth, Donald E. "Routines for the METAFONT logo, as found in *The METAFONTbook*". ftp://cs.stanford.edu/pub/tex/dist/lib/logo.mf, 1993b.

Nienhuys, Han-Wen. https://github.com/hanwen/mftrace/blob/master/tfm.py, 2006.

Pepping, Simon, editor. *EuroTEX 2001 proceedings.* NTG, 2001.

Vieth, Ulrik. "The mflogo package". mirror.ctan.org/macros/latex/contrib/mflogo/mflogo.pdf, 1999.

◇ Linus Romer
Oberseestrasse 7
Schmerikon, 8716
Switzerland
linus.romer (at) gmx dot ch

# LaTeX3 News

Issue 9, March 2014

## Contents

## Hiatus?

Well, it's been a busy couple of years. Work has slowed on the LaTeX3 codebase as all active members of the team have been — shall we say — busily occupied with more pressing concerns in their day-to-day activities.

Nonetheless, Joseph and Bruno have continued to fine-tune the LaTeX3 kernel and add-on packages. Browsing through the commit history shows bug fixes and improvements to documentation, test files, and internal code across the entire breadth of the codebase.

Members of the team have presented at two TUG conferences since the last LaTeX3 news. (Has it really been so long?) In July 2012, Frank and Will travelled to Boston; Frank discussed the challenges faced in the past and continuing to the present day due to the limits of the various TeX engines; and, Frank and Will together covered a brief history and recent developments of the expl3 code.

In 2013, Joseph and Frank wrote a talk on complex layouts, and the "layers" ideas discussed in LaTeX3; Frank went to Tokyo in October to present the work. Slides of and recordings from these talks are available on the LaTeX3 website.

These conferences are good opportunities to introduce the expl3 language to a wider group of people; in many cases, explaining the rationale behind why expl3 looks a little strange at first helps to convince the audience that it's not so weird after all. In our experience, anyone that's been exposed to some of the more awkward expansion aspects of TeX programming appreciates how expl3 makes life much easier for us.

## expl3 in the community

While things have been slightly quieter for the team, more and more people are adopting expl3 for their own use. A search on the TeX Stack Exchange website for either "`expl3`" or "`latex3`" at time of writing yield around one thousand results each.

In order to help standardise the prefixes used in expl3 modules, we have developed a registration procedure for package authors (which amounts to little more than notifying us that their package uses a specific prefix, which will often be the name of the package itself). Please contact us via the `latex-l` mailing list to register your module prefixes and package names; we ask that you avoid using package names that begin with `l3...` since expl3 packages use this internally. Some authors have started using the package prefix `lt3...` as a way of indicating their package builds on expl3 in some way but is not maintained by the LaTeX3 team.

In the prefix database at present, some thirty package prefixes are registered by fifteen separate individuals (unrelated to the LaTeX3 project — the number of course grows if you include packages by members of the team). These packages cover a broad range of functionality:

**acro** Interface for creating (classes of) acronyms

**hobby** Hobby's algorithm in PGF/TiKZ for drawing optimally smooth curves.

**chemmacros** Typesetting in the field of chemistry.

**classics** Traditional-style citations for the classics.

**conteq** Continued (in)equalities in mathematics.

**ctex** A collection of macro packages and document classes for Chinese typesetting.

**endiagram** Draw potential energy curve diagrams.

**enotez** Support for end-notes.

**exsheets** Question sheets and exams with metadata.

**lt3graph** A graph data structure.

**newlfm** The venerable class for memos and letters.

**fnpct** Interaction between footnotes and punctuation.

**GS1** Barcodes and so forth.

**hobete** Beamer theme for the Univ. of Hohenheim.

**kantlipsum** Generate sentences in Kant's style.

**lualatex-math** Extended support for mathematics in LuaLaTeX.

**media9** Multimedia inclusion for Adobe Reader.

**pkgloader** Managing the options and loading order of other packages.

**substances** Lists of chemicals, etc., in a document.

**withargs** Ephemeral macro use.

**xecjk** Support for CJK documents in XeLaTeX.

**xpatch, regexpatch** Patch command definitions.

**xpeek** Commands that peek ahead in the input stream.

**xpinjin** Automatically add pinyin to Chinese characters

**zhnumber** Typeset Chinese representations of numbers

**zxjatype** Standards-conforming typesetting of Japanese for XeLaTeX.

Some of these packages are marked by their authors as experimental, but it is clear that these packages have been developed to solve specific needs for typesetting and document production.

The expl3 language has well and truly gained traction after many years of waiting patiently.

## *A logo for the LaTeX3 Programming Language*

To show that expl3 is ready for general use Paulo Cereda drew up a nice logo for us, showing a hummingbird (agile and fast — but needs huge amounts of energy) picking at "l3". Big thanks to Paulo!

## *Recent activity*

LaTeX3 work has only slowed, not ground to a halt. While changes have tended to be minor in recent times, there are a number of improvements worth discussing explicitly.

1. Bruno has extended the floating point code to cover additional functions such as inverse trigonometric functions. These additions round out the functionality well and make it viable for use in most cases needing floating point mathematics.

2. Joseph's refinement of the experimental galley code now allows separation of paragraph shapes from margins/cutouts. This still needs some testing!

3. For some time now expl3 has provided "native" drivers although they have not been selected by default in most cases. These have been revised to improve robustness, which makes them probably ready to enable by default. The improvements made to the drivers have also fed back to more "general" LaTeX code.

## *Work in progress*

We're still actively discussing a variety of areas to tackle next. We are aware of various "odds and ends" in expl3 that still need sorting out. In particular, some experimental functions have been working quite well and it's time to assess moving them into the "stable" modules, in particular the l3str module for dealing with catcode-twelve token lists more commonly known in expl3 as *strings*.

Areas of active discussion including issues around uppercasing and lowercasing (and the esoteric ways that this can be achieved in TeX) and space skipping (or not) in commands and environments with optional arguments. These two issues are discussed next.

### *Uppercasing and lowercasing*

The commands `\tl_to_lowercase:n` and `\tl_to_uppercase:n` have long been overdue for a good hard look. From a traditional TeX viewpoint, these commands are simply the primitive `\lowercase` and `\uppercase`, and in practice it's well known that there are various limitations and peculiarities associated with them. We know these commands are good, to one extent or another, for three things:

1. Uppercasing text for typesetting purposes such as all-uppercase titles.

2. Lowercasing text for normalisation in sorting and other applications such as filename comparisons.

3. Achieving special effects, in concert with manipulating `\uccode` and the like, such as defining commands that contain characters with different catcodes than usual.

We are working on providing a set of commands to achieve all three of these functions in a more direct and easy-to-use fashion, including support for Unicode in LuaLaTeX and XeLaTeX.

### *Space-skipping in xparse*

We have also re-considered the behaviour of space-skipping in xparse. Consider the following examples:

```
\begin{dmath}         \begin{dmath}[label=foo]
[x y z] = [1 2 3]     x^2 + y^2 = z^2
\end{dmath}           \end{dmath}
```

In the first case, we are typesetting some mathematics that contains square brackets. In the second, we are assigning a label to the equation using an optional argument, which also uses brackets. The fact that both work correctly is due to behaviour that is specifically programmed into the workings of the `dmath` environment of breqn: spaces before an optional argument are explicitly forbidden. At present, this is also how commands and environments defined using xparse behave. But consider a pgfplots environment:

```
\begin{pgfplot}
  [
    % plot options
  ]
  \begin{axis}
    [
      % axis options
    ]
    ...
  \end{axis}
\end{pgfplot}
```

This would seem like quite a natural way to write such environments, but with the current state of xparse this syntax would be incorrect. One would have to write either of these instead:

```
\begin{pgfplot}%          \begin{pgfplot}[
  [                          % plot options
   % plot options           ]
  ]
```

Is this an acceptable compromise? We're not entirely sure here — we're in a corner because the humble `[` has ended up being part of both the syntax and semantics of a LaTeX document.

Despite the current design covering most regular use-cases, we have considered adding a further option to xparse to define the space-skipping behaviour as desired by a package author. But at this very moment we've rejected adding this additional complexity, because environments that change their parsing behaviour based on their intended use make a LaTeX-based language more difficult to predict; one could imagine such behaviour causing difficulties down the road for automatic syntax checkers and so forth. However, we don't make such decisions in a vacuum and we're always happy to continue to discuss such matters.

## *. . . and for 2014 onwards*

There is one (understandable) misconception that shows up once in a while with people claiming that

$$\text{expl3} = \text{LaTeX3}.$$

However, the correct relation would be a subset,

$$\text{expl3} \subset \text{LaTeX3},$$

with expl3 forming the Core Language Layer on which there will eventually be several other layers on top that provide

- higher-level concepts for typesetting (Typesetting Foundation Layer),
- a designer interface for specifying document structures and layouts (Designer Layer),
- and finally a Document Representation Layer that implements document level syntax.

Of those four layers, the lowest one — expl3 — is available for use and with xparse we have an instance of the Document Representation Layer modeled largely after LaTeX 2$_\varepsilon$ syntax (there could be others). Both can be successfully used within the current LaTeX 2$_\varepsilon$ framework and as mentioned above this is increasingly happening.

The middle layers, however, where the rubber meets the road, are still at the level of prototypes and ideas (templates, ldb, galley, xor and all the good stuff) that need to be revised and further developed to arrive at a LaTeX3 environment that can stand on its own and that is to where we want to return in 2014.

An overview on this can be found in the answer to "What can *I* do to help the LaTeX3 project?" on Stack Exchange,[1] which is reproduced below in slightly abridged form. This is of course not the first time that we have discussed such matters, and you can find similar material in other publications such as those at `http://latex-project.org`; e.g., the architecture talk given at the TUG 2011 conference.

[1] `http://tex.stackexchange.com/questions/45838`

## *What can you do for the LaTeX3 project?*

**By Frank Mittelbach**

My vision of LaTeX3 is really a system with multiple layers that provide interfaces for different kinds of roles. These layers are

- the underlying engine (some TeX variant)
- the programming layer (the core language, i.e., expl3)
- the typesetting foundation layer (providing higher-level concepts for typesetting)
- the typesetting element layer (templates for all types of document elements)
- the designer interface foundation layer
- the class designer layer (where instances of document elements with specific settings are defined)
- document representation layer (that provides the input syntax, i.e., how the author uses elements)

If you look at it from the perspective of user roles then there are at least three or four roles that you can clearly distinguish:

- The Programmer (template and functionality provider)
- The Document Type Designer (defines which elements are available; abstract syntax and semantics)
- The Designer (typography and layout)
- The Author (content)

As a consequence the LaTeX3 Project needs different kinds of help depending on what layer or role we are looking at.

The "Author" is using, say, list structures by specifying something like `\begin{itemize} \item` in his documents. Or perhaps by writing `<ul> ... </ul>` or whatever the UI representation offers to him.

The "Document Type Designer" defines what kind of abstract document elements are available, and what attributes or arguments those elements provide at the author level. E.g., he may specify that a certain class of documents provides the display lists "enumerate", "itemize" and "description".

The "Programmer" on the other hand implements templates (that offer customizations) for such document elements, e.g., for display lists. What kind of customization possibilities should be provided by the "Programmer" is the domain of the "Document Designer"; he drives what kind of flexibility he needs for the design. In most cases the "Document Designer" should be able to simply select templates (already written) from a template library and only focus on the design, i.e., instantiating the templates with values so that the desired layout for "itemize" lists, etc., is created.

In real life a single person may end up playing more than one role, but it is important to recognise that all of them come with different requirements with respect to interfaces and functionality.

### *Programming Layer*

The programming layer consists of a core language layer (called expl3 (EXP erimental L aTeX 3) for historical reasons and now we are stuck with it `:-)`) and two more components: the "Typesetting Foundation Layer" that we are currently working on and the "Typesetting Element Layer" that is going to provide customizable objects for the design layer. While expl3 is in many parts already fairly complete and usable the other two are under construction.

Help is needed for the programming layer in

- helping by extending and completing the regression test suite for expl3
- helping with providing good or better documentation, including tutorials
- possibly helping in coding additional core functionality — but that requires, in contrast to the first two points, a good amount of commitment and experience with the core language as otherwise the danger is too high that the final results will end up being inconsistent

Once we are a bit further along with the "Typesetting Foundation Layer" we would need help in providing higher-level functionality, perhaps rewriting existing packages/code for elements making use of extended possibilities. Two steps down the road (once the "Designer Layer" is closer to being finalized) we would need help with developing templates for all kinds of elements.

In summary for this part, we need help from people interested in programming in TeX and expl3 and/or interested in providing documentation (but for this a thorough understanding of the programming concepts is necessary too).

### *Design Layer*

The intention of the design layer is to provide interfaces that allow specifying layout and typography styles in a declarative way. On the implementation side there are a number of prototypes (most notably xtemplate and the recent reimplementation of ldb). These need to be unified into a common model which requires some more experimentation and probably also some further thoughts.

But the real importance of this layer is not the implementation of its interfaces but the conceptual view of it: provisioning a rich declarative method (or methods) to describe design and layout. I.e., enabling a designer to think not in programs but in visual representations and relationships.

So here is the big area where people who do not feel they can or want to program TEX's bowels can help. What would be extremely helpful (and in fact not just for LATEX3) would be

- collecting and classifying a *huge* set of layouts and designs
  - designs for individual document elements (such as headings, TOCs, etc)
  - document designs that include relationships between document elements
- thinking about good, declarative ways to specify such designs
  - what needs to be specified
  - to what extent and with what flexibility

I believe that this is a huge task (but rewarding in itself) and already the first part of collecting existing design specifications will be a major undertaking and will need coordination and probably a lot of work. But it will be a huge asset towards testing any implementations and interfaces for this layer later on.

### *Document Interface Layer*

If we get the separation done correctly, then this layer should effectively offer nothing more than a front end for parsing the document syntax and transforming it into an internal standardised form. This means that on this layer one should not see any (or not much) coding or computation.

It is envisioned that alternative document syntax models can be provided. At the moment we have a draft solution in xparse. This package offers a document syntax in the style of LATEX 2$\varepsilon$, that is with *-forms, optional arguments in brackets, etc., but with a few more bells and whistles such as a more generalized concept of default values, support for additional delimiters for arguments, verbatim-style arguments, and so on. It is fairly conventional though. In addition when it was written the clear separation of layers wasn't well-defined and so the package also contains components for conditional programming that I no longer think should be there.

Bottom line on what is needed for this layer is to

- think about good syntax for providing document content from "the author" perspective
- think about good syntax for providing document content from an "application to typesetting" perspective, i.e., the syntax and structure for automated typesetting where the content is prepared by a system/application rather than by a human

The two most likely need strict structure (as automation works much better with structures that do not have a lot of alternative possibilities and shortcuts, etc.) and even when just looking at the human author a lot of open questions need answering. And these answers may or may not be to painfully stick with existing LATEX 2$\varepsilon$ conventions in all cases (or perhaps with any?).

None of this requires coding or expl3 experience. What it requires is familiarity with existing input concepts, a feel for where the pain points currently are and the willingness to think and discuss what alternatives and extensions could look like.

### *In Summary*

Basically help is possible on any level and it doesn't need to involve programming. Thoughts are sprinkled throughout this article, but here are a few more highlights:

- help with developing/improving the core programming layer by
  - joining the effort to improve the test suite
  - help improving the existing (or not existing) documentation
  - joining the effort to produce core or auxiliary code modules
- help on the design layer by
  - collecting and classifying design tasks
  - thinking and suggesting ways to describe layout requirements in a declarative manner
- help on shaping the document interface layer

These concepts, as well as their implementation, are under discussion on the list `latex-l`.[2] The list has only a fairly low level of traffic right now as actual implementation and development tasks are typically discussed directly among the few active implementors. But this might change if more people join.

### *And something else . . .*

The people on the LATEX3 team are also committed to keeping LATEX 2$\varepsilon$ stable and even while there isn't that much to do these days there remains the need to resolve bug reports (if they concern the 2e core), provide new distributions once in a while, etc. All this is work that takes effort or remains undone or incomplete. Thus here too, it helps the LATEX3 efforts if we get help to free up resources.

[2]Instructions for joining and browsing archives at: `http://latex-project.org/code.html`

# Introduction to presentations with beamer

Thomas Thurnherr

## Abstract

The document class beamer provides flexible commands to prepare presentations with an appealing look. Here I introduce the basics of the beamer class intended for new users with a basic knowledge of LaTeX. I cover a range of topics from how to create a first slide to dynamic content and citations.

## 1 Introduction

The LaTeX document class beamer [1] was written to help with the creation of presentations held using a projector. In German, the English word *Beamer* describes a projector, which is likely the reason for Till Tantau, the package author, to choose this particular name. Many macros available in the standard LaTeX document classes are used in beamer, although sometimes the result might look different. The beamer package comes with extensive documentation, which is included in most TeX distributions (such as TeX Live) and available online on CTAN. As with other document classes, the output document is likely in portable document format (PDF). This imposes certain limitations on animations well-known from commercial software. However, it has always been a strength of (LA)TeX to let the author focus on the content, and beamer extends this concept to slide presentations.

## 2 The very basics

To create a presentation, we set the document class to beamer:

```
\documentclass{beamer}
```

The main difference between standard LaTeX document classes and beamer is that content does not continuously "flow" across multiple pages, but is limited to a single slide. The environment name frame is used for slides, usually to produce a single slide (sometimes several, but we will get to that later). A frame contains a title and a body. Furthermore, at the bottom of every frame, beamer automatically adds a navigation menu.

```
\begin{frame}
  \frametitle{Slide title}
  %Slide body
\end{frame}
```

## 3 In the preamble

As with the standard LaTeX document classes, the preamble serves to load packages, define the content of the title slide, and alter the appearance of the presentation.

### 3.1 Presentation title

Beamer reuses the standard LaTeX macros to create the title page: \title, \author, and \date.

```
\title{Beamer presentation title}
\author{Presenter's name}
\date{\today}
```

We use these further below to create a title page frame.

### 3.2 Presentation appearance

In beamer, "themes" change the appearance of a presentation. Themes define the style and the color of a presentation. By default, beamer loads the rather bland default theme. To change the theme to something more appealing, we can use the following command in the preamble with a theme name we like:

```
\usetheme{default} % default theme
```

There are a great number of themes distributed with LaTeX. They are often named after cities. Try for example: Berkeley, Madrid, or Singapore, to name a few (figure 1). Also, look for beamer theme galleries online.

## 4 Presentation slides

### 4.1 Creating a basic frame

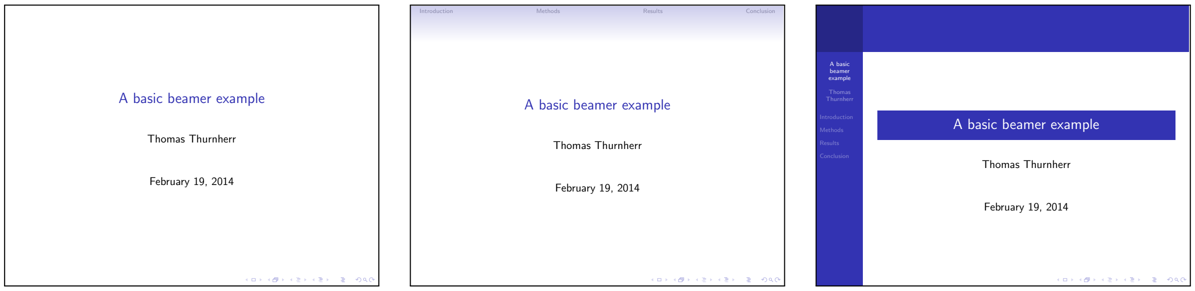
A frame may contain a number of different things, including simple text, formulas, figures, tables, etc. Most often, however, a frame contains numbered or bulleted lists. To create lists, we use the standard list environments: enumerate and itemize. An example of a bulleted list is shown below:

```
\begin{frame}
  \frametitle{List types in \LaTeX}
  \begin{itemize}
    \item Bulleted list: itemize
    \item Numbered list: enumerate
    \item Labeled list: description
  \end{itemize}
\end{frame}
```

Although the result looks different, lists work in the same way as in other document classes; they can be nested and customized to your needs.

### 4.2 Title page frame

We have already seen how to define a title in the preamble. Now we want to use this title to create a title page frame.


A basic beamer example

Thomas Thurnherr

February 19, 2014

Introduction Methods Results Conclusion

A basic beamer example

Thomas Thurnherr

February 19, 2014

A basic beamer example
Thomas Thurnherr
Introduction
Methods
Results
Conclusion

A basic beamer example

Thomas Thurnherr

February 19, 2014

**Figure 1**: Title page frames for beamer themes: `default`, `Singapore`, and `Berkeley`.

```
\begin{frame}
  \titlepage
  % alternatively \maketitle can be used
\end{frame}
```

### 4.3 Table of contents

To add structure to a presentation and create an outline, we can use `\section` and `\subsection`, together with `\tableofcontents`. These commands are used outside of frames. To create an outline at the beginning of a presentation, we use:

```
\section{Presentation Outline}
\begin{frame}
  \frametitle{Outline}
  \tableofcontents
\end{frame}
```

For long presentations, it may make sense to show the outline again at the beginning of a new `\section`. We use the option `currentsection` to `\tableofcontents` to highlight the current section.

```
\section{New Section Title}
\begin{frame}
    \frametitle{Outline}
    \tableofcontents[currentsection]
  \end{frame}
```

### 4.4 Adding figures

Frames are single entities and therefore figures and tables do not need to be wrapped in their respective floating environments. Also, we probably do not wish to add a caption. Therefore, to show a figure, `\includegraphics` with appropriate alignment and scaling (see the `graphicx` package [3]) is sufficient:

```
\begin{frame}
  \frametitle{Adding a figure to a frame}
  \centering
  \includegraphics[width=0.8\linewidth]
    {some-figure-file}
\end{frame}
```

Similarly with tables, we omit the `table` environment and directly use `tabular`.

### 4.5 Defining multiple columns

By default, content is stacked vertically. Therefore, if you have a list, then a figure, then a text paragraph, first the list is produced, with the figure below and lastly the text. Often, it's desirable to position content next to each other. There are two (identical) methods to split a slide horizontally into two or more columns: the `minipage` environment and the `beamer`-specific `columns` environment.

```
\begin{frame}
  \frametitle{Two column example: minipage}
  \begin{minipage}{0.48\linewidth}
    % content column 1
  \end{minipage}
  \quad % adds some whitespace
  \begin{minipage}{0.48\linewidth}
    % content column 2
  \end{minipage}
\end{frame}

\begin{frame}
  \frametitle{Two column example: columns}
  \begin{columns}
    \begin{column}{0.48\linewidth}
      % content column 1
    \end{column}
    \quad
    \begin{column}{0.48\linewidth}
      % content column 2
    \end{column}
  \end{columns}
\end{frame}
```

## 5 Simple animations

It may be an exaggeration to use the word "animations". What I will show is merely how to add, remove and replace parts of the content, primarily text. However, I believe this is good enough to keep the audience interested, everything else is just a distraction.

## 5.1 Add items dynamically

The beamer command \pause adds content gradually, by pausing and waiting for the presenter to press a button. For example, we can use \pause in a list to reveal one item after another. You might wonder how this is possibly translated into a PDF. There is really no magic to it; LaTeX just produces three slides with the same page number, adding an extra item one each subsequent slide. Try for yourself:

```
\begin{frame}
  \frametitle{Usage of pause}
  \begin{itemize}
    \item First item, shown with the slide
      \pause
    \item Next item, revealed after pressing
      a button
      \pause
    \item Last item, revealed after pressing
      a button again
  \end{itemize}
\end{frame}
```

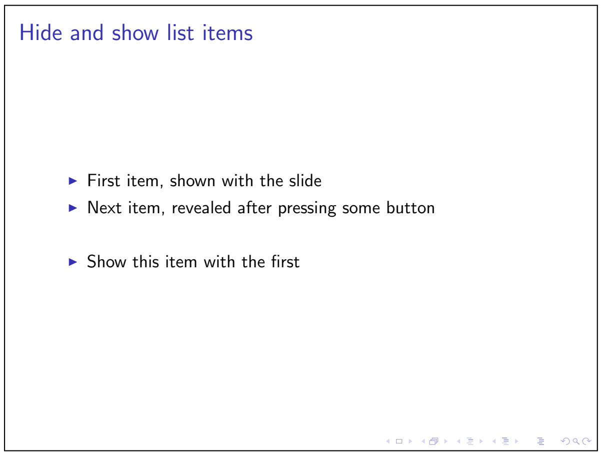
## 5.2 Hide and show content

"Overlays" is a slightly more sophisticated concept. Overlays use pointed brackets to hide, reveal and overwrite content. For example, the specification \item<1-> means: "from slide 1 on" (see figure 2).

```
\begin{frame}
  \frametitle{Hide and show list items}
  \begin{itemize}
    \item<1-> First item, shown with the slide
    \item<2-> Next item, revealed after
      pressing some button
    \item<3-> Last item, revealed again after
      pressing some button
    \item<1-> Show this item with the first
  \end{itemize}
\end{frame}
```

We can also combine ranges of numbers. Assuming more than 7 overlays, to show an item on all but slides 3 and 6, we use: \item<-2,4-5,7->. Items always occupy their space, even if they are not shown. Joseph Wright's article elsewhere in this issue provides more examples [5].

This syntax works with other content too, as implemented in the commands \uncover and \only. The difference between them is that \uncover occupies space when hidden, whereas \only does not, and can therefore be used to overwrite previous content.


Hide and show list items

- First item, shown with the slide
- Next item, revealed after pressing some button

- Show this item with the first

**Figure 2**: Hide and show list items.

```
\begin{frame}
  \frametitle{Hide and show content}
  \uncover<1> { % adds content }
  \uncover<2> { % add additional content }
\end{frame}

\begin{frame}
  \frametitle{Hide and overwrite content}
  \only<1> { % adds content }
  \only<2> { % replaces previous content }
\end{frame}
```

Similar to the itemize example above, much more sophisticated overlays can be created using \uncover and \only.

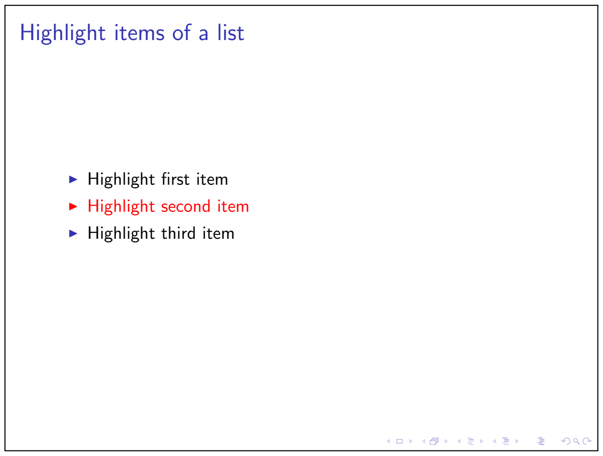
## 5.3 Highlighting items

Besides hiding and revealing, we can also highlight text upon a button press. In beamer, this is called an alert (see figure 3):

```
\begin{frame}
  \frametitle{Highlight items of a list}
  \begin{itemize}
    \item<alert@1> Highlight first item
    \item<alert@2> Highlight second item
    \item<alert@3> Highlight third item
    \item<4- | alert@4> Combine reveal and
      highlight
  \end{itemize}
\end{frame}
```

## 6 Citations and bibliography

You can generate a bibliography the same way as with a standard LaTeX document class. Personally, I prefer to show the bibliography entries on the same slide where they are cited. I use the biblatex package [2]


Highlight items of a list

- Highlight first item
- Highlight second item
- Highlight third item

**Figure 3**: Highlight list items.

for this, which provides a variety of methods. The commands `\footcite` and`\footfullcite` produce references according to the style (e.g. `authoryear`), and full references respectively. I use an external BibTeX file to store references. Here is an example.

```
% Preamble
\usepackage[backend=biber, maxnames=2,
  firstinits=true,style=authoryear]{biblatex}
\bibliography{path/to/references.bib}

% Citations
\begin{frame}
  \frametitle{Citing other people's work}
  \begin{itemize}
    \item Full citation
      \footfullcite{knuth86}
    \item Author-year citation
      \footcite{knuth86}
  \end{itemize}
```

If your preferred style is not available, you can define your own citation style using the `biblatex` command `\DeclareCiteCommand`. The invocation below prints references as footnotes, showing the author, year, and journal title.

```
% Preamble
\usepackage[backend=biber, maxnames=2,
  firstinits=true]{biblatex}
\bibliography{path/to/references.bib}
\DeclareNameAlias{sortname}{first-last}
\DeclareCiteCommand{\footcustomcite}{}{%
    \footnote{\printnames[author]{author},
      \printfield{year},
      \printfield{journaltitle}
      \printfield{booktitle}}}{;}{}
```

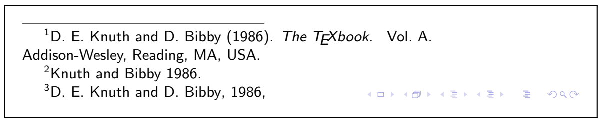

[1]D. E. Knuth and D. Bibby (1986). *The TEXbook*. Vol. A. Addison-Wesley, Reading, MA, USA.
[2]Knuth and Bibby 1986.
[3]D. E. Knuth and D. Bibby, 1986,

**Figure 4**: Full, `authoryear`-style, and custom citation of *The TEXbook*.

```
% Citations
\footcustomcite{knuth86}
```

Figure 4 shows a citation (for *The TEXbook*) in the full, `authoryear`, and above custom styles.

## 7 Creating handouts

If you teach a class or give a talk it might be appropriate to provide handouts. The `beamer` document class option `handout` reduces overlays to a single slide and removes the navigation bar at the bottom. In addition, we can combine multiple slides to a single physical page, using the `pgfpages` package [4]:

```
\documentclass[handout]{beamer}
\usepackage{pgfpages}
\pgfpagesuselayout{4 on 1}[a4paper,
  border shrink=5mm, landscape]
```

## References

[1] beamer — A LATEX class for producing presentations and slides. `http://www.ctan.org/pkg/beamer`. Accessed: 2014-02-19.

[2] biblatex — Bibliographies in LATEX using BibTeX for sorting only. `http://www.ctan.org/pkg/biblatex`. Accessed: 2014-02-19.

[3] graphicx — Enhanced support for graphics. `http://www.ctan.org/pkg/graphicx`. Accessed: 2014-02-19.

[4] pgf — Create PostScript and PDF graphics in TEX. `http://www.ctan.org/pkg/pgf`. Accessed: 2014-02-19.

[5] Joseph Wright. The `beamer` class: Controlling overlays. *TUGboat*, 35(1):31–33, 2014.

◇ Thomas Thurnherr
`texblog (at) gmail dot com`
`http://texblog.org`

# The beamer class: Controlling overlays

Joseph Wright

There was a question recently on the TEX StackExchange site (Gil, 2014) about the details of how slide overlays work in the beamer class (Tantau, Wright, and Miletić, 2013). The question itself was about a particular input syntax, but it prompted me to think that a slightly more general examination of how overlays would be helpful to beamer users.

A word of warning before I start: don't overdo overlays! Having text or graphics appear or disappear on a slide can be useful but is easy to over-use. I'm going to focus on the mechanics here, but that doesn't mean that they should be used in every beamer frame you create.

## 1 Overlay basics

Before we get into the detail of how beamer deals with overlays, I'll first give a bit of background to what they are. The beamer class is built around the idea of frames:

```
\begin{frame}
  \frametitle{A title}
  % Frame content
\end{frame}
```

which can produce one or more slides: individual pages of output that will appear on the screen. These separate slides within a frame are created using overlays, which is the way the beamer manual describes the idea of having the content of individual slides varying. Overlays are "contained" within a single frame: when we start a new frame, any overlays from the previous one stop applying.

The most basic way to create overlays is to explicitly set up individual items to appear on a particular slide within the frame. That's done using the (optional) overlay argument that beamer enables for many document components; this overlay specification is given in angle brackets. The classic example is a list, where the items can be made to appear one at a time.

```
\begin{frame}
  \begin{itemize}
    \item<1-> Visible from the 1st slide
    \item<2-> Visible from the 2nd slide
    \item<3-> Visible from the 3rd slide
    ...
  \end{itemize}
\end{frame}
```

As you can see, the overlay specification here is simply the first slide number we want the item to be on followed by a - to indicate "and following slides". We can make things more specific by giving only a single slide number, giving an ending slide number and so on.

```
\begin{frame}
  \begin{itemize}
    \item<1> Visible on the 1st only
    \item<-3> Visible on the
      1st to 3rd slides
    \item<2-4,6> Visible on the
      2nd to 4th slides, and the 6th slide
  \end{itemize}
\end{frame}
```

The syntax is quite powerful, but there are at least a couple of issues. First, the slide numbers are hard-coded. That means that if I want to add something else in before the first item I've got to renumber everything. Secondly, I'm having to repeat myself. Luckily, beamer offers a way to address both of these concerns.

## 2 Auto-incrementing the overlay

The first tool beamer offers is the special symbol + in overlay specifications. This is used as a place holder for the "current overlay", and is automatically incremented by the class. To see it in action, I'll rewrite the first overlay example without any fixed numbers.

```
\begin{frame}
  \begin{itemize}
    \item<+-> Visible from the 1st slide
    \item<+-> Visible from the 2nd slide
    \item<+-> Visible from the 3rd slide
    ...
  \end{itemize}
\end{frame}
```

What's happening here? Each time beamer finds an overlay specification, it automatically replaces all of the + symbols with the current overlay number. It then advances the overlay number by 1. So in the above example, the first + is replaced by a 1, the second by a 2 and the third by a 3. So we get the same behaviour as in the hard-coded case, but this time if I add another item at the start of the list I don't have to renumber everything.

There are of course a few things to notice. The first overlay in a frame is number 1, and that's what beamer sets the counter to at the start of each frame. To get the second item in the list to appear on slide 2, we still require an overlay specification for the first item: I could have skipped the <1-> in the hard-coded example and nothing would have changed. The second point is that *every* + in an overlay specification gets replaced by a given value. We'll see later there are places you might accidentally add a + to mean "advance by 1": don't do that!

## 3 Reducing redundancy

Using the + approach has made our overlays flexible, but I've still had to be repetitive. Handily, beamer helps out there too by adding an optional argument to the list which inserts an overlay specification for each line:

```
\begin{frame}
  \begin{itemize}[<+->]
    \item Visible from the 1st slide
    \item Visible from the 2nd slide
    \item Visible from the 3rd slide
     ...
  \end{itemize}
\end{frame}
```

Notice that this is needs to be inside the "normal" `[ ... ]` set up for an optional argument. Applying an overlay to every item might not be exactly what you want: you can still override individual lines in the standard way.

```
\begin{frame}
  \begin{itemize}[<+->]
    \item Visible from the 1st slide
    \item Visible from the 2nd slide
    \item Visible from the 3rd slide
    \item<1-> Visible from the 1st slide
     ...
  \end{itemize}
\end{frame}
```

Remember not to overdo this effect: just because it's easy to reveal every list line by line doesn't mean you should!

## 4 Repeating the overlay number

The + syntax is powerful, but as it always increments the overlay number it doesn't allow us to remove the hard-coded numbers from a case such as

```
\begin{frame}
  \begin{itemize}
    \item<1-> Visible from the 1st slide
    \item<1-> Visible from the 1st slide
    \item<2-> Visible from the 2nd slide
    \item<2-> Visible from the 2nd slide
     ...
  \end{itemize}
\end{frame}
```

For this case, beamer offers another special symbol, a single period '`.`', as in:

```
\begin{frame}
  \begin{itemize}
    \item<+-> Visible from the 1st slide
    \item<.-> Visible from the 1st slide
    \item<+-> Visible from the 2nd slide
    \item<.-> Visible from the 2nd slide
     ...
  \end{itemize}
\end{frame}
```

What happens here is that `.` can be read as "repeat the overlay number of the last +". So the two + overlay specifications create one slide each, while the two lines using `.` in the specification 'pick up' the overlay number of the preceding +. (The beamer manual describes the way this is actually done, but I suspect that's less clear than thinking of this as a repetition!)

Depending on the exact use case, you might want to combine this with the "reducing repeated code" optional argument, with `<.->` as an override.

```
\begin{frame}
  \begin{itemize}[<+->]
    \item Visible from the 1st slide
    \item<.-> Visible from the 1st slide
    \item Visible from the 2nd slide
    \item<.-> Visible from the 2nd slide
     ...
  \end{itemize}
\end{frame}
```

## 5 Offsets

A combination of + and `.` can be used to convert many "hard-coded" overlay set ups into "relative" ones, where the slide numbers are generated by beamer without you having to work them out in advance. However, there are still cases it does not cover. To allow even more flexibility, beamer has the concept of an "offset": an adjustment to the number that is automatically inserted. Offset values are given in parentheses after the + or `.` symbol they apply to, for example:

```
\begin{frame}
  \begin{itemize}
    \item<+(1)-> Visible from the 2nd slide
    \item<+(1)-> Visible from the 3rd slide
    \item<+->    Visible from the 3rd slide
  \end{itemize}
\end{frame}
```

Notice that this adjustment only applies to the substitution, so both the second and third lines above end up as `<3->` after the automatic replacement. If you try the demo, you'll also notice that none of the items appear on the first slide!

Perhaps a more realistic example for where an offset is useful is the case of revealing items "out of order", where the full list makes sense in some other way. With hard-coded numbers this might read

```
\begin{frame}
  \begin{itemize}
    \item<1-> Visible from the 1st slide
    \item<2-> Visible from the 2nd slide
    \item<1-> Visible from the 1st slide
    \item<2-> Visible from the 2nd slide
    ...
  \end{itemize}
\end{frame}
```

which can be made "flexible" with a set up such as

```
\begin{frame}
  \begin{itemize}
    \item<+-> Visible from the 1st slide
    \item<+-> Visible from the 2nd slide
    \item<.(-1)-> Visible from the
                  1st slide
    \item<.-> Visible from the 2nd slide
    ...
  \end{itemize}
\end{frame}
```

or the equivalent

```
\begin{frame}
  \begin{itemize}
    \item<+-> Visible from the 1st slide
    \item<.(1)-> Visible from the 2nd slide
    \item<.-> Visible from the 1st slide
    \item<+-> Visible from the 2nd slide
    ...
  \end{itemize}
\end{frame}
```

As shown, we can use both positive and negative offsets, and these work equally well for `+` and `.` auto-generated values. You have to be slightly careful with negative offsets; while beamer will add additional slides for positive offsets, if you offset to below a final value of `0` then errors will crop up. With this rather advanced setup, which version is easiest for you to follow will be down to personal preference.

Notice that positive offsets do not include a `+` sign, but are just given as an unsigned integer: remember what I said earlier about all `+` symbols being replaced. If you try something like `<+(+1)>`, your presentation will compile but you'll have a lot of slides!

## 6 Pausing general text

The beamer class offers a very simple `\pause` command to split general material into overlays. A classic problem that people run into is combining that idea with the `+` approach to making overlays. For example, the following creates *four* slides:

```
\begin{frame}
  \begin{itemize}
    \item<+-> Visible from the 1st slide
    \item<+-> Visible from the 2nd slide
  \end{itemize}
  \pause
  Text after the list
\end{frame}
```

If you read the beamer manual carefully, this is what is supposed to happen here, but the more important question is how to get what you (probably) want: three slides.

The answer is to use `\onslide`: the `\pause` command is by far the most basic way of making overlays, and simply doesn't "know" how to work with `+-`. In contrast, `\onslide` uses exactly the same syntax we've already seen for overlays:

```
\begin{frame}
  \begin{itemize}
    \item<+-> Visible from the 1st slide
    \item<+-> Visible from the 2nd slide
  \end{itemize}
  \onslide<+->
  Text after the list
\end{frame}
```

As we are then using the special `+` syntax for all of the overlays, everything is properly tied together and gives the expected result: three slides.

## 7 Summary

The beamer overlay feature can help you set up complex and flexible overlays to generate slides with dynamic content. By using the tools carefully, you can make your input easier to read and maintain.

## References

Gil, Yossi. "Relative overlay specification in beamer?" `http://tex.stackexchange.com/q/154521`, 2014.

Tantau, Till, J. Wright, and V. Miletić. "The beamer class". Available from CTAN, `macros/latex/contrib/beamer`, 2013.

◇ Joseph Wright
2, Dowthorpe End
Earls Barton
Northampton
NN6 0NH
United Kingdom
`joseph.wright (at) morningstar2 dot co dot uk`

# Boxes and more boxes

Boxes and more boxes

Ivan R. Pagnossin

## Abstract

Boxes are a key concept in (LA)TEX as well as an useful tool, though not a very well known one. I believe this is due to the success of LATEX in providing a high level interface to TEX, which allows the user to produce beautiful documents without getting acquainted with the concept of boxes. But it is useful to know the box concept because it enhances our understanding of TEX, helps us to avoid and solve problems and can even propose solutions for unusual circumstances.

In this paper we reproduce, step by step, the reflection effect in the title of this article in order to gain some knowledge of the box concept.

## 1 Box basics

Among the fundamental ingredients of a LATEX document, there are *boxes:* imaginary rectangles which enclose letters, lines, pages, paragraphs, figures, tables etc. Indeed, TEX does not typeset letters or characters, lines ... but boxes!

You may have already used boxes to avoid breaking some word or maybe to frame an equation. But they are far more useful than this. A simple but instructive example is the reflection effect in the title of this paper, which requires one customized box. Let's learn something about it.

Figure 1 shows the box associated with the letter **g**. It has three dimensions: *height*, *width* and *depth*. It is these dimensions that LATEX actually cares about, not the content of the box. In other words, LATEX does not "see" the letter **g**, only its box. This is the important fact to understand.

We can ask LATEX to show us this box: just write `\framebox{g}`. Well, first we must set `\fboxsep` to zero, in order to remove the extra space around **g**: `\setlength{\fboxsep}{0in}`.

Much more important than just showing the box, `\framebox` builds a box with whatever you want inside it. For instance, `\framebox{Boxes}`. Now, **Boxes** is a *single* box, as unbreakable as the letter **g** itself. By the way, this is another property to keep in mind: *boxes cannot be broken.*

Another interesting property is this: *the contents of a box need not lie inside it.* You may have noticed that, given the contents as an argument, the `\framebox` command sets the dimensions of the box to those of the contents (in reality, to the "sub-boxes" that compose the contents). But you can define the dimensions explicitly as well. For example,

```
\framebox[.67in]{Boxes}
```

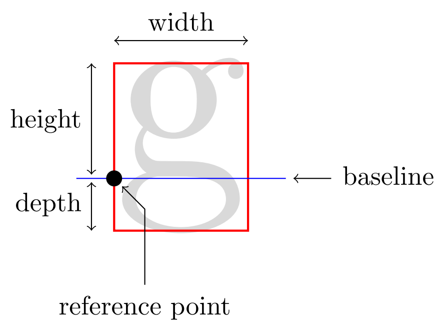


**Figure 1**: The dimensions of a box. Everything that is visible in a document produced by LATEX is in one or more boxes, which is why they are so fundamental.

produces $\boxed{\quad\text{Boxes}\quad}$, a box $\frac{2}{3}$ in wide. That is, a box which occupies more space than its contents. You can also define the alignment of the content with the box, to the left, center or right, as shown below:

- `\framebox[.67in][l]{Boxes}` $\boxed{\text{Boxes}\qquad}$
- `\framebox[.67in][c]{Boxes}` $\boxed{\quad\text{Boxes}\quad}$
- `\framebox[.67in][r]{Boxes}` $\boxed{\qquad\text{Boxes}}$

In a similar fashion, we may create a box which occupies less space than its contents; a zero-width box, for instance:

```
\framebox[0in][l]{Boxes}
```

Try it yourself and see that **Boxes** overlaps the text at its right. This happens because, as we have seen, LATEX uses the dimensions of the letters (boxes) to place them side by side in a row. However, the box we have just created occupies no (horizontal) space. It looks like it is not there, though its contents are (try also changing the alignment parameter).

## 2 Reflecting, scaling, coloring

Another command we will need to produce the reflection effect is `\scalebox`, defined in the package graphicx. Its syntax is `\scalebox{`$f_x$`}[`$f_y$`]{`*a box*`}`. It changes the dimensions of the box *and its contents* according to the horizontal and vertical scaling factors $f_x$ and $f_y$ (respectively) and creates a new box that encloses this new content. Thus, as a first step toward our goal, to get **BoxesBoxes** we write

```
\scalebox{1}[-1]{Boxes}Boxes
```

BoxesBoxes

This means we instruct LATEX to multiply the width by $f_x = 1$ (hence, nothing changes on the horizontal) and the height ($> 0$ in our case) and the depth ($= 0$) by $f_y = -1$. As a result, we have a vertical reflection of **Boxes**. A hint: the commands defined by graphicx receive boxes as arguments, not just figures (which are boxes too). Since the package was designed to handle figures, it is common to think that its commands apply only to figures, but this is not true.

Let's proceed. How can we place Boxes below **Boxes**? Answer: by constructing it in a way that occupies no horizontal space, that is, by putting it in a zero-width box (new code in black, previous code in gray):

Boxes
Boxes

```
\makebox[0in][l]{\scalebox{1}[-1]{Boxes}}Boxes
```

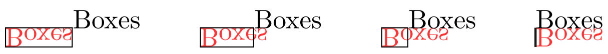
We have also swapped `\framebox` for `\makebox`. These two commands are equivalent, except that the former draws a frame around the box, while the latter does not.

Finally, we choose the reflection color (use the package xcolor):

Boxes
Boxes

```
  \makebox[0in][l]{\scalebox}{1}[-1]{%
    \textcolor{red!80}{Boxes}}Boxes
```

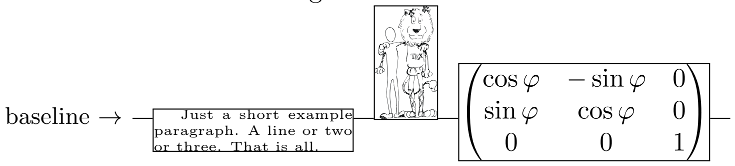
The sequence below illustrates the changing of the box width, from its *natural width* (defined by the content) down to zero. Notice that the non-inverted **Boxes** always starts immediately after the frame (the box), not after the content (Boxes).



## 3 Boxes upon boxes

There are boxes far more complex than these. As I mentioned, everything that is visible in a document produced by LaTeX is in one or more boxes. There are even invisible elements in your document that are boxes too. An example is the paragraph indentation, which is nothing more than an empty box of width `\parindent`. In all cases, all of the concepts we have just seen remain valid.

Let's see an example. The line below contains three boxes plus some filling lines (which are boxes too): the first is an entire paragraph, the second is a figure and the third a mathematical table (or matrix, if you prefer). All of them have height, width and depth, are unbreakable, and are placed side by side with their *reference points* aligned (fig. 1) over an imaginary line called the *baseline*, represented by the horizontal filling lines.


baseline →
Just a short example paragraph. A line or two or three. That is all.
cos φ −sin φ 0
sin φ cos φ 0
0 0 1

The paragraph box is aligned on the baseline of its first line. The figure, on the other hand, inserted with the `\includegraphics` command (package graphicx), has depth zero (but since we've enclosed it in an `\fbox` for the example, it is not quite zero here). Finally, the matrix has unequal height and depth (it's centered on the math axis; the LaTeX code that produces this line is at the end of this paper). The important thing here is to notice that **the placement of all these boxes follow exactly the same rules as those followed by letters**, since they are all in boxes.

So, whenever you insert a figure or table in your document, see them as "big **g** letters". In other words, *see the boxes!*

## 4 Further reading

To learn more about this subject, you can study the environment minipage, the commands `\parbox`, `\raisebox` and `\rule` in section 4.7 of [1] and/or appendix A.2 of [2]. Chapter 11 of [3] is mandatory. Moreover, have a look at the commands defined by the package graphicx.

## Appendix: Our unusual line of text

The code used to produce the unusual text line discussed above in this paper is shown below. You will need the packages graphicx and amsmath to run it.

```
\setlength\fboxsep{0pt}\noindent\hrulefill
\fbox{\begin{minipage}[t]{0.16\textwidth}
  \setlength{\parindent}{1em}
  \tiny Just a short example paragraph.
        A line or two or three. That is all.
\end{minipage}}%
\hrulefill
\fbox{%
  \includegraphics[width=0.05\textwidth]{img}}%
\hrulefill
\fbox{$\left(\begin{matrix}
  \cos\varphi & -\sin\varphi & 0 \\
  \sin\varphi &  \cos\varphi & 0 \\
  0           &  0           & 1
\end{matrix}\right)$}%
\hrulefill
```

## Acknowledgments

The author thanks Karl Berry and Barbara Beeton for invaluable help and improvements to this article.

## References

[1] H. Kopka and P. W. Daly. *A Guide to LaTeX*. Addison-Wesley, 3rd edition, 2004.

[2] F. Mittelbach and M. Goossens. *The LaTeX Companion*. Addison-Wesley, 2nd edition, 2004.

[3] D. E. Knuth. *The TeXbook*. Addison-Wesley, 1984.

◇ Ivan R. Pagnossin
`ivan dot pagnossin (at) gmail dot com`

## Glisterings: Glyphs, long labels

Peter Wilson

Ek gret effect men write in place lite;
Th' entente is al, and nat the lettres space.

*Troilus and Criseyde*, Geoffrey Chaucer

The aim of this column is to provide odd hints or small pieces of code that might help in solving a problem or two while hopefully not making things worse through any errors of mine. This installment presents some items about glyphs.

### 1 Asterism

A symbol called an *asterism* is a simple kind of ornament consisting of three asterisks, and is supplied as a glyph in some fonts. It is typically used as an anonymous division, looking like this:

⁂

Stephen Moye [8] posted a macro to make one if it was not available otherwise. This was based on some earlier code from Peter Flynn [4]. The code below is my version.

```
\newcommand*{\asterism}{%
  \raisebox{-.3em}[1em][0em]{%  OK for 10-12pt
    \setlength{\tabcolsep}{0.05em}%
    \begin{tabular}{@{}cc@{}}%
      \multicolumn{2}{c}*\\[-.75em]%
        *&*%
    \end{tabular}%
  }}
```

The asterism above was printed by:

```
\par{\centering \asterism\par}
```

### 2 Raising a character

'Maximus_Rumpas' wrote to `ctt` along the following lines:

*I am writing some Latin text within a document: GALL : REG : IACO : MAG : BRITA : REG I need to raise the colon between the abbreviated text to the centre of the text line rather than, as normal, aligned at the bottom of the text. I use* `\textperiodcentered` *for a single period but I can't find anything similar for colons.*

Heiko Oberdiek [9] responded with:

The following solution centers the colon using an uppercase letter for comparison. Also it makes the colon active inside an environment for easier writing.

```
\documentclass{article}
\begingroup
  \lccode`\~=`\:%
  \lowercase{\endgroup
  \newenvironment{vccolon}{%
    \catcode`\:=\active
    \let~\textcoloncentered
    \ignorespaces
  }{\ifhmode\unskip\fi}}
\newcommand{\textcoloncentered}{}
\DeclareRobustCommand*{\textcoloncentered}{%
  \begingroup
    \sbox0{T}%
    \sbox2{:}%
    \dimen0=\ht0 %
    \advance\dimen0 by -\ht2 %
    \dimen0=.5\dimen0 %
    \raisebox{\dimen0}{:}%
  \endgroup}

\begin{document}
\begin{vccolon}
GALL : REG : IACO : MAG : BRITA : REG
\end{vccolon}
\end{document}
```

Following on from this Dan Luecking suggested using a `\valign`:

```
\def\textcoloncentered{%
  \valign{&##\cr\vphantom{T}\cr\vfil\hbox{:}%
  \vfil\cr}}
```

also remarking that perhaps a simple box raised by some multiple of ex would do as well.

I tried all three suggestions and decided that

```
\DeclareRobustCommand*{\textcoloncentered}{%
  \raisebox{.2ex}{:}}
```

gave a satisfying result, also enabling the height to be adjusted to optically center the colon if necessary..

Heiko's result:

GALL : REG : IACO : MAG : BRITA : REG

Dan's result:

GALL : REG : IACO : MAG : BRITA : REG

My result:

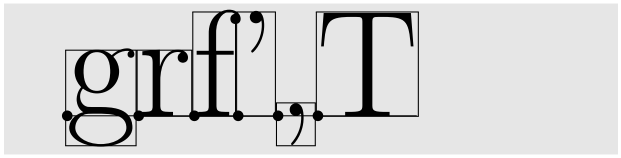
GALL : REG : IACO : MAG : BRITA : REG

### 3 Boxing a glyph

Paul Kaletta wanted to be able to draw a box around a glyph similar to the example in chapter 11 of *The TeXbook*. Herbert Voß responded with [11] (slightly edited):

```
\documentclass{article}
\usepackage[T1]{fontenc}
\usepackage{lmodern}
\newsavebox\CBox
\makeatletter
\def\Gbox#1{\begingroup
  \unitlength=1pt
  \fboxsep=0pt\sbox\CBox{#1}%
  \leavevmode
  \put(0,0){\line(1,0){\strip@pt\wd\CBox}}%
  \fbox{#1}%
  \put(-\strip@pt\wd\CBox,0){\circle*{4}}
  \endgroup}
\makeatother
\begin{document}
\begingroup
\fontsize{2cm}{2.2cm}\selectfont
\Gbox{g}\Gbox{r}\Gbox{f}%
\Gbox{'}\Gbox{,}\Gbox{T}
\endgroup
\end{document}
```

Herbert's demonstration code results in:



## 4 Glyph widths

'PmI' wrote to `ctt`:

*I'm trying to put some text in a box, so that I can calculate the box dimensions, but neither hbox nor mbox seem to want to perform linebreaks, and parbox needs a width argument, which is actually what I want to compute so it's useless . . . example follows:*

```
\filltestbox{I need \\ the width \\
            of this text}
\begin{minipage}{\testboxwidth}
```

Several respondents mentioned that the varwidth package [1] does this, but two other solutions were provided for when the text was simple [3].

Ulrike Fischer proposed using a tabular and measuring its width.

```
\newsavebox\testbox
\newcommand{\filltestbox}[1]{%
  \savebox\testbox{%
    \begin{tabular}{@{}l@{}}#1\end{tabular}}}
\newcommand*{\testboxwidth}{\the\wd\testbox}
```

Alternatively, as an exercise Donald Arseneau, the author of the varwidth package, proposed this:

```
\newcommand{\filltestbox}[1]{%
  \setbox\testbox\vbox{%
  \def\\{\unskip\egroup\hbox\bgroup%
         \ignorespaces}%
  \hbox\bgroup\ignorespaces #1\unskip\egroup}}
```

which makes a vbox (called `\testbox`) containing a list of hboxes whose contents are the pieces of text between `\\`.

In my limited testing, the two approaches yield the same final result. For instance:

```
\filltestbox{Some \\
             text of which I want to know \\
             the width. \\
             There can be \\
             only one paragraph.}
\fbox{%
\begin{minipage}{\testboxwidth}
\usebox\testbox
\end{minipage}}
```

results in:

Some
text of which I want to know
the width.
There can be
only one paragraph.

## 5 Font size

Gonzalo Medina Arellano asked on `ctt`:

*Let's say I use 12pt as a class option. How can I find the exact values of the size obtained with the standard commands* `\tiny`*,* `\scriptsize`*, . . . ,* `\huge`*, and* `\Huge`*?*

Several respondents, Bob Tennent [10] among them, suggested looking at the appropriate `.clo` file, such as `size12.clo` for the article or report classes' `12pt` option or `bk12.clo` for the book class, which lists the font and baseline sizes for the several size commands.

Dan Luecking [6] added to this, saying (edited a bit):

*You can determine the current font size within a document without knowing what size command was last issued. The macro* `\f@size` *holds the font size [as a number (10, 12, 14.4, . . . )] and is updated with each size change. For convenience we can define another macro without an* `@` *sign to access it, as in:*

```
\makeatletter
  \newcommand*{\currentfontsize}{\f@size}
\makeatother
```

*Then* `\currentfontsize` *would print it and*
`\typeout{\currentfontsize}`
*would display it on the terminal screen and in the log file.*

It turns out that the font sizes are the same in `size*.clo` and `book*clo` (the differences are in various margin settings). The unofficial LaTeX reference manual provides a table of the font sizes at [5].

I don't believe in labels. I want to do the best I can, all the time. I want to be progressive without getting both feet off the ground at the same time.

*Television and radio interview, March 15, 1964*, Lyndon B. Johnson

## 6 Long labels

Ernest posted to `comp.text.tex` saying [2]:

*I'm trying to change the description environment, so that when labels exceed a certain length, the text following the label starts in the next line instead of starting in the same line. The LaTeX Companion book explains how to do this, however, when I've tried I've found that the lines that contain a long label are typeset a little bit too close to the previous line, not with the usual baselineskip . . .*

The standard `description` environment uses `\descriptionlabel` for setting the contents of the `\item` macro. The *Companion* [7, §3.3] describes various methods of modifying the standard layout by using a different definition for `\descriptionlabel` and/or creating a new kind of `description` list. Ernest's requirement can be met by a modified version of `\descriptionlabel`, the default definition of which is:

```
\newcommand*{\descriptionlabel}[1]{%
  \hspace{\labelsep}%
  \normalfont\bfseries #1}
```

The following is an example of the default appearance of a `description` list:

**Short** A short label.

**Longer label** A longer label.

**A very long label exceeding the available width**
A long label with the text also longer than a single line.

**Medium label** The more typical length of a label and some text.

As you can easily see, it does not handle long `\item` labels in a graceful manner.

This version of the `\descriptionlabel` meets Ernest's requirements:

```
\usepackage{calc} % or xparse
\newlength{\dlabwidth}
\newcommand*{\widedesclabel}[1]{%
  \settowidth{\dlabwidth}{\textbf{#1}}%
  \hspace{\labelsep}%
  \ifdim\dlabwidth>\columnwidth
    \parbox{\columnwidth-\labelsep}%
      {\textbf{#1}\strut}%
  \else
    \textbf{#1}
  \fi}
\let\descriptionlabel\widedesclabel
```

which, when applied to the previous example yields:

**Short** A short label.

**Longer label** A longer label.

**A very long label exceeding the available width**
A long label with the text also longer than a single line.

**Medium label** The more typical length of a label and some text.

### References

[1] Donald Arseneau. The `varwidth` package, March 2009. `http://mirror.ctan.org/macros/latex/contrib/varwidth`.

[2] Ernest. Description environment. Post to `comp.text.tex` newsgroup, 17 June 2010.

[3] Ulrike Fischer and Donald Arseneau. Re: line breaks in boxes (or 'how do i get paragraph parsing in hbox/mbox?'). Post to `comp.text.tex` newsgroup, 26–27 November 2009.

[4] Peter Flynn. Re: Uncommon typography. Post to `comp.text.tex` newsgroup, 4 June 2007.

[5] George Greenwade et al. LaTeX: An unofficial reference manual. `http://svn.gna.org/viewcvs/*checkout*/latexrefman/trunk/latex2e.html#Font-styles`. Package home page: `http://home.gna.org/latexrefman`.

[6] Dan Luecking. Re: How to find the fontsize? Post to `comp.text.tex` newsgroup, 24 August 2010.

[7] Frank Mittelbach and Michel Goossens. *The LaTeX Companion*. Addison Wesley, second edition, 2004. ISBN 0-201-36299-6.

[8] Stephen Moye. Re: asterism. Post to `xetex` mailing list, 11 January 2010.

[9] Heiko Oberdiek. Re: Raising a colon to center. Post to `comp.text.tex` newsgroup, 23 November 2009.

[10] Bob Tennent. Re: How to find the fontsize? Post to `comp.text.tex` newsgroup, 24 August 2010.

[11] Herbert Voß. Re: How to draw a box around the boundaries of a glyph? Post to `comp.text.tex` newsgroup, 27 March 2009.

◇ Peter Wilson
12 Sovereign Close
Kenilworth, CV8 1SQ
UK
herries dot press (at) earthlink dot net

# The **pkgloader** and **lt3graph** packages: Toward simple and powerful package management for LaTeX

Michiel Helvensteijn

## Abstract

This article introduces the `pkgloader` package. I recently wrote this package to address one of the major frustrations of LaTeX: package conflicts. It also introduces `lt3graph`, a LaTeX3 library used by `pkgloader` to do most of the heavy lifting.

## 1 Introduction

LaTeX package conflicts are a common source of frustration. If you are reading this article, you're probably experienced enough with LaTeX to have encountered them more than once. I cannot improve upon the words of Freek Dijkstra [3] on the subject:

"Package conflicts are a hell."

Package conflicts can exist because of the sheer power of TeX [4], the language on which LaTeX is based. Not only is it Turing complete [8], but most of the language can be redefined from within the language itself. This was famously demonstrated with the TeX script `xii.tex`, written by David Carlisle [1] (if you haven't seen it yet, download and compile it now; it's awesome). LaTeX packages can not only add new definitions, but also remove and modify existing ones. They can offer domain specific languages [6], monkey-patch the core language to hook into existing commands [7], and even change the meaning of individual symbols by altering their category code [5]. Put simply, LaTeX packages have free rein. This power can be quite useful, but makes it too easy for independent package authors to step on each others' toes.

A different type of conflict concerns package options. If a package is requested more than once with different options, LaTeX bails out with an error message. This is an understandable precaution. Because of the way package loading works, LaTeX has no way to apply the second set of options. The package will have already been loaded with the first set.

Most package authors are well aware of these problems. Document authors are told to avoid certain package combinations, or to load packages in some specific order. Some of the larger packages are designed to test for the presence of other packages in order to circumvent known conflicts. Unfortunately, this is all done in an ad hoc fashion.

Solving these problems on a case-by-case basis takes time and effort for both document and package authors. It pollutes the code, makes maintenance more difficult, and confuses new users. We need a systematic approach to resolve package conflicts. This is where the power of TeX comes in handy. A package can be written to oversee the package loading process: a LaTeX package manager. It should be easy enough to use for the casual document author, yet powerful enough to allow package authors to hook into it to simplify their development process.

Section 2 introduces `pkgloader`, which I wrote to fulfill this rôle. Reading this section should be enough to give you a general idea of how to use it. Section 3 describes `lt3graph`, a utility library using the LaTeX3 programming layer which does most of the heavy lifting for `pkgloader`. Finally, Section 4 goes through some of the more advanced and planned features of `pkgloader`.

Here is a quick glimpse of `pkgloader` in use:

```
\RequirePackage{pkgloader}
   ...
   \documentclass{article}
   ...
   \usepackage{algorithm}  ⎫
   \usepackage{hyperref}   ⎬ any order
   \usepackage{float}      ⎭
   ...
\begin{document}
   ...
\end{document}
```

**Warning:** This package is still under development. Some of the features described in this article may not yet be fully implemented, and the presented syntax may still change in the coming months. However, its main purpose and the fundamental ideas underlying its implementation are here to stay.

**Community collaboration:** I intend for the development and maintenance of this package to be as open as possible to community collaboration. This package has a very wide scope, and is rather invasive. If done right, it has the potential to become widely used and improve the LaTeX experience for document authors and package authors alike. If done wrong, it will break things and annoy many people.

I hope to bring some useful domain knowledge to the development effort, but there are many LaTeX gurus out there who have more experience and insight than I do. If you like the idea of `pkgloader` and would like to contribute in any way, I encourage you to contact me personally, or to file issues or pull requests through the `pkgloader` Github page:
`github.com/mhelvens/latex-pkgloader`

## 2 The **pkgloader** package (Part 1)

This package was inspired by my PhD research [2], which happens to be all about conflicts between independently developed modules. I also took cues from similar libraries and standards for other languages, such as JavaScript, which is surprisingly similar to LaTeX in many ways.

## 2.1 How to use the package manager

LaTeX packages are generally loaded with one of the commands **\usepackage**, **\RequirePackage**, or **\RequirePackageWithOptions**. In a similar way, document classes are loaded with **\documentclass**, **\LoadClass** or **\LoadClassWithOptions**. Normally, when such a command is reached, the relevant class or package is loaded on the spot. The idea behind pkgloader is to make it the very first file you load: before the document class, and before any other package. Thus, the main file for a LaTeX document using pkgloader would be structured like this:

```
\RequirePackage{pkgloader}
    ⟨document class and packages in any order⟩
\LoadPackagesNow                  } optional
. . .                             }
\begin{document}
    . . .
\end{document}
```

The area between **\RequirePackage{**pkgloader**}** and **\LoadPackagesNow** is called the *pkgloader area*. Inside this area, the loading of all classes and packages is postponed. It also ends automatically upon reaching the end of the preamble. The package manager analyzes the intercepted loading requests and executes them in the proper order and with the proper options, lifting this burden from the user.

## 2.2 A conflict resolution database

The pkgloader package does *not* analyze the actual code of each package in order to detect conflicts. In fact, because TeX is Turing complete, this would be mathematically impossible. The package manager is backed by a database of *rules* for recognizing and resolving known conflicts, as well as performing other neat tricks. The following shows some examples:

```
\Load {float} before {hyperref}
\Load {algorithm} after {hyperref}
\Load {fixltx2e} always early
      because {it fixes some imperfections
               in LaTeX2e}
\Load error if {algorithms && pseudocode}
      because {they provide the same
               functionality and conflict
               on many command names}
```

The first two rules encode some workarounds for the hyperref package, which is notorious for causing conflicts. The first one says that float must be loaded before hyperref. Similarly, the second rule ensures that hyperref is loaded before algorithm. These are the rules that would allow the code on page 39 to compile without problems. Note that neither rule actually loads any packages. They simply tell the package manager how to treat certain pairs of packages, should they ever be requested together in a single document.

The third rule states that fixltx2e must always be loaded, and must be loaded early. The fourth rule states that the algorithms and pseudocode packages should never be loaded together. They both also include a textual reason, which documents the rule, and is included in certain error messages.

To better understand how these rules work, let's dive into their underlying model: a directed graph.

## 3 The lt3graph Package

The pkgloader package is written in the experimental LaTeX3 programming layer expl3, which gives us something akin to a traditional imperative programming language, with data structures, while loops, and so on. Let's face it, TeX is no one's first choice for a programming language. But expl3 makes it bearable. So kudos to the LaTeX3 team!

To represent graphs, I wrote a data-structure package called lt3graph.[1] This library was born as a means to an end, but has grown into a full-fledged general-purpose data structure for representing and analyzing directed graphs.

A directed graph contains vertices (nodes) and edges (arrows). Using lt3graph, an example graph may be defined as follows:

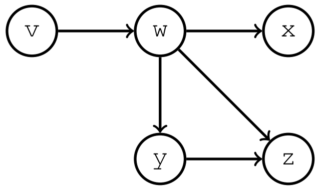
```
\ExplSyntaxOn
\graph_new:N    \l_my_graph
\graph_put_vertex:Nn \l_my_graph {v}
\graph_put_vertex:Nn \l_my_graph {w}
\graph_put_vertex:Nn \l_my_graph {x}
\graph_put_vertex:Nn \l_my_graph {y}
\graph_put_vertex:Nn \l_my_graph {z}
\graph_put_edge:Nnn \l_my_graph {v} {w}
\graph_put_edge:Nnn \l_my_graph {w} {x}
\graph_put_edge:Nnn \l_my_graph {w} {y}
\graph_put_edge:Nnn \l_my_graph {w} {z}
\graph_put_edge:Nnn \l_my_graph {y} {z}
\ExplSyntaxOff
```

Each vertex is identified by a *key*, which, to this library, is a string: a list of characters with category code 12 and spaces with category code 10. An edge is then declared between two vertices by referring to their keys. By supplying an additional argument to the functions above, you can store arbitrary data in

[1] Bearing the prefix lt3 rather than the more common prefix l3 indicates that the package is not officially supported by the LaTeX3 team.

a vertex or edge for later retrieval. Let's use Ti*k*Z to visualize this graph:

```
\newcommand{\vrt}[1]
  {\node(#1){\ttfamily\vphantom{Iy}#1};}
\begin{tikzpicture}[every path/.style=
                 {line width=1pt,->}]
   \matrix[nodes={circle,draw},
         row sep=1cm, column sep=1cm,
         execute at begin cell=\vrt]
      { v  &  w  &  x  \\
          &  y  &  z  \\ };
   \ExplSyntaxOn
      \graph_map_edges_inline:Nn
         \l_my_graph
         { \draw (#1) to (#2); }
   \ExplSyntaxOff
\end{tikzpicture}
```



Just to be clear, this library does not, inherently, understand any Ti*k*Z. What it does is help you to analyze the structure of your graph. For example, does it contain a cycle?

```
\ExplSyntaxOn
   \graph_if_cyclic:NTF
      \l_my_graph {Yep} {Nope}
\ExplSyntaxOff
```

Nope

Indeed, there are no cycles in this graph. While we're at it, is vertex `w` reachable from vertex `y`?

```
\ExplSyntaxOn
   \graph_acyclic_if_path_exist:NnnTF
      \l_my_graph {y} {w} {Yep} {Nope}
\ExplSyntaxOff
```

Nope

Quite true. Finally, and most importantly, you can interpret the graph as a dependency graph and list its vertices in topological order:

```
\ExplSyntaxOn
  \clist_new:N \LinearClist
  \graph_map_topological_order_inline:Nn
     \l_my_graph
     { \clist_put_right:Nn
       \LinearClist {\texttt{#1}} }
\ExplSyntaxOff
$ \LinearClist $
```

`v`, `w`, `x`, `y`, `z`

A topological order is not uniquely determined. The important thing is that the constraints imposed by the graph are respected.

## 4 The `pkgloader` package (Part 2)

The `pkgloader` package uses graph vertices to represent LaTeX packages and arrows to encode package ordering rules. At the end, selected packages are loaded in topological order.

Let's take a more detailed look at some of the features of `pkgloader`.

### 4.1 Rules

Each **`\Load`** rule can contain a number of different clauses. We look at them one by one.

It usually contains a *package description*, consisting of a name, a set of options and a minimal version, just like the **`\usepackage`** command:

```
\Load [options] {package-name} [version]
```

It can contain a *condition clause*, indicating when the rule should be applied. This takes the form of a Boolean formula in `expl3` style, in which the atomic propositions are package names:

```
\Load {pkg1} if {pkg2 || pkg3 && !pkg4}
```

This rule would load `pkg1` if either `pkg2` will be loaded too, or if `pkg3` will be loaded but `pkg4` will not. Alternatively, the condition clause can be **`always`**, indicating that the rule should be applied under any conditions. Finally, the keywords **`if loaded`** can be used to apply the rule only if the package named in the package description is requested anyway. This is the default behavior, but the keywords can be included to make it explicit.

There is one exception to the structure described above. Instead of a package description, a rule can contain the **`error`** keyword, followed by a condition clause, to describe conditions that should never occur — usually invalid package combinations:

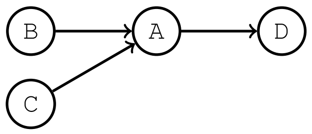
```
\Load error if {pkgX && pkgY}
```

When multiple condition clauses are present in a single rule, their *disjunction* is used. In other words, the rule is applied if *any* of its conditions is satisfied.

A non-error rule may contain an *order clause*, to ensure that the package described by the rule is loaded in a specific order with regard to other packages:

```
\Load {A} after {B,C} before {D}
```

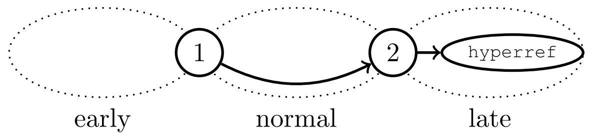
This rule ensures that if package `A` is ever loaded, it is never loaded before `B` or `C`, and never after `D`. This is where the graph representation comes in. The above rule would yield a graph like this:



This can take care of specific known package ordering conflicts. But some packages should, as a rule of thumb, be loaded before all other packages, or after all others, unless specified otherwise. A typical example is the `hyperref` package, which should almost always be loaded late in the run. For this, the **`early`** and **`late`** clauses may be used:

```
\Load {hyperref} late
```

The **`early`** and **`late`** clauses work by ordering the package relative to one of two placeholder nodes:



These two nodes are always present in the graph. Ordering a package **`early`** is intuitively the same as ordering it '**`before {1}`**'. And ordering it **`late`** is the same as ordering it '**`after {2}`**'. All packages that are, after considering all rules, not (indirectly) ordered '**`before {1}`**' or '**`after {2}`**' are automatically ordered '**`after {1} before {2}`**'. A rule can have any number of order clauses, and all are taken into account when one of the conditions of the rule is satisfied.

Finally, a rule can be annotated with a *reason*, explaining why it was created:

```
\Load {comicsans} always
    because {that font is awesome!}
```

This text does not have any effect on the behavior of the rule. It is meant for human consumption, though should not be formatted in any way. It should be semantically and grammatically correct when following the words "This rule was created because . . . ". It can also be used for citing relevant sources. It is used in certain `pkgloader` error messages and may eventually be used to generate documentation.

## 4.2 Rulesets

You may be wondering: who makes up these rules?

Short answer: Anyone. Rules can be placed directly inside the `pkgloader` area, but they can also be bundled in a `.sty` file. By default, `pkgloader` loads a recommended set of rules, allowing the average user to get started without any hassle. But this behavior can be overwritten using package options:

```
\RequirePackage[recommended=false,
                my-better-rules]
               {pkgloader}
   ...
\LoadPackagesNow
```

This means: the `pkgloader-recommended.sty` file, which is usually preloaded by default, should *not* be loaded for this document. Instead, load the `pkgloader-my-better-rules.sty` file.

Take note of the following: every ruleset should be bundled in a `pkgloader-`⟨*something*⟩`.sty` file, and can then be loaded by specifying ⟨*something*⟩ as a package option.

So basically, any user can create rules for their own documents, or distribute custom rulesets, e.g., through CTAN. But primarily, I expect two groups of people to author `pkgloader` rules:

**The LaTeX community:** The `recommended` ruleset would, ideally, be populated further through the efforts of anyone who diagnoses and solves package conflicts. Perhaps through websites like `tex.stackexchange.com`, or by filing issues or pull-requests to the `pkgloader` Github page.

**Package authors:** `pkgloader` will eventually be directly usable for package authors just as for document authors, to include their own rules from right inside their packages. Rather than manually scanning for and fixing potential conflicts, they could leverage `pkgloader`, as in:

```
\RequirePackage{pkgloader}
   \Load me before {some-pkg}
   \Load me after {some-other-pkg}
\ProcessRulesNow
```

It may be possible to apply such rules in the same LaTeX run in which they are encountered. But if not, the package manager will know what to do in the next run through the use of auxiliary files. This functionality has not yet been implemented.

## 4.3 Error messages

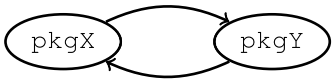
There are two types of error messages that may be generated by `pkgloader`.

The first type of error message happens when an **`error`** rule is triggered. It looks like this:

```
A combination of packages fitting
the following condition was requested:
    ⟨condition⟩
This is an error because ⟨reason⟩.
```

The second type of error message is a bit more interesting. Since rules can effectively come from any source, it is possible to apply rules that contradict each other. To give an (unrealistic) example:

```
\Load {pkgX} always before {pkgY}
         because {pkgX is better}
\Load {pkgY} always before {pkgX}
         because {pkgY is better}
```



A potential circular ordering is not necessarily a problem, so long as both rules are never applied in the same run. But in this exact example, the following error message will be generated:

```
There is a cycle in the requested
package loading order:
          pkgX
   --1--> pkgY
   --2--> pkgX
The circular reasoning is as follows:
(1) 'pkgX' is to be loaded before
    'pkgY' because pkgX is better.
(2) 'pkgY' is to be loaded before
    'pkgX' because pkgY is better.
```

Whenever this happens, the user may want to reconsider one of their included rulesets, or file a bug-report to the responsible party — especially if the circularity comes from the `recommended` ruleset.

### 4.4 Options and versions

The package manager need not be confined to playing with the package loading order. While intercepting package loading requests, it will be able to accumulate package options and versions as well, and then combine them in a multitude of flexible ways. Possible ways of combining option-lists include:

- Concatenate all option-lists for the same package into a single list, in any arbitrary order.
- Interpret `key=value` options, and generate an error message when two different instances call for two different values for the same key.
- Provide a tailor-made function for combining option lists for a specific package.

As for package versions, it would make sense to take the 'maximum' version-string that is encountered, and use that to load the actual package.

As of this writing, neither of these features has been implemented.

## 5 Conclusion

I hope this article and the packages described therein have been useful and/or inspiring. And I hope to have convinced you that the idea of a LaTeX package manager is worth pursuing.

Both packages are available on CTAN:

`www.ctan.org/pkg/pkgloader`
`www.ctan.org/pkg/lt3graph`

I love feedback, and I love questions. I can be reached through the e-mail address and website specified in the signature block below the references. Any kind of feedback or patches regarding one of the packages should go through their Github pages:

`github.com/mhelvens/latex-pkgloader`
`github.com/mhelvens/latex-lt3graph`

Happy TeXing!

### References

[1] David Carlisle. xii.tex, December 1998. http://www.ctan.org/pkg/xii.

[2] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. *SIGPLAN Notices*, 46(2):13–22, February 2011. Proceedings of GPCE'10, October 10–13, 2010, Eindhoven, The Netherlands. http://doi.acm.org/10.1145/1942788.1868298.

[3] Freek Dijkstra. LaTeX package conflicts, June 2012. http://www.macfreek.nl/memory/LaTeX_package_conflicts.

[4] Donald E. Knuth. *The TeXbook*. Addison-Wesley, Reading, MA, USA, 1986.

[5] Seamus. What are category codes?, April 2011. http://tex.stackexchange.com/questions/16410/what-are-category-codes.

[6] Wikipedia. Domain-specific language, 2014. http://en.wikipedia.org/wiki/Domain-specific_language.

[7] Wikipedia. Monkey patch, January 2014. http://en.wikipedia.org/w/index.php?title=Monkey_patch&oldid=586056263.

[8] Wikipedia. Turing completeness, 2014. http://en.wikipedia.org/wiki/Turing_completeness.

◇ Michiel Helvensteijn
Leiden University,
Niels Bohrweg 1,
2333 CA, Leiden,
the Netherlands
mhelvens+latex (at) gmail dot com
http://mhelvens.net

# An overview of Pandoc

Massimiliano Dominici

## Abstract

This paper is a short overview of Pandoc, a utility for the conversion of Markdown-formatted texts to many output formats, including LaTeX and HTML.

## 1 Introduction

Pandoc is software, written in Haskell, whose aim is to facilitate conversion between some lightweight markup languages and the most widespread 'final' document formats.[1] On the program's website [3], Pandoc is described as a sort of 'swiss army knife' for converting between different formats, and in fact it is able to read simple files written in LaTeX or HTML; but it is of lesser use when trying to translate LaTeX documents with non-trivial constructs such as commands defined by the user or by a dedicated package.

Pandoc shows its real utility, in my opinion, when what is needed is to obtain several output formats from a single source, as in the case of a document distributed online (HTML), in print form (PDF via LaTeX) and for viewing on tablets or ebook readers (EPUB). In such cases one may find that writing the document in a rich format (e.g. LaTeX) and converting later to other markup languages often poses significant problems because of the different 'philosophies' that underlie each language. It is advisable, instead, to choose as a starting point a language that is 'neutral' by design. A good candidate for this role is a lightweight markup language, and in particular Markdown, of which Pandoc is an excellent interpreter.

In this article we will briefly discuss the concept of a 'lightweight markup language' with particular reference to Markdown (§ 2), and then we will review Pandoc in more details (§ 3) before drawing our conclusions (§ 5).

## 2 Lightweight markup languages: Markdown

Before getting to the heart of the matter, it is advisable to say a few words about lightweight markup languages (LML) in general. They are designed with the explicit goal of minimizing the impact of the markup instructions within the document, with a particular emphasis on the *readability* of the text by a *human being*, even when the latter does not know the (few) conventions that the program follows in order to format the document.

These languages are mainly used in two fields: documentation of code (reStructuredText, AsciiDoc, etc.) and management of contents for the web (Markdown, Textile, etc.). In the case of code documentation, the use of an LML is a good choice, because the documentation is interspersed in the code itself, so it should be easy to read by a developer perusing the code; but at the same time it should be able to be converted to presentation formats (PDF and HTML, traditionally, but today many IDEs include some form of visualization for the internal documentation). In the case of web content, the emphasis is placed on the ease of writing for the user. Many content management systems already provide plugins for one or more of those languages and the same is true for static site generators[2] that are usually built around one of them and often provide support for others. The various wiki dialects can be considered another instance of LML.

The actual 'lightness' of an LML depends greatly on its ultimate purpose. In general, an LML conceived for code documentation will be more complex and less readable than one conceived for web content management, which in turn will often not be capable of general semantic markup. A paradigmatic example of this second category is Markdown that, in its original version, stays rigorously close to the minimalistic approach of the first LMLs. The following citation from its author, John Gruber, explains his intentions in designing Markdown:

> Markdown is intended to be as easy-to-read and easy-to-write as is feasible. Readability, however, is emphasized above all else. A Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions.[3]

The only output format targeted by the reference implementation of Markdown is HTML; indeed, Markdown also allows raw HTML code. Gruber has

Translation by the author from his original in *ArsTeXnica* #15, April 2013, "Una panoramica su Pandoc", pp. 31–38.

[1] Strictly speaking, LaTeX isn't a 'final' document format in the same way PDF, ODF, DOC, EPUB, etc. are. But, from the point of view of a Pandoc user, LaTeX is a 'final' — or intermediate, at least — product.

[2] Static site generators are a category of programs that build a website in HTML starting from source files written in a different format. The HTML pages are produced beforehand, usually on a local computer, and then loaded on the server. Websites built this way share a great resemblance with old websites written directly in HTML, but unlike those, in the building process it is possible to use templates, share metadata across pages, and create structure and content programmatically. Static site generators constitute an alternative to the more popular dynamic server applications.

[3] [1], `http://daringfireball.net/projects/markdown/syntax#philosophy`. A significant contribution to the design of Markdown was made by Aaron Swartz.

**Table 1**: Markdown syntax: inline elements.

| Element | Markdown | LaTeX | HTML |
|---|---|---|---|
| Links | [link](http://example.net) | \href{link}{%<br>  http://example.net} | <a href="http://example.net/"><br>    link</a> |
| Emphasis | _emphasis_<br>*emphasis* | \emph{emphasis}<br>\emph{emphasis} | <em>emphasis</em><br><em>emphasis</em> |
| Strong emphasis | __strong__<br>**strong** | \textbf{strong}<br>\textbf{strong} | <strong>strong</strong><br><strong>strong</strong> |
| Verbatim | ‘printf()‘ | \verb\|printf()\| | <code>printf()</code> |
| Images | ![Alt](/path/to/img.jpg) | \includegraphics{img} | <img src="/path/to/img.jpg"<br>  alt="Alt" /> |

**Table 2**: Markdown syntax: block elements.

| Element | Markdown | LaTeX | HTML |
|---|---|---|---|
| Sections | # Title #<br>## Title ##<br>... | \section{Title}<br>\subsection{Title}<br>... | <h1>Title</h1><br><h2>Title</h2><br>... |
| Quotation | > This paragraph<br>> will show<br>> as quote. | \begin{quote}<br>This paragraph<br>will show<br>as quote.<br>\end{quote} | <blockquote><p><br>    This paragraph<br>    will show<br>    as quote.<br>  </p></blockquote> |
| Itemize | * First item<br>* Second item<br>* Third item | \begin{itemize}<br>\item First item<br>\item Second item<br>\item Third item<br>\end{itemize} | <ul><br><li>First item</li><br><li>Second item</li><br><li>Third item</li><br></ul> |
| Enumeration | 1. First item<br>2. Second item<br>3. Third item | \begin{enumerate}<br>\item First item<br>\item Second item<br>\item Third item<br>\end{enumerate} | <ol><br><li>First item</li><br><li>Second item</li><br><li>Third item</li><br></ol> |
| Verbatim | Text paragraph.<br><br>    grep -i ’\$’ <file | Text paragraph.<br><br>\begin{verbatim}<br>grep -i ’\$’ <file<br>\end{verbatim} | <p>Text paragraph.</p><br><br><pre><code><br>  grep -i ’\$’ &lt;file<br></code></pre> |

always adhered to these initial premises and has consistently refused to extend the language beyond the original specifications. This stance has caused a proliferation of variants, so that every single implementation constitutes an ‘enhanced’ version. Famous websites like GitHub, reddit and Stack Overflow, all support their own Markdown flavour; and the same is true for conversion programs like MultiMarkdown or Pandoc itself, which also introduce new output formats. It’s not necessary, here, to examine the details of the different flavours; the reader can get an idea of the basic formatting rules from tables 1 and 2. Of course, in the reference implementation there is no LaTeX output, so I have provided the most logical translation. In the following sections we will see how Pandoc works in practice.

## 3 An overview of Pandoc

As mentioned in the introduction, Pandoc is primarily a Markdown interpreter with several output formats: HTML, LaTeX, ConTeXt, DocBook, ODF, OOXML, other LMLs such as AsciiDoc, reStructuredText and Textile (a complete list can be found in [3]). Pandoc can also convert, with severe restrictions, a source file in LaTeX, HTML, DocBook, Textile or reStructuredText to one of the aforementioned output formats. Moreover it extends the syntax of Markdown, introducing new elements and providing customization for the elements already available in the reference implementation.

## 3.1 Markdown syntax extensions

Markdown provides, by design, a very limited set of elements. Tables, footnotes, formulas, and bibliographic references have no specific markup in Markdown. The author's intent is that all markup exceeding the limits of the language should be expressed in HTML. Pandoc maintains this approach (and, for LaTeX or ConTeXt output, allows the use of raw TeX code) but makes it unnecessary, since it introduces many extensions, giving the user proper markup for each of the elements mentioned above. In the following paragraphs we'll take a look at these extensions.

**Metadata** Metadata for title, author and date can be included at the beginning of the file, in a text block, each preceded by the character `%`, as in the following example.

```
% Title
% First Author; Second Author
% 17/02/2013
```

The content for any of these elements can be omitted, but then the respective line must be left blank (unless it is the last element, i.e. the date).

```
% Title
%
% 17/02/2013

%
% First Author; Second Author
% 17/02/2013

% Title
% First Author; Second Author
```

Since version 1.12 metadata support has been substantially extended. Now Pandoc accepts multiple metadata blocks in YAML format, delimited by a line of three hyphens (`---`) at the top and a line of three hyphens (`---`) or three dots (`...`) at the bottom.[4] This gives the user a high level of flexibility in setting and using variables for templates (see section 3.1).

YAML structures metadata in arrays, thus allowing for a finer granularity. The user may specify in his source file the following code:

```
---
author:
- name: First Author
- affiliation: First Affiliation
- name: Second Author
- affiliation: Second Affiliation
---
```

[4] YAML Ain't Markup Language, http://www.yaml.org/.

and then, in the template

```
$for(author)$
$if(author.name)$
$author.name$
$if(author.affiliation)$ ($author.affiliation$)
$endif$
$else$
$author$
$endif$
$endfor$
```

to get a list of authors with (if present) affiliations.

As we will see in section 3.1, a YAML block can also be used to build a bibliographic database.

**Footnotes** Since the main purpose of Markdown and its derivatives is readability, the mark and the text of a footnote should usually be split. It is recommended to write the footnote text just below the paragraph containing the mark, but this is not strictly required: the footnotes could be collected at the beginning or at the end of the document, for instance. The mark is an arbitrary label enclosed in the following characters: `[^...]`. The same label, followed by '`:`' must precede the footnote text.

When the footnote text is short, it is possible to write it directly inside the text, without the label. The output from all the footnotes is collected at the end of the document, numbered sequentially. Here is an example of the input:

```
Paragraph containing[^longnote] a
footnote too long to be written directly
inside the text.

[^longnote]: A footnote too long to be
written inside the text without causing
confusion.

New paragraph.^[A short note.]
```

**Tables** Again, syntax for tables is based on considerations of readability of the source. The alignment of the cells composing the table is immediately visible in the alignment of the text with respect to the dashed line that divides the header from the rest of the table; this line must *always* be present, even when the header is void.[5] When the table includes cells with more than one line of text, it is mandatory

[5] In fact, it is the header, if present, or the first line of the table that sets the alignment of the columns. Aligning the remaining cells is not needed, but it is recommended as an aid for the reader.

to enclose it between two dashed lines. In this case the width of each column in the source file is used to compute the width of the equivalent column in the output table. Multicolumn or multirow cells are not supported. The caption can precede or follow the table; it is introduced by the markup ':' (alternatively: Table:) and must be divided from the table itself by a blank line:

```
-----------------------------------------
        Right    Centered    Left
------------- -------------- -------------
         Text      Text      Text
      aligned    aligned     aligned
        right     center     left

     New cell    New cell    New cell
-----------------------------------------

Table: Alignment
```

There's an alternative syntax to specify the alignment of the individual columns: divide columns with the character '|' and use the character ':' in the dashed line below the header to specify, through its placement, the column's alignment, as shown in the following example:

```
Right    | Centered    | Left
--------:|:-----------:|:-------------
Text     | Text        | Text
aligned  | aligned     | aligned
right    | center      | left
         |             |
New cell | New cell    | New cell

: Alignment by ':'
```

In the examples above, the cells cannot contain 'vertical' material (multiple paragraphs, verbatim blocks, lists). 'Grid' tables (see the example below) allow this, at the cost of not being to specify the column alignments.

```
+-------------+-------------+----------------+
| Text        | Lists       | Code           |
+=============+=============+================+
| Paragraph.  | * Item 1    | ~~~            |
|             | * Item 2    | \def\PD{%      |
| Paragraph.  |             |  \emph{Pandoc} |
|             | * Item 3    | ~~~            |
+-------------+-------------+----------------+
| New cell    | New cell    | New cell       |
+-------------+-------------+----------------+
```

**Figures** As shown in table 1, Markdown allows for the use of inline images with the following syntax:

`![Alternative text](/path/image)`

where 'Alternative text' is the description that HTML uses when the image cannot be viewed. Pandoc adds to that one more feature: if the image is divided by blank lines from the remaining text, it will be interpreted as a floating object with its own caption taken from the 'Alternative text'.

**Listings** In standard Markdown, verbatim text is marked by being indented by four spaces or one tab. To that, Pandoc adds the ability to specify identifiers, classes and attributes for a given block of 'verbatim' material. Pandoc will treat them in different ways, depending on the output format and the command line options; in some circumstances, they will simply be ignored. To achieve this, Pandoc introduces an alternative syntax for listings of code: instead of indented blocks, they are represented by blocks delimited by sequences of three or more tildes (~~~) or backticks (```); identifiers, classes and attributes must follow that initial 'rule', enclosed in braces. In the following example[6] we can see a listing of Python code with, in this order: an identifier, the class that marks it as Python code, another class that specifies line numbering and an attribute that marks the starting point of the numbering.

````
~~~ {#bank .python .numberLines startFrom="5"}
class BankAccount(object):
    def __init__(self, initial_balance=0):
        self.balance = initial_balance
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
    def overdrawn(self):
        return self.balance < 0
my_account = BankAccount(15)
my_account.withdraw(5)
print my_account.balance
~~~
````

It is possible to use identifiers, class and attributes for inline code, too:

```
The return value of the `printf`{.C} function
is of type `int`.
```

By default, Pandoc uses a simple verbatim environment for code that doesn't need highlighting and the Highlighting environment, defined in the template's preamble (see 3.1) and based on Verbatim from fancyvrb, when highlighting is needed. If the option --listings is given on the command line, Pandoc uses the lstlistings environment from listings every time a code block is encountered.

**Formulas** Pandoc supports mathematical formulas quite well, using the usual TeX syntax. Expressions enclosed in dollar signs will be interpreted as inline formulas; expressions in double dollar signs

[6] From http://wiki.python.org/moin/SimplePrograms.

will be interpreted as displayed formulas. This is all comfortably familiar to a TeX user.

The way these expressions will be treated depends on the output format. For TeX (LaTeX/ConTeXt) output, the expressions are passed without modifications, except for the substitution of the delimiters for display math: `\[...\]` instead of `$$...$$`. When HTML (or similar) output is required, the behavior is controlled by command line options. Without options, Pandoc will try to render the formulas by means of Unicode characters. Other options allow for the use of some of the most common JavaScript libraries for visualizing math on the web: MathJax, LaTeXMathML and jsMath. It is also possible, always by means of a command line switch, to render formulas as images or to encode them as MathML.[7]

Pandoc is also able to parse simple macros and expand them in output formats different from the supported TeX dialects. This feature, though, is only available in the context of math rendering.

**Citations** Pandoc can build a bibliography (and manage citations inside the text) using a database in any of several common formats (BibTeX, EndNote, ISI, etc.). The database file must *always* be specified as the argument of the option `--bibliography`. Without other options on the command line, Pandoc will include citations and bibliographic entries as plain text, formatted following the bibliographic style 'Chicago author–date'. The user may specify a different style by means of the option `--csl`, whose argument is the name of a CSL style file[8] and may also specify that the bibliographic apparatus will be managed by natbib or biblatex. In this case Pandoc will not include in the LaTeX output citations and entries in extended form, but only the required commands. The options to get this behavior are, respectively, `--natbib` and `--biblatex`.

The user must type citations in the form `[@key1; @key2;...]` or `@key1` if the citation should not be enclosed in round brackets. A dash preceding the label suppresses the author's name (when supported by the citation format). Bibliographic references are always placed at the end of the document.

[7] The web pages for these different rendering engines for math on the web are `http://www.mathjax.org`, `http://math.etsu.edu/LaTeXMathML`, `http://www.math.union.edu/~dpvc/jsmath` and `http://www.w3.org/Math`.

[8] CSL (`http://citationstyles.org`), *Citation Style Language*, is an open format, XML-based, language to describe the formatting of citations and bibliographies. It is used in several citation managers, such as Zotero, Mendeley, and Papers. A detailed list of the available styles can be found in `http://zotero.org/styles`.

```
---
references:
- author:
    family: Gruber
    given:
    - John
  id: gruber13:_markd
  issued:
    year: 2013
  title: Markdown
  type: no-type
  publisher: <http://daringfireball.net/
              projects/markdown/>
- volume: 32
  page: 272-277
  container-title: TUGboat
  author:
    family: Kielhorn
    given:
    - Axel
  id: kielhorn11:_multi
  issued:
    year: 2011
  title: Multi-target publishing
  type: article-journal
  issue: 3
...
```

**Figure 1**: A YAML bibliographic database (line break in the url is editorial).

Since version 1.12 native support for citations has been split from the core functions of Pandoc. In order to activate this feature, one must now use an external filter (`--filter pandoc-citeproc`, to be installed separately).[9]

A new feature is that bibliographic databases can now be built using the `references` field inside a YAML block. Finding the correct encoding for a YAML bibliographic database can be a little tricky, so it is recommended, if possible, to convert from an existing database in one of the formats recognized by Pandoc (among them BibTeX), using the `biblio2yaml` utility, provided together with the `pandoc-citeproc` filter. The YAML code for the first two items in this article's bibliography, created by converting the `.bib` file, is shown in figure 1.

A YAML field can be used also for specifying the CSL style for citations (`csl` field), or the external bibliography file, if required (`bibliography` field).

**Raw code (HTML or TeX)** All implementations of Markdown, of whichever flavour, allow for the use of raw HTML code, written without modifications in the output, as we mentioned in section 2. Pandoc extends this feature, allowing for the use of TeX raw code, too. Of course, this works only for LaTeX/ConTeXt output.

[9] The filter is not needed when using natbib or biblatex directly instead of the native support.

```
  1  \documentclass$if(fontsize)$[$fontsize$]$endif$
  2    {article}
  3  \usepackage{amssymb,amsmath}
  4  \usepackage{ifxetex}
  5  \ifxetex
  6    \usepackage{fontspec,xltxtra,xunicode}
  7    \defaultfontfeatures{Mapping=tex-text,
  8                         Scale=MatchLowercase}
  9  \else
 10     \usepackage[utf8]{inputenc}
 11  \fi
 12  $if(natbib)$
 13  \usepackage{natbib}
 14  \bibliographystyle{plainnat}
 15  $endif$
 16  $if(biblatex)$
 17  \usepackage{biblatex}
 18  $if(biblio-files)$
 19  \bibliography{$biblio-files$}
 20  $endif$
 21  $endif$

     ...

113  $body$
114
115  $if(natbib)$
116  $if(biblio-files)$
117  $if(biblio-title)$
118  $if(book-class)$
119  \renewcommand\bibname{$biblio-title$}
120  $else$
121  \renewcommand\refname{$biblio-title$}
122  $endif$
123  $endif$
124  \bibliography{$biblio-files$}
125  $endif$
126  $endif$
127  $if(biblatex)$
128  \printbibliography
129    $if(biblio-title)$[title=$biblio-title$]$endif$
130  $endif$
131  $for(include-after)$
132  $include-after$
133  $endfor$
134  \end{document}
```

**Figure 2**: Fragments of the default Pandoc v1.11 template for LaTeX.

**Templates** One of the most interesting features of Pandoc is the use of customized templates for the different output formats. For HTML-derived and TeX-derived output formats this can be achieved in two ways. First of all, the user may generate only the document 'body' and then include it inside a 'master' (for TeX output, with `\input` or `\include`). In this way, an *ad hoc* preamble can be built beforehand. This is in fact the default behaviour for Pandoc; to obtain a complete document, including a preamble, the command line option `--standalone` (or its equivalent `-s`) is used.

It is also possible to build more flexible templates, useful for different projects with different features, providing for a moderate level of customization. As the reader can see in figure 2, a template is substantially a file in the desired output format (in this case LaTeX) interspersed with variables and control flow statements introduced by a dollar sign. The expressions will be evaluated during the compilation and replaced by the resulting text. For instance, at line 113 of the listing in figure 2 we find the expression `$body$`, which will be replaced by the document body. Above, at lines 12–21, we can find the sequence of commands that will include in the final output all the resources needed to generate a bibliography by means of natbib or biblatex. This code will be activated only if the user has specified either `--natbib` or `--biblatex` on the command line. The code to print the bibliography can be found at the end of the listing, at lines 124–130.

In this way it is possible to define all the desired variables and the respective compiler options. The user can thus change the default template to specify, e.g., among the options that may be passed to the class, not only the body font, but a generic string containing more options.[10] We would replace the first line in the listing of figure 2 with the following:

```
\documentclass$if(clsopts)$[$clsopts$]$endif$
```

Then we can compile with the following options:

```
pandoc -s -t latex --template=mydefault \
  -V clsopts=a4paper,12pt -o test.tex test.md
```

given that we saved the modified template in the current directory by the name `mydefault.latex`.

Since version 1.12, 'variables' can be replaced by YAML 'metadata', specified either inside the source file or on the command line using the `-M` option.

## 4 Problems and limitations

We've seen many nice features of Pandoc. Not surprisingly, Pandoc also has some limitations and shortcomings. Some of these shortcomings are tied to the particular LML used by Pandoc. For instance, Markdown doesn't allow semantic markup.[11] This kind of limitation can be addressed using an additional level

[10] This can be also be achieved via the variable `$fontsize`. (Since Pandoc 1.12, the default LaTeX template includes separate variables for body font size, paper size and language, and a generic `$classoption` variable for other parameters.)

[11] This is not an issue necessarily pertinent to all LMLs since some of them provide methods to define objects that behave like LaTeX macros, either through pre- or post-processing (`txt2tags`) or by taking advantage of conceptually close structures (the 'class' of a `span` in HTML, in Textile). In any case,

of abstraction, using preprocessors like gpp or m4, as illustrated by Aditya Mahajan in [4]. Of course, this clashes with the initial purpose of readability and introduces further complexity, though the use of m4 need not significantly increase the amount of extra markup.

Other problems, though, unexpectedly arise in the process of conversion to LaTeX output. For instance, the cross reference mechanism is calibrated to HTML and shows all its shortcomings with regard to the LaTeX output. The cross reference is in fact generated by means of a hypertext anchor and not by the normal use of \label and \ref. As a typical example, let's consider a labelled section referenced later in the text, as in the following:

```
## Basic elements ## {#basic}
[...]
As we have explained in
[Basic elements](#basic)
```

we get this result:

```
\hyperdef{}{basic}{%
  \subsection{Basic elements}\label{basic}
}
[...]
As we have explained in
\hyperref[basic]{Basic elements}
```

which is not exactly what a LaTeX user would expect ... Of course one could directly use \label and \ref, but they will be ignored in all non-TeX output formats. Or, we could use a preprocessor to get two different intermediate source files, one for HTML and for LaTeX (and maybe a third for ODF/OOXML, etc.), but by now the original point of using Pandoc is being lost.

Formulas, too, may cause some problems. Pandoc recognizes only inline and display expressions. The latter are always translated as displaymath environments. It is not possible to specify a different kind of environment (equation, gather, etc.) unless one of the workarounds discussed above is employed, with all the consequent drawbacks also noted.

It should be stressed, however, that Pandoc is a program in active development and that several features present in the current version were not available a short time ago. So it is certainly possible that all or some of the shortcomings that a LaTeX user finds in the current version of Pandoc will be addressed in the near or mid-term. It is also possible, to some extent, to extend or modify Pandoc's behaviour by means of scripts, as noted at http://johnmacfarlane.net/pandoc/scripting.html. One major drawback, until recently, was the mandatory use of Haskell for such scripts (a drawback for me, at least ...). The current version also allows Python, thus making easier the task of creating such scripts.[12]

though, the philosophy behind LMLs doesn't support such markup methods.

## 5 Conclusion

To conclude this overview, I consider Pandoc to be the best choice for a project requiring multiple output formats. The use of a 'neutral' language in the source file makes it easier to avoid the quirks of a specific language and the related problems of translation to other languages. For a LaTeX user in particular, being able to type mathematical expressions 'as in LaTeX' and to use a BibTeX database for bibliographic references are also two strong points.

One should not expect to find in Pandoc an easy solution for every difficulty. Limitations of LMLs in general, and some flaws specific to the program, entail the need for workarounds, making the process less immediate. This doesn't change the fact that, if the user is aware of such limitations and the project can bear them, Pandoc makes obtaining multiple output formats from a single source extremely easy.

## References

[1] John Gruber. Markdown. http://daringfireball.net/projects/markdown/, 2013.

[2] Axel Kielhorn. Multi-target publishing. *TUGboat*, 32(3):272–277, 2011. http://tug.org/TUGboat/tb32-3/tb102kielhorn.pdf.

[3] John MacFarlane. *Pandoc*: a universal document converter. http://johnmacfarlane.net/pandoc/, 2013.

[4] Aditya Mahajan. How I stopped worrying and started using Markdown like TeX. http://randomdeterminism.wordpress.com/2012/06/01/how-i-stopped-worring-and-started-using-markdown-like-tex/, 2012.

[5] Wikipedia. Lightweight markup language. http://en.wikipedia.org/wiki/Lightweight_markup_language, 2013.

◇ Massimiliano Dominici
Pisa, Italy
mlgdominici (at) gmail dot com

[12] Another option is writing one's own custom 'writer' in Lua. A writer is essentially a program that translates the data structure, collected by the 'reader', in the format specified by the user. Having Lua installed on the system is not required, since a Lua interpreter is embedded in Pandoc. See http://johnmacfarlane.net/pandoc/README.html#custom-writers.

# Numerical methods with LuaLaTeX

Juan I. Montijano, Mario Pérez, Luis Rández and Juan Luis Varona

## Abstract

An extension of TeX known as LuaTeX has been in development for the past few years. Its purpose is to allow TeX to execute scripts written in the general purpose programming language Lua. There is also LuaLaTeX, which is the corresponding extension for LaTeX.

In this paper, we show how LuaLaTeX can be used to perform tasks that require a large amount of mathematical computation. With LuaLaTeX instead of LaTeX, we achieve important improvements: since Lua is a general purpose language, rendering documents that include evaluation of mathematical algorithms is much easier, and generating the PDF file becomes much faster.

## Introduction

TeX (and LaTeX) is a document markup language used to typeset beautiful papers and books. Although it can also do programming commands such as conditional execution, it is not a general purpose programming language. Thus there are many tasks that are easily done with other programming languages, but are very complicated or very slow when done with TeX. Due to this limitation, auxiliary programs have been developed to assist TeX with common tasks related to document preparation. For instance, bibtex or biber to build bibliographies, and makeindex or xindy to generate indexes. In both cases, sorting a list alphabetically is a relatively simple task for most programming languages, but it is very complicated to do with TeX, hence the desire for auxiliary applications.

Another shortcoming of TeX is the computation of mathematical expressions. One of the most common uses of TeX is to compose mathematical formulas, and it does this extremely well. However TeX is not good at computing mathematics. For instance, TeX itself does not have built-in functions to compute a square root or a sine. Although it is possible to compute mathematical functions with the help of auxiliary packages written in TeX, internally these packages must compute functions using only addition, subtraction, multiplication and division operations — and a very large number of them. This is difficult to program (for package developers) and slow in execution.

To address the need to do more complex functions within TeX, an extension of TeX called LuaTeX was undertaken a few years ago. (The leaders of the project and main developers are Taco Hoekwater, Hartmut Henkel and Hans Hagen.) The idea was to enhance TeX with a previously existing general purpose programming language. After careful evaluation of possible candidates, the language chosen was Lua (see `http://www.lua.org`), a powerful, fast, lightweight, embeddable scripting language that has, of course, a free software license suitable to be used with TeX. Moreover, Lua is easy to learn and use, and anyone with basic programming skills can use it without difficulty. (Many examples of Lua code can be found later in this article, and also, for example, at `http://rosettacode.org/wiki/Category:Lua`, and `http://lua-users.org/`.)

LuaTeX is not TeX, but an extension of TeX, in the same way that pdfTeX or XeTeX are extensions. In fact, LuaTeX includes pdfTeX (it is an extension of pdfTeX, and offers backward compatibility), and also has many of the features of XeTeX.

LuaTeX is still in a beta stage, but the current versions are usable (the first public beta was launched in 2007, and when this paper was written in January 2013, the release used was version 0.74).

It has many new features useful for typographic composition; and examples can be seen at the project web site `http://www.luatex.org`, and some papers using development versions have been published in *TUGboat*, among them [3, 2, 4, 5, 7, 11]. Most of those articles are devoted to the internals and are very technical, only for true TeX wizards; we do not deal with this in this paper. Instead, our goal is to show how the mathematical power of the embedded language Lua can be used in LuaTeX. Of course, when we build LaTeX over LuaTeX, we get so-called LuaLaTeX, which will be familiar to regular LaTeX users.

All the examples in this paper are done with LuaLaTeX. It is important to note that the current version of LuaTeX is not meant for production and beta users are warned of possible future changes in the syntax. However, the examples in this article use only a few general Lua-specific commands, so it is likely these examples will continue to work with future versions.

To process a LuaLaTeX document we perform the following steps: First, we must compile with LuaLaTeX, not with LaTeX; how to do this depends on the editor being used. Second, we must load the package luacode with `\usepackage{luacode}`. Then, inside LuaLaTeX, we can jump into Lua mode with the command `\directlua`; moreover, we can define Lua routines in a `\begin{luacode}`...`\end{luacode}` environment (also `{luacode*}` instead of `{luacode}` can be used); the precise syntax can be found in the manual “The luacode package” (by Manuel Pégourié-Gonnard) [10]. In the examples, we do not explain all the details of the code; they are left to the reader’s intuition.

In this paper we present four examples. The first is very simple: the computation of a trigonometric table. In the other examples we use the LaTeX packages tikz and pgfplots to show Lua’s ability to produce graphical output. Some mathematical skill may be necessary to fully understand the examples, but the reader can nevertheless see how Lua is able to manage the computation-intensive job. In any case, we do not explore the more complex possibilities, which involve writing Lua programs that load existing Lua modules or libraries to perform a wide range of functions and specialized tasks.

## 1 First example: a trigonometric table

To show how to use Lua, let us begin with a simple but complete example. Observe the following document, which embeds some Lua source code. Typesetting it with LuaLaTeX, we get the trigonometric table shown in Figure 1.

```
\documentclass{article}
\usepackage{luacode}

\begin{luacode*}
  function trigtable ()
    for t=0, 45, 3 do
      x=math.rad(t)
      tex.print(string.format(
      '%2d$^{\\circ}$ & %1.5f & %1.5f & %1.5f '
        .. '& %1.5f \\\\',
      t, x, math.sin(x), math.cos(x),
        math.tan(x)))
    end
  end
\end{luacode*}
\newcommand{\trigtable}
  {\luadirect{trigtable()}}

\begin{document}
\begin{tabular}{rcccc}
  \hline
  & $x$ & $\sin(x)$ & $\cos(x)$ & $\tan(x)$ \\
  \hline
  \trigtable
  \hline
\end{tabular}
\end{document}
```

The `luacode*` environment contains a small Lua program with a function named `trigtable` (with no arguments). This function consists of a loop with a variable `t` representing degrees. Lua converts `t` to radians with `x=math.rad(t)`; then, Lua computes the sine, the cosine and the tangent. Inside Lua mode, it “exports” to LaTeX with `tex.print`; note that we escape any backslash by doubling it. Moreover, we have taken into account the following notation to give format to numbers:

- `%2d` indicates that a integer number must be displayed with 2 digits.
- `%1.5f` indicates that a floating point number must be displayed with 1 digit before the decimal point and 5 digits after it.

The LaTeX part has the skeleton of a tabular built with the data exported by Lua.

| | $x$ | $\sin(x)$ | $\cos(x)$ | $\tan(x)$ |
|---|---|---|---|---|
| 0° | 0.00000 | 0.00000 | 1.00000 | 0.00000 |
| 3° | 0.05236 | 0.05234 | 0.99863 | 0.05241 |
| 6° | 0.10472 | 0.10453 | 0.99452 | 0.10510 |
| 9° | 0.15708 | 0.15643 | 0.98769 | 0.15838 |
| 12° | 0.20944 | 0.20791 | 0.97815 | 0.21256 |
| 15° | 0.26180 | 0.25882 | 0.96593 | 0.26795 |
| 18° | 0.31416 | 0.30902 | 0.95106 | 0.32492 |
| 21° | 0.36652 | 0.35837 | 0.93358 | 0.38386 |
| 24° | 0.41888 | 0.40674 | 0.91355 | 0.44523 |
| 27° | 0.47124 | 0.45399 | 0.89101 | 0.50953 |
| 30° | 0.52360 | 0.50000 | 0.86603 | 0.57735 |
| 33° | 0.57596 | 0.54464 | 0.83867 | 0.64941 |
| 36° | 0.62832 | 0.58779 | 0.80902 | 0.72654 |
| 39° | 0.68068 | 0.62932 | 0.77715 | 0.80978 |
| 42° | 0.73304 | 0.66913 | 0.74314 | 0.90040 |
| 45° | 0.78540 | 0.70711 | 0.70711 | 1.00000 |

**Figure 1**: A trigonometric table.

## 2 Second example: Gibbs phenomenon

Now and in what follows, we will use graphics to show the output of some mathematical routines. A very convenient way to do it is by means of the PGF/Ti*k*Z package (Ti*k*Z is a high-level interface to PGF) by Till Tantau (the huge manual [12] of the current version 2.10 has more than 700 pages of documentation and examples); two short introductory papers are [8, 9]. Based on PGF/Ti*k*Z, the package pgfplots (by Christian Feuersänger [1]) has additional facilities to plot mathematical functions like $y = f(x)$ (or a parametric function $x = f(t)$, $y = g(t)$) or visualize data in two or three dimensions. For instance, pgfplots can draw the axis automatically, as usual in any graphic software.

For completeness, let us start showing the syntax of pgfplots by means of a data plot; this is an example extracted from its very complete manual (more than 400 pages in the present version 1.7). After loading `\usepackage{pgfplots}`, the code

```
\begin{tikzpicture}
\begin{axis}[xlabel=Cost, ylabel=Error]
```

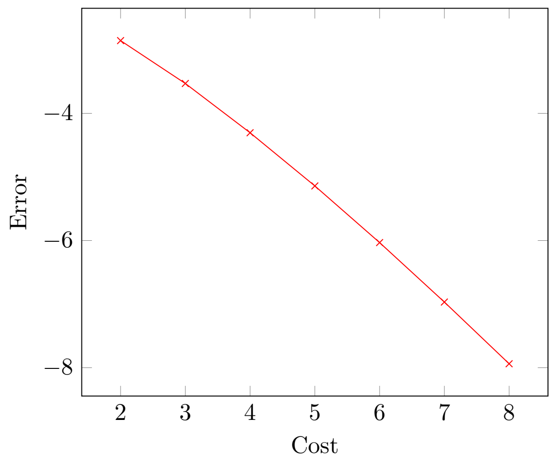


**Figure 2**: Plotting of a data table with pgfplots.

```
 \addplot[color=red,mark=x] coordinates {
  (2,-2.8559703) (3,-3.5301677) (4,-4.3050655)
  (5,-5.1413136) (6,-6.0322865) (7,-6.9675052)
  (8,-7.9377747)
 };
\end{axis}
\end{tikzpicture}
```

generates the plot in Figure 2. Before going on, note that in future versions the packages PGF/Ti*k*Z and pgfplots could, internally, use LuaLaTeX themselves in a way transparent to the user. This would allow extra power, calculating speed, and simplicity, but this is not yet available and we will not worry about it in this paper.

In the next example we consider the Gibbs phenomenon. Using LuaLaTeX, the idea is to compute a data table with Lua (easy to program, powerful and fast in the execution), and plot it with pgfplots.

The Gibbs phenomenon is the peculiar way in which the Fourier series of a piecewise continuously differentiable periodic function behaves at a jump discontinuity, where the $n$-th partial sum of the Fourier series has large oscillations near the jump. It is explained in many harmonic analysis texts, but for the purpose of this paper the reader can refer to [13].

In our case we consider the function $f(x) = (\pi - x)/2$ in the interval $(0, 2\pi)$ extended by periodicity to the whole real line (it has discontinuity jumps at $2j\pi$ for every integer $j$). Its Fourier series is

$$f(x) = \sum_{k=1}^{\infty} \frac{\sin(kx)}{k}.$$

To show the Gibbs phenomenon, we evaluate the partial sum $\sum_{k=1}^{n} \frac{\sin(kx)}{k}$ (for $n = 30$) with Lua to generate a table of data, and we plot it with pgfplots.

In the `.tex` file, we include the following Lua to compute the partial sum (function `partial_sum`) and to export the data with the syntax required by pgfplots (function `print_partial_sum`):

```
\begin{luacode*}
-- Fourier series
function partial_sum(n,x)
    partial = 0;
    for k = 1, n, 1 do
        partial = partial + math.sin(k*x)/k
    end;
    return partial
end

-- Code to write PGFplots data as coordinates
function print_partial_sum(n,xMin,xMax,npoints,
                           option)
    local delta = (xMax-xMin)/(npoints-1)
    local x = xMin
    if option~=[[]] then
        tex.sprint("\\addplot[" .. option
                   .. "] coordinates{")
    else
        tex.sprint("\\addplot coordinates{")
    end
    for i=1, npoints do
        y = partial_sum(n,x)
        tex.sprint("("..x..","..y..")")
        x = x+delta
    end
    tex.sprint("}")
end
\end{luacode*}
```

Then, we also define the command

```
\newcommand\addLUADEDplot[5][]{%
  \directlua{print_partial_sum(#2,#3,#4,#5,
                               [[#1]])}%
}
```

which will be used to call the data from pgfplots. Here, the parameters have the following meaning: `#2` indicates the number of terms to be added ($n = 30$ in our case); the plot will be done in the interval [`#3`, `#4`] (from $x = 0$ to $10\pi$) sampled with `#5` points (to get a very smooth graphic and to show the power of the method we use 1000 points); finally, the optional argument `#1` is used to manage optional arguments in the `\addplot` environment (for instance color [grayscaled for *TUGboat*], width of the line, ...).

Now, the plot is generated by

```
\pgfplotsset{width=.9\hsize}
\begin{tikzpicture}\small
\begin{axis}[
  xmin=-0.2, xmax=31.6,
  ymin=-1.85, ymax=1.85,
  xtick={0,5,10,15,20,25,30},
  ytick={-1.5,-1.0,-0.5,0.5,1.0,1.5},
  minor x tick num=4,
  minor y tick num=4,
```

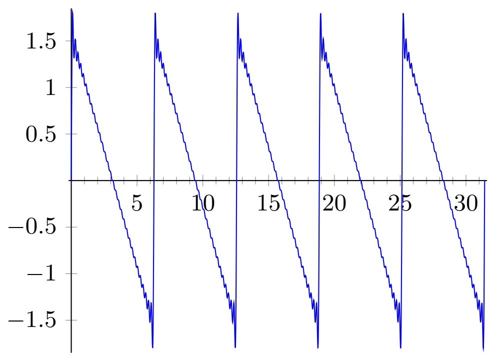


**Figure 3**: The partial sum $\sum_{k=1}^{30} \frac{\sin(kx)}{k}$ of the Fourier series of $f(x) = (\pi - x)/2$ illustrating the Gibbs phenomenon.

```
  axis lines=middle,
  axis line style={-}
  ]
% SYNTAX: Partial sum 30, from x = 0 to 10*pi,
% sampled in 1000 points.
\addLUADEDplot[color=blue,smooth]{30}
  {0}{10*math.pi}{1000};
\end{axis}
\end{tikzpicture}
```

See the output in Figure 3.

## 3 Third example: Runge-Kutta method

A differential equation is an equation that links an unknown function and its derivatives, and these equations play a prominent role in engineering, physics, economics, and other disciplines. When the value of the function at an initial point is fixed, a differential equation is known as an initial value problem. The mathematical theory of differential equations shows that, under very general conditions, an initial value problem has a unique solution. Usually, it is not possible to find the exact solution in an explicit form, and it is necessary to approximate it by means of numerical methods.

One of the most popular methods to integrate numerically an initial value problem

$$\begin{cases} y'(t) = f(t, y(t)), \\ y(t_0) = y_0 \end{cases}$$

is the *classical* Runge-Kutta method of order 4. With it, we compute in an approximate way the values $y_i \simeq y(t_i)$ at a set of points $\{t_i\}$ starting from $i = 0$ and with $t_{i+1} = t_i + h$ for every $i$ by the algorithm

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

where

$$\begin{cases} k_1 = hf(t_i, y_i), \\ k_2 = hf(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1), \\ k_3 = hf(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2), \\ k_4 = hf(t_i + h, y_i + k_3). \end{cases}$$

See, for instance, [15].

Here, we consider the initial value problem

$$\begin{cases} y'(t) = y(t)\cos\bigl(t + \sqrt{1 + y(t)}\bigr), \\ y(0) = 1. \end{cases}$$

In the Lua part of our `.tex` file, we compute the values $\{(t_i, y_i)\}$ and export them with pgfplots syntax by means of

```
\begin{luacode*}
-- Differential equation y'(t) = f(t,y)
-- with f(t,y) = y * cos(t+sqrt(1+y)).
-- Initial condition: y(0) = 1
function f(t,y)
    return y * math.cos(t+math.sqrt(1+y))
end

-- Code to write PGFplots data as coordinates
function print_RKfour(tMax,npoints,option)
    local t0 = 0.0
    local y0 = 1.0
    local h = (tMax-t0)/(npoints-1)
    local t = t0
    local y = y0
    if option~=[[]] then
        tex.sprint("\\addplot[" .. option
                   .. "] coordinates{")
    else
        tex.sprint("\\addplot coordinates{")
    end
    tex.sprint("("..t0..","..y0..")")
    for i=1, npoints do
        k1 = h * f(t,y)
        k2 = h * f(t+h/2,y+k1/2)
        k3 = h * f(t+h/2,y+k2/2)
        k4 = h * f(t+h,y+k3)
        y = y + (k1+2*k2+2*k3+k4)/6
        t = t + h
        tex.sprint("("..t..","..y..")")
    end
    tex.sprint("}")
end
\end{luacode*}
```

Also, we define the command

```
\newcommand\addLUADEDplot[3][]{%
  \directlua{print_RKfour(#2,#3,[[#1]])}%
}
```

to call the Lua routine from the LaTeX part (the parameter `#2` indicates the final value of $t$, and `#3` is the number of sampled points).

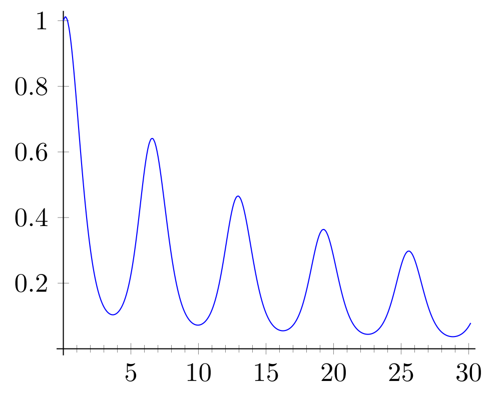


**Figure 4**: Solution of the differential equation $y'(t) = y(t)\cos\big(t + \sqrt{1 + y(t)}\big)$ with initial condition $y(0) = 1$.

Then, the graphic of Figure 4, which shows the solution of our initial value problem, is created by means of

```
\pgfplotsset{width=0.9\hsize}
\begin{tikzpicture}
\begin{axis}[xmin=-0.5, xmax=30.5,
  ymin=-0.02, ymax=1.03,
  xtick={0,5,...,30}, ytick={0,0.2,...,1.0},
  enlarge x limits=true,
  minor x tick num=4, minor y tick num=4,
  axis lines=middle, axis line style={-}
  ]
% SYNTAX: Solution of the initial value problem
% in the interval [0,30] sampled at 200 points
\addLUADEDplot[color=blue,smooth]{30}{200};
\end{axis}
\end{tikzpicture}
```

## 4 Fourth example: Lorenz attractor

The Lorenz attractor is a strange attractor that arises in a system of equations describing the 2-dimensional flow of a fluid of uniform depth, with an imposed vertical temperature difference. In the early 1960s, Lorenz [6] discovered the chaotic behavior of a simplified 3-dimensional system of this problem, now known as the Lorenz equations:

$$\begin{cases} x'(t) = \sigma(y(t) - x(t)), \\ y'(t) = -x(t)z(t) + \rho x(t) - y(t), \\ z'(t) = x(t)y(t) - \beta z(t). \end{cases}$$

The parameters $\sigma$, $\rho$, and $\beta$ are usually assumed to be positive. Lorenz used the values $\sigma = 10$, $\rho = 28$ and $\beta = 8/3$. The system exhibits a chaotic behavior for these values; in fact, it became the first example of a chaotic system. A more complete description can be found in [14].

Figure 5 shows the numerical solution of the Lorenz equations calculated with $\sigma = 3$, $\rho = 26.5$ and $\beta = 1$. Six orbits starting at several initial points close to $(0, 1, 0)$ are plotted in different colors; all of them converge to the 3-dimensional chaotic attractor known as the Lorenz attractor.

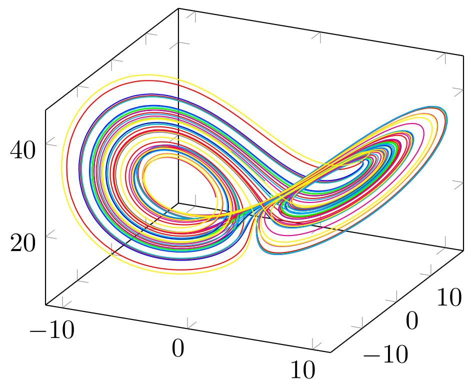


**Figure 5**: The Lorentz attractor (six orbits starting at several initial points).

The Lua part of the program uses a discretization of the Lorenz equations (technically, this is the explicit Euler method $y_{i+1} = y_i + hf(t_i, y_i)$, which is less precise than the Runge-Kutta method of the previous section, but enough to find the attractor):

```
\begin{luacode*}
-- Differential equation of Lorenz attractor
function f(x,y,z)
    local sigma = 3
    local rho = 26.5
    local beta = 1
    return {sigma*(y-x),
            -x*z + rho*x - y,
            x*y - beta*z}
end

-- Code to write PGFplots data as coordinates
function print_LorAttrWithEulerMethod(h,npoints,
                                      option)
    -- The initial point (x0,y0,z0)
    local x0 = 0.0
    local y0 = 1.0
    local z0 = 0.0
    -- add random number between -0.25 and 0.25
    local x = x0 + (math.random()-0.5)/2
    local y = y0 + (math.random()-0.5)/2
    local z = z0 + (math.random()-0.5)/2
    if option ~= [[]] then
        tex.sprint("\\addplot3[" .. option
                   .. "] coordinates{")
    else
        tex.sprint("\\addplot3 coordinates{")
    end
    -- dismiss the first 100 points
    -- to go into the attractor
    for i=1, 100 do
```

```
        m = f(x,y,z)
        x = x + h * m[1]
        y = y + h * m[2]
        z = z + h * m[3]
    end
    for i=1, npoints do
        m = f(x,y,z)
        x = x + h * m[1]
        y = y + h * m[2]
        z = z + h * m[3]
        tex.sprint("("..x..","..y..","..z..")")
    end
    tex.sprint("}")
end
\end{luacode*}
```

The function which calls the Lua part from the LaTeX part is

```
\newcommand\addLUADEDplot[3][]{%
  \directlua{print_LorAttrWithEulerMethod
             (#2,#3,[[#1]])}%
}
```

Here, the parameter #2 gives the step of the discretization, and #3 is the number of points.

The LaTeX part is the following. In it, we call the Lua function six times with different colors:

```
\pgfplotsset{width=.9\hsize}
\begin{tikzpicture}
\begin{axis}
% SYNTAX: Solution of the Lorenz system
% with step h=0.02 sampled at 1000 points.
 \addLUADEDplot[color=red,smooth]{0.02}{1000};
 \addLUADEDplot[color=green,smooth]{0.02}{1000};
 \addLUADEDplot[color=blue,smooth]{0.02}{1000};
 \addLUADEDplot[color=cyan,smooth]{0.02}{1000};
 \addLUADEDplot[color=magenta,smooth]{0.02}{1000};
 \addLUADEDplot[color=yellow,smooth]{0.02}{1000};
\end{axis}
\end{tikzpicture}
```

## References

[1] C. Feuersänger, *Manual for Package PGFPLOTS.* http://mirror.ctan.org/graphics/pgf/contrib/pgfplots/doc/pgfplots.pdf

[2] H. Hagen, *LuaTeX: Halfway to version 1*, *TUGboat*, Volume 30 (2009), No. 2, 183–186. http://tug.org/TUGboat/tb30-2/tb95hagen-luatex.pdf

[3] T. Hoekwater and H. Henkel, *LuaTeX 0.60: An overview of changes*, *TUGboat*, Volume 31 (2010), No. 2, 174–177. http://tug.org/TUGboat/tb31-2/tb98hoekwater.pdf

[4] P. Isambert, *Three things you can do with LuaTeX that would be extremely painful otherwise*, *TUGboat*, Volume 31 (2010), No. 3, 184–190. http://tug.org/TUGboat/tb31-3/tb99isambert.pdf

[5] P. Isambert, *OpenType fonts in LuaTeX*, *TUGboat*, Volume 33 (2012), No. 1, 59–85. http://tug.org/TUGboat/tb33-1/tb103isambert.pdf

[6] E. N. Lorenz, *Deterministic Nonperiodic Flow*, J. Atmospheric Sci., Volume 20 (1963), No. 2, 130–141. http://dx.doi.org/10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2

[7] A. Mahajan, *LuaTeX: A user's perspective*, *TUGboat*, Volume 30 (2009), No. 2, 247–251. http://tug.org/TUGboat/tb30-2/tb95mahajan-luatex.pdf

[8] A. Mertz and W. Slough, *Graphics with PGF and TikZ*, *TUGboat*, Volume 28 (2007), No. 1, 91–99. http://tug.org/TUGboat/tb28-1/tb88mertz.pdf

[9] A. Mertz and W. Slough, *A TikZ tutorial: Generating graphics in the spirit of TeX*, *TUGboat*, Volume 30 (2009), No. 2, 214–226. http://tug.org/TUGboat/tb30-2/tb95mertz.pdf

[10] M. Pégourié-Gonnard, *The luacode package.* http://mirror.ctan.org/macros/luatex/latex/luacode/luacode.pdf

[11] A. Reutenauer, *LuaTeX for the LaTeX user: An introduction*, *TUGboat*, Volume 30 (2009), No. 2, 169. http://tug.org/TUGboat/tb30-2/tb95reutenauer.pdf

[12] T. Tantau, *TikZ & PGF.* http://mirror.ctan.org/graphics/pgf/base/doc/generic/pgf/pgfmanual.pdf

[13] Wikipedia, *Gibbs phenomenon.* http://en.wikipedia.org/wiki/Gibbs_phenomenon

[14] Wikipedia, *Lorenz system.* http://en.wikipedia.org/wiki/Lorenz_system

[15] Wikipedia, *Runge-Kutta methods.* http://en.wikipedia.org/wiki/Runge-Kutta_methods

◇ Juan I. Montijano, Mario Pérez, Luis Rández and Juan Luis Varona
Universidad de Zaragoza (Zaragoza, Spain) and Universidad de La Rioja (Logroño, Spain)
{monti,mperez,randez} (at) unizar dot es and jvarona (at) unirioja dot es

# Parsing PDF content streams with LuaTeX

Taco Hoekwater

## Abstract

The new `pdfparser` library in LuaTeX allows parsing of external PDF content streams directly from within a LuaTeX document. This paper explains its origin and usage.

## 1 Background

Docwolves' main product is an infrastructure to facilitate paperless meetings. One part of the functionality is handling meeting documents, and to do so it offers the meeting participants a method to distribute, share, and comment on such documents by means of an intranet application, as well as an iPad app.

Meeting documents typically consist of a meeting agenda, followed by included appendices, combined into a single PDF file. Such documents can have various revisions, for example if a change has been made to the agenda or if an appendix has to be added or removed. After such a change, a newly combined PDF document is re-distributed.

Annotations can be made on these documents and these can then be shared with other meeting participants, or just communicated to the server for safekeeping. Like documents, annotations can be updated as well.

All annotations are made on the iPad, with an (implied) author and an intended audience. Annotations apply to a specific part of the source text, and come in a few types (highlight, sticky note, freehand drawing). The iPad app communicates with a network server to synchronize shared annotations between meeting participants.

## 2 The annotation update problem

The server–client protocol aims to be as efficient as possible, especially in the case of communication with the iPad app, since bandwidth and connection costs can be an issue.

This means that for any annotation on a referenced document, only the document's internal identification, the (PDF) page number, and the beginning and end word indices on the page are communicated back and forth. This is quite efficient, but gives rise to the following problem:

> When a document changes, e.g. if an extra meeting item is added, all annotations following that new item have to be updated because their placement is off.

Editor's note: Originally published in *ConTeXt Group: Proceedings, 6th meeting*, pp. 19–23. Reprinted with permission.

The full update process is quite complicated; the issue this paper deals with is that the server software needs to know what words are on any PDF page, as well as their location on that page, and therefore its text extraction process has to agree with the same process on the iPad.

## 3 PDF text extraction

Text extraction is a two-step process. The actual drawing of a PDF page is handled by PostScript-style postfix operators. These are contained in objects that are called page content streams.

After decompression of the PDF, the beginning of a content stream might look like this:

```
59 0 obj
<< /Length 4013 >>
stream
0 g 0 G
1 g 1 G
q
0 0 597.7584 448.3188 re f
Q
0 g 0 G
1 0 0 1 54.7979 44.8344 cm
...
```

Here `g`, `G`, `q`, `re`, `f`, `Q`, and `cm` are all (postfix) operators, and the numeric values are all arguments. As you see, not all operators take the same number of arguments (`g` takes one, `q` zero, and `re` four). Other operators may take, for instance, string-valued arguments instead of numeric ones. There are a bit more than half a dozen different types.

To process such a stream easily, it is best to separate the task (at least conceptually) into two separate tasks. First there is a lexing stage, which entails converting the raw bytes into combinations of values and types (tokens) that can be acted upon.

Separate from that, there is the interpretation stage, where the operators are actually executed with the tokenized arguments that have preceded it.

### 3.1 PDF text extraction on the iPad

It is very easy on an iPad to display a representation of a PDF page, and Apple also provides a convenient interface to do the lexing of PDF content streams that is the first step in getting the text from the page. But to find out where the PDF objects are, one has to interpret the PDF document stream oneself, and that is the harder part of the text extraction operation.

### 3.2 PDF text extraction on the server

On the server side, there is a similar problem at a different stage: displaying a PDF is easy, and even literal text extraction is easy (with tools like `pdftotext`). However, that does not give you the location of the text on the page. On the server, Apple's lexing interface is not available, and the available PDF library (`libpoppler`) does not offer similar functionality.

## 4 Our solution

We needed to write text extraction software that can be used on both platforms, to ensure that the same releases of server and iPad software always agreed perfectly on the what and where of the text on the PDF page.

Both platforms use a stream interpreter written by ourselves in C, with the iPad software starting from the Apple lexer, and the server software starting from a new lexer written from scratch.

The prototype and first version of the newly created stream interpreter as well as the server-side lexer were written in Lua. LuaTEX's `epdf` library — `libpoppler` bindings for Lua, see below — were a very handy tool at that stage. The code was later converted back to C for compilation into a server-side helper application as well as the iPad App, but originally it was written as a `texlua` script.

A side effect of this development process is that the lexer could be offered as a new LuaTEX extension, and so that is exactly what we have done.

## 5 About the 'epdf' library

This library is written by Hartmut Henkel, and it provides Lua access to the `poppler` library included in LuaTEX. For instance, it is used by ConTEXt to preserve links in external PDF figures.

The library is fairly extensive, but a bit low-level, because it closely mimics the `libpoppler` interface. It is fully documented in the LuaTEX reference manual, but here is a small example that extracts the page cropbox information from a PDF document:

```
local function run (filename)
   local doc = epdf.open(filename)
   local cat = doc:getCatalog()
   local numpages = doc:getNumPages()
   local pagenum  = 1
   print ('Pages: ' .. numpages)
   while pagenum <= numpages do
      local page = cat:getPage(pagenum)
      local cbox = page:getCropBox()
      print (string.format(
           'Page %d: [%g %g %g %g]',
           pagenum, cbox.x1, cbox.y1,
           cbox.x2, cbox.y2))
      pagenum = pagenum + 1
   end
end
run(arg[1])
```

## 6 Lexing via poppler

As said above, a lexer converts bytes in the input text stream into tokens, and these tokens have types and values. `libpoppler` provides a way to get one byte from a stream using the `getChar()` method, and it also applies any stream filters beforehand, but it does not create such tokens.

### 6.1 Poppler limitations

There is no way to get the full text of a stream as a whole; it has to be read byte by byte.

Also, if the page content consists of an array of content streams instead of a single entry, the separate content streams have to be manually concatenated.

And content streams have to be 'reset' before the first use.

Here is some example code for reading a stream, using the `epdf` library:

```
function parsestream(stream)
  local self = { streams = {} }
  local thetype = type(stream)
  if thetype == 'userdata' then
    self.stream = stream:getStream()
  elseif thetype == 'table' then
    for i,v in ipairs(stream) do
      self.streams[i] = v:getStream()
    end
    self.stream = table.remove(
                    self.streams,1)
  end
  self.stream:reset()
  local byte = getChar(self)
  while byte >= 0 do
    ...
    byte = getChar(self)
  end
  if self.stream then
    self.stream:close()
  end
end
```

In the code above, any interesting things you want to do are inserted at the `...` spot. The example makes use of one helper function, `getChar`, which looks like this:

```
local function getChar(self)
  local i = self.stream:getChar()
  if (i<0) and (#self.streams>0) then
    self.stream:close()
    self.stream = table.remove(
                   self.streams, 1)
    self.stream:reset()
    i = getChar(self)
  end
  return i
end
```

## 7 Our own lexer: 'pdfscanner'

The new lexer we wrote does create tokens. Its Lua interface accepts either a poppler stream, or an array of such streams. It puts PDF operands on an internal stack and then executes user-selected operators.

The library `pdfscanner` has only one function, `scan()`. Usage looks like this:

```
require 'pdfscanner'
function scanPage(page)
  local stream = page:getContents()
  local ops = createOperatorTable()
  local info = createParserState()
  if stream then
    if stream:isStream()
       or stream:isArray() then
      pdfscanner.scan(stream, ops, info)
    end
  end
end
```

The above functions `createOperatorTable()` and `createParserState()` are helper functions that create arguments of the proper types.

### 7.1 The `scan()` function

As you can see, the `scan()` function takes three arguments, which we describe here.

The first argument should be either a PDF stream object or a PDF array of PDF stream objects (those options comprise the possible return values of `<Page>:getContents()` and `<Object>:getStream()` in the `epdf` library).

The second argument should be a Lua table where the keys are PDF operator name strings and the values are Lua functions (defined by you) that are used to process those operators. The functions are called whenever the scanner finds one of these PDF operators in the content stream(s).

Here is a possible definition of the helper function `createOperatorTable()`:

```
function createOperatorTable()
  local ops = {}
  -- handlecm is defined below
  ops['cm'] = handlecm
  return ops
end
```

The third argument is a Lua variable that is passed on to provide context for the processing functions. This is needed to keep track of the state of the PDF page since PDF operators, and especially those that change the graphics state, can be nested.[1]

[1] In Lua this could have been handled by upvalues or global variables. This third argument was a concession made to the planned conversion to C.

In its simplest form, the creation of this page state looks like this:

```
function createParserState()
  local stack = {}
  stack[1] = {}
  stack[1].ctm =
    AffineTransformIdentity()
  return stack
end
```

Internally, `pdfscanner.scan()` loops over the input stream content bytes, creating tokens and collecting operands on an internal stack until it finds a PDF operator. If that operator's name exists in the given operator table, then the associated Lua function is executed. After that function has run (or when there is no function to execute) the internal operand stack is cleared in preparation for the next operator, and processing continues.

The processing functions are called with two arguments: the `scanner` object itself, and the `info` table that was passed as the third argument to `pdfscanner.scan`.

The `scanner` argument to the processing functions is needed because it offers various methods to get the actual operands from the internal operand stack.

### 7.2 Extracting tokens from the scanner

The lowest-level function available in `scanner` is `scanner:pop()` which pops the top operand of the internal stack, and returns a Lua table where the object at index one is a string representing the type of the operand, and object two is its value.

The list of possible operand types and associated Lua value types is:

| | |
|---|---|
| `integer` | ⟨*number*⟩ |
| `real` | ⟨*number*⟩ |
| `boolean` | ⟨*boolean*⟩ |
| `name` | ⟨*string*⟩ |
| `operator` | ⟨*string*⟩ |
| `string` | ⟨*string*⟩ |
| `array` | ⟨*table*⟩ |
| `dict` | ⟨*table*⟩ |

In the cases of `integer` or `real`, the value is always a Lua (floating point) number.

In the case of `name`, the leading slash is always stripped.

In the case of `string`, please bear in mind that PDF supports different types of strings (with different encodings) in different parts of the PDF document, so you may need to reencode some of the results; `pdfscanner` always outputs the byte stream without reencoding anything. `pdfscanner` does not differentiate between literal strings and hexadecimal strings (the hexadecimal values are decoded), and it treats the stream data for inline images as a string that is the single operand for `EI`.

In the case of `array`, the table content is a list of `pop` return values.

In the case of `dict`, the table keys are PDF name strings and the values are `pop` return values.

While parsing a PDF document that is known to be valid, one usually knows beforehand what the types of the arguments will be. For that reason, there are a few more `scanner` methods defined:

- `popNumber()` takes a number object off of the operand stack.
- `popString()` takes a string object off ...
- `popName()` takes a name object off ...
- `popArray()` takes an array object off ...
- `popDict()` takes a dictionary object off ...
- `popBool()` takes a boolean object off ...

A simple operator function could therefore look like this. `handlecm` was used in an example above; the PDF `cm` operator “concatenates” onto the current transformation matrix. (The `Affine...` functions used here are left as an exercise to the reader).

```
function handlecm (scanner, info)
  local ty = scanner:popNumber()
  local tx = scanner:popNumber()
  local d  = scanner:popNumber()
  local c  = scanner:popNumber()
  local b  = scanner:popNumber()
  local a  = scanner:popNumber()
  local t = AffineTransformMake(a,b,c,d,tx,ty)
  local stack = info.stack
  local state = stack[#stack]
  state.ctm =
    AffineTransformConcat(state.ctm,t)
end
```

Finally, there is also the `scanner:done()` function which allows you to quit the processing of a stream once you have learned everything you want to learn. Specifically, this comes in handy while parsing `/ToUnicode`, because there usually is trailing garbage that you are not interested in. Without `done`, processing only ends at the end of the stream, wasting CPU cycles.

## 8 Summary

The new `pdfparser` library in LuaTEX allows parsing of external PDF content streams directly from within a LuaTEX document. While this paper explained its usage, the formal documentation of the new library is the LuaTEX reference manual. Happy LuaTEX-ing!

◇ Taco Hoekwater
Docwolves B.V.
`http://luatex.org`

# ModernDvi: A high quality rendering and modern DVI viewer

Antoine Bossard and Takeyuki Nagao

## Abstract

TeX users have long relied on the device independent file format (DVI) to preview their documents while editing. However, innovation has been scarce in this area, and users have to rely on years-old, or even decades-old software, facing increasing compatibility issues with modern systems. In this paper, we describe ModernDvi, a new DVI viewer Windows Store application, offering high quality and fast rendering, wait-free, outperforming existing solutions in these areas. Additionally, ModernDvi has been built around today's usability standards and expectations: tablets, touch-friendly, high-resolution output are examples of addressed issues.

## 1 Introduction: The DVI file format

DVI is a file format, namely the "DeVice Independent" file format. DVI documents are typically produced by the TeX and LaTeX typesetting programmes. TeX and its high-level abstraction LaTeX, which is written in the TeX macro language, were introduced by Donald E. Knuth [7] and Leslie Lamport [10], respectively, as solutions for producing high-quality documents dealing with mathematics and science in general, and especially their complex formulæ and notations. Hàn Thế Thành later created an extension of TeX called pdfTeX [5] enabling direct output of portable document format (PDF) in addition to the traditional DVI format. There are thus two different kinds of output by TeX and LaTeX, viz. DVI and PDF. Although the PDF format has become more popular, some people still need and depend on the classical DVI format. In fact, an experiment with the total 5814 LaTeX documents collected from the preprints of arXiv [6] in the single month of January 2012, shows that 4168 items (approx. 72%) can be compiled by both LaTeX and pdfLaTeX, and that 540 items (approx. 9%) work with LaTeX but not with pdfLaTeX [13].

The DVI format is minimalistic. Roughly speaking, it is a binary format consisting of commands to (i) define or select a font to utilize, (ii) draw a single character or a filled rectangle at the current reference point, (iii) manipulate internal integer registers (including the current reference point and the identifier of the current font), (iv) include binary data (called DVI specials) for various purposes, and (v) mark the beginning and ending of pages/documents.

The simplicity of the DVI format facilitates creation of tools and applications to help authors prepare manuscripts and post-process existing articles. Such tools include DVI viewers which render and display the contents of a DVI file on screen, and also converters to various formats including PostScript (dvips), PDF (dvipdfm), bitmap images (dvipng), etc. It is much harder to create such tools for the PDF format, since that format is more complicated and thus difficult to parse and analyse the encoded data.

A major drawback of the DVI format is that it requires external files and/or tools to completely render its contents in some use cases. For example, if an author includes a figure in his paper (e.g. using Encapsulated PostScript format), then the generated DVI file contains a DVI special that consists of a code fragment in the PostScript language to include the specified image file. This means that the DVI file is not self-contained, and one needs a rasterizer of PostScript, e.g. Ghostscript, to render its contents. Another common issue is the lack of the feature to embed fonts. A DVI file actually contains only the name (such as cmr10) and the size of the utilized fonts. There is no standardized way to embed raster or vector fonts to a DVI file. This difficulty can be overcome by using pdf(LA)TeX which provides the features of including external PDF files and embedding TrueType and Type 1 fonts.

## 2 Previous works

A handful of DVI viewers exist; however, many are not updated any more: mdvi [2] and windvi [15] are examples. Well-known viewers for Unix-based platforms include xdvi [17]. Still usable alternatives are even harder to find on the MS Windows operating system. YAP [16] is the DVI viewer of choice on Windows as it is bundled with the MiKTeX distribution [16]. One can also cite dviout [14] as another DVI viewer on Windows.

A user looking for a viable solution to work with DVI files will face several issues. First, all existing viewers are now considered legacy software: they have been designed for decades-old operating systems and do not meet modern requirements regarding software, hardware, interface, and usability in general. The first problem a user may encountered is software compatibility: for example, as of the Mountain Lion release of Mac OS X, X11 is not included any more, which will thus hamper the installation and usage of the usual viewer on Unix, xdvi. Even more radical changes to an OS (e.g. at the device driver layer) may completely break compatibility and make a legacy viewer unusable. Additionally, classic workstations are now on their way out, and mobile devices, touch screens or other advanced interface mechanisms are in full swing. An unadapted user interface such

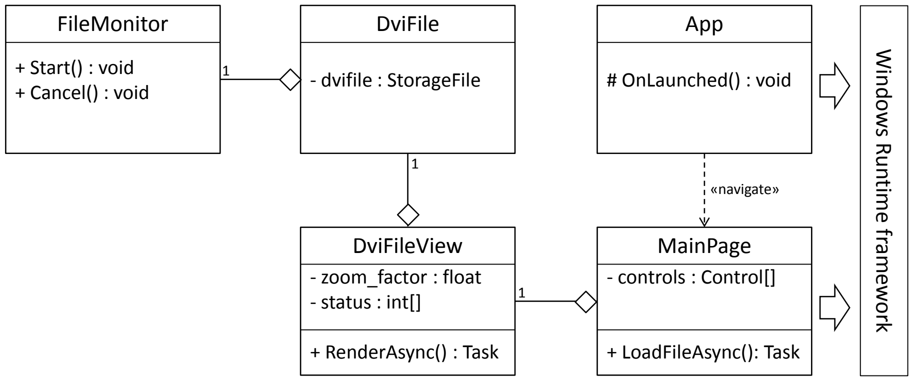


**Figure 1**: Simplified UML class diagram.

as that of a legacy viewer will thus severely harm usability, or even make it completely unusable.

Hardware evolution affects more than user interfaces. Processors have also seen important changes in their architectures. Now, every device uses a multi-core CPU, thus allowing faster or more calculations at the same time: this is parallel processing. Legacy DVI software has usually not been designed to utilise such increased computing power and will thus perform poorly compared to modern applications in general, such ad word processors.

Finally, rendering quality of DVI viewers is usually not on par with modern displays and their high resolutions. Again, when they were introduced, these systems had to cope with much more restrictive hardware and software environments that we have now. Nowadays, users' expectations regarding display quality are very high, and legacy viewers are lacking in this area.

We could continue enumerating shortcomings in legacy DVI viewers (e.g. font generation before rendering, slow page scrolling, etc.), but the point is clear. By proposing our new DVI viewer, named ModernDvi, we are first aiming at filling the gaps of legacy DVI viewers, gaps which have been increasing over time due to constant technology evolution. The introduction of original features as detailed in Section 4 is also an important part of our work.

## 3 ModernDvi: System overview

In this section, we give some insight into ModernDvi's structure by first addressing software engineering considerations, then rendering techniques, and finally parallel processing.

### 3.1 Software engineering

ModernDvi is a Windows Store application [11]. It makes use of the Windows Runtime API and is thus compatible with x86 and ARM processor architectures. Windows Store applications can be deployed to any Windows 8 device, including PCs and tablets. Porting ModernDvi to the Windows Phone platform is also possible, but this remains work in progress.

This app has been developed using the C# programming language, and thus features an object-oriented architecture. Let us give an overview of the main objects defined in ModernDvi. Common to every Windows Store app, the entry point of the application is located inside the `App` class. The app itself is built around the `MainPage` class which defines the application view port, displaying controls and visual elements in general. Each DVI document is associated with an instance of the `DviFile` class which, amongst others, importantly stores the system handle to the DVI file, a critical app issue detailed further in Section 5. The `DviFile` class also holds a reference to a `FileMonitor` object which is in charge of monitoring file changes made to the DVI document. Then, in order to display the content of a DVI document (i.e. an instance of `DviFile`), the class `DviFileView` is instantiated. Such an object is in charge of rendering each page of the DVI document, and thus contains several view settings such as page dimension and zooming information. Changing parameters like the zoom factor will automatically create a new `DviFileView` object. Figure 1 shows a simplified UML class diagram of ModernDvi.

An important point is that `DviFileView` objects are completely isolated, especially from the view port objects of `MainPage` so as to facilitate the multi-thread approach detailed in Section 3.3 below. The sole relation between these two classes is a reference to a `DviFileView` instance inside `MainPage`; this reference is used to add (i.e. display) the view inside the view port. The DVI document is loaded with the `LoadFileAsync()` method of `MainPage`, and rendered with the `RenderAsync()` method of `DviFileView`.

## 3.2 Rendering

The rendering of a DVI file is performed in two stages. In the first stage, the content of the DVI file is parsed and the result stored in an object, which contains a font table (i.e. a mapping from font numbers to font names) and the DVI commands for each page. All the required TeX font metrics and other files (such as PK fonts and virtual fonts) are loaded into memory for later reference. The bounding box of every page is computed at this point, in parallel using multiple threads.

In the second stage, the contents of each page is rendered lazily. More precisely, it is only at the moment when a page is about to appear in the view that the rendering of the page is undertaken. The engine prepares an off-screen buffer for the page, and rasterizes the glyphs onto this buffer according to the DVI commands corresponding to the page. The off-screen buffer is then converted to an image file (using Portable Network Graphics format) and stored in memory. Compression is utilized here in order to save memory space.

## 3.3 Parallel processing

As recalled in Section 2, parallel processing is now a common feature of our modern devices, from PCs to smartphones via tablets. Modern applications are thus expected to make use of this increased computational power available, and this is what we achieved with ModernDvi.

ModernDvi uses parallelisation for two distinct tasks. First, DVI rendering, as briefly detailed in Section 3.2, needs to compute the bounding box of each page of the DVI document. This is a time consuming task. So, by executing these calculations in parallel, we were able to significantly speed up this process. In practice, we relied on the `Task.Run()` function of the `System.Threading.Tasks` class of the API which queues its parameter to run on the thread pool managed by the framework.

Then, once pages have been prepared in memory, comes the display phase: bringing inside the view port each of the (visible) pages. This task is also time consuming since it involves I/O stream operations, UI elements' instantiation and display, and of course placement routines for correct positioning of the pages and their corresponding visual elements in the view port. So again, instead of performing these tasks sequentially, one after the other, we have devised a parallel solution for this lengthy process and observed significant time gains.

Because of the strongly asynchronous nature of C# for apps, user actions in the UI are often partly postponed after their start via the keyword combination `async`/`await`, and the program returns to the UI thread. The merit of this approach is that the UI is always responsive, that is non-blocking, lock-free and wait-free. However, this can also be problematic as it means, in our case, that a user can request a document load (i.e. view refresh) several times, with most of these requests being still processed, that is not completed yet. To address this issue, ModernDvi tracks the rendering state of each page of the DVI document through a `status[]` array indexed on document pages, whose values are as follows:

0: page not displayed (and not pending); this is the default status of each page.

1: page pending; the program is preparing the document page.

2: page ready to be displayed; the program has finished preparing the page.

3: page displayed; the program has finished adding the page into the view.

Each view refresh task is thus accessing this array, which is a class member of `DviFileView`. So, we have to regulate the accesses of these asynchronous tasks to this array to avoid race conditions and concurrent access problems. This problem is solved by using the compare-and-swap (CAS) mechanism which is implemented in C# by the `CompareExchange` method of the `System.Threading.Interlocked` class. Using this, we are able to automatically check and update the rendering status of a document page. The practical result is that the Interlocked functionality allows us to (1) avoid performing the rendering of one page several times, and (2) avoid displaying the same rendered page several times.

So, thanks to this approach, our DVI document rendering process is (1) performed asynchronously, thus always retaining the UI responsiveness with, for example, very smooth scrolling, and (2) handling multiple page preparations and displays in parallel, thanks to multiple threads. The number of threads is managed by the framework thread pool and is thus transparent to the developer. Obviously, the more cores in your device, the more threads can be running concurrently. An excerpt of the corresponding source code is given in Figure 2.

## 4 Notable features

We present in this section several notable features of ModernDvi.

### 4.1 No font generation needed

Existing DVI viewers generate on-demand PK font files from, for instance, METAFONT source files [8, 9] or PostScript font files [1]. The main problem with

```
IEnumerable<Task> tasks = Enumerable.Range(0,
                       visible_pages).Select(i => {
 return Task.Run( () => {
  int current_page = first_page + i;
  int original_status = System.Threading.Interlocked
   .CompareExchange (ref status[current_page], 1, 0);

  if (original_status == 0) {
   pages[current_page] = DviFile.ctx
              .CreateRenderablePage(dvifile.Document
              .GetPage(current_page));
   status[current_page] = 2;
  }
 });
});
await Task.WhenAll(tasks);
```

**Figure 2**: Parallel execution with `Task.Run()`.

this approach is the rendering time delay faced by the user upon the DVI document loading.

ModernDvi has adopted a different approach: PK font files are generated in advance and bundled inside the application so that no font generation phase is required at any time. Thus, we can achieve a significant speed-up of the initial loading phase compared to legacy DVI viewers.

Of course, to maximise usability we need to include with ModernDvi the fonts needed by users. Many hundreds of fonts are available for TEX usage; looking at the CTAN font area [4] gives a good overview of the situation. So, we conducted an experiment to measure the popularity of these fonts, and thus obtained a list of the most-used fonts. Specifically, we collected preprints published on arXiv [6] for the year of 2012 and analysed the resulting 48 772 samples of LATEX source files to see which fonts they used, i.e. which font files are needed for their rendering. Table 1 shows the results with, not surprisingly, Computer Modern leading the list.

**Table 1**: Most-used fonts in 2012 arXiv.org preprints.

| Rank | Font | Freq. | % | Cumul. % |
|---|---|---|---|---|
| 1 | cmsy10 | 64 830 | 4.69 | 4.69 |
| 2 | cmr10 | 60 321 | 4.36 | 9.05 |
| 3 | cmmi10 | 52 705 | 3.81 | 12.87 |
| 4 | cmbx12 | 48 774 | 3.53 | 16.39 |
| 5 | ptmr8r | 44 209 | 3.20 | 19.59 |
| 6 | cmex10 | 41 872 | 3.03 | 22.62 |
| 7 | cmr8 | 39 221 | 2.84 | 25.46 |
| 8 | cmbx10 | 36 725 | 2.66 | 28.12 |
| 9 | cmr6 | 33 900 | 2.45 | 30.57 |
| 10 | cmmi8 | 30 895 | 2.23 | 32.80 |
| 11 | (others) | 928 916 | 67.20 | 100.00 |
| Total | | 1 382 368 | | |

We observed that about 1,000 fonts sufficed to render all gathered DVI files, which we took as representative of most documents, given the broad area covered by arXiv. We thus gathered the corresponding METAFONT source files and generated PK files for each font using the `mktexpk` utility [3]. Particular care has been taken to avoid any licensing violations for the fonts bundled in ModernDvi, with problematic fonts replaced by freely available ones; for instance, Linotype Palatino has been replaced by URW Palladio. Out of the ≈1,000 fonts identified, 769 have been included in ModernDvi. Excluded fonts are either rarely used or for exotic languages.

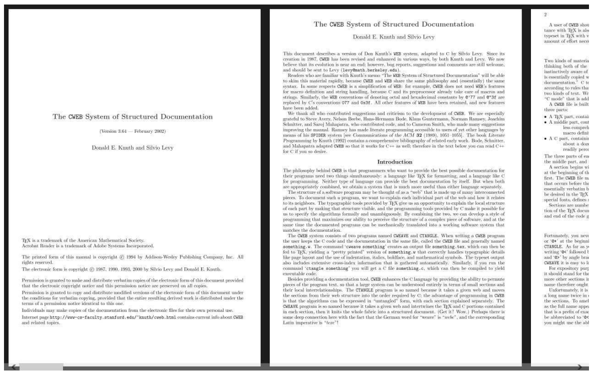
## 4.2 File change monitoring

ModernDvi includes a file monitoring system which triggers a new document rendering (refresh) upon any file change. Concretely, if this feature is enabled, ModernDvi will check at a specific time interval whether changes have occurred to the loaded document. Such changes are detected as follows.

1. Initialise a `LastModified` object of type `DateTimeOffset` with the `DateModified` value of the `BasicProperties` instance returned by a first call to `GetBasicPropertiesAsync` on the current document.
2. Start the timer (`DispatchTimer` object).
3. On timer tick, retrieve a new instance of `BasicProperties` via a call to `GetBasicPropertiesAsync` on the current document. Then compare the stored `LastModified` value with the `DateModified` value of the `BasicProperties` instance just retrieved. If `LastModified` is smaller (i.e. older) than `DateModified`, request a new rendering of the DVI document. Finally, set `LastModified` to `DateModified`.

Additional care needed to be taken regarding `GetBasicPropertiesAsync`. Even though it is not mentioned in the documentation [12], multiple calls to `GetBasicPropertiesAsync` on the same file will throw an exception. Only one call to `GetBasicPropertiesAsync` is allowed at a time: one has to wait for the previous call to return before making another call. And because this method is asynchronous (i.e. awaited, see Section 3.3), we have to enforce a guard to avoid doing so. This is again achieved by using the CAS mechanism `CompareExchange` method of the `System.Interlocked` class.

Lastly, we must note that this monitoring system does not work for DVI files that are contained inside an archive (see Section 4.5). This is due to the limitations imposed by the Windows Runtime API, limitations induced by security concerns. See Section 5 for additional details.



The CWEB System of Structured Documentation

(Version 3.64 — February 2002)

Donald E. Knuth and Silvio Levy

TeX is a trademark of the American Mathematical Society.
Acrobat Reader is a trademark of Adobe Systems Incorporated.

The printed form of this manual is copyright © 1994 by Addison-Wesley Publishing Company, Inc. All rights reserved.

The electronic form is copyright © 1987, 1990, 1993, 2000 by Silvio Levy and Donald E. Knuth.

Permission is granted to make and distribute verbatim copies of the electronic form of this document provided that the electronic copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of the electronic form of this document under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Individuals may make copies of the documentation from the electronic files for their own personal use.

Internet page `http://www-cs-faculty.stanford.edu/~knuth/cweb.html` contains current info about CWEB and related topics.

The CWEB System of Structured Documentation

Donald E. Knuth and Silvio Levy

This document describes a version of Don Knuth's WEB system, adapted to C by Silvio Levy. Since its creation in 1987, CWEB has been revised and enhanced in various ways, by both Knuth and Levy. We now believe that its evolution is near an end; however, bug reports, suggestions and comments are still welcome, and should be sent to Levy (`levy@math.berkeley.edu`).

Readers who are familiar with Knuth's memo "The WEB System of Structured Documentation" will be able to skim this material rapidly, because CWEB and WEB share the same philosophy and (essentially) the same syntax. In some respects CWEB is a simplification of WEB: for example, CWEB does not need WEB's features for macro definition and string handling, because C and its preprocessor already take care of macros and strings. Similarly, the WEB conventions of denoting octal and hexadecimal constants by `@'77` and `@"3f` are replaced by C's conventions `077` and `0x3f`. All other features of WEB have been retained, and new features have been added.

We thank all who contributed suggestions and criticism to the development of CWEB. We are especially grateful to Steve Avery, Nelson Beebe, Hans-Hermann Bode, Klaus Guntermann, Norman Ramsey, Joachim Schnitter, and Saroj Mahapatra, who contributed code, and to Cameron Smith, who made many suggestions improving the manual. Ramsey has made literate programming accessible to users of yet other languages by means of his SPIDER system [see *Communications of the ACM* **32** (1989), 1051–1055]. The book *Literate Programming* by Knuth (1992) contains a comprehensive bibliography of related early work. Bode, Schnitter, and Mahapatra adapted CWEB so that it works for C++ as well; therefore in the text below you can read C++ for C if you so desire.

**Introduction**

The philosophy behind CWEB is that programmers who want to provide the best possible documentation for their programs need two things simultaneously: a language like TeX for formatting, and a language like C for programming. Neither type of language can provide the best documentation by itself. But when both are appropriately combined, we obtain a system that is much more useful than either language separately.

The structure of a software program may be thought of as a "web" that is made up of many interconnected pieces. To document such a program, we want to explain each individual part of the web and how it relates to its neighbors. The typographic tools provided by TeX give us an opportunity to explain the local structure of each part by making that structure visible, and the programming tools provided by C make it possible for us to specify the algorithms formally and unambiguously. By combining the two, we can develop a style of programming that maximizes our ability to perceive the structure of a complex piece of software, and at the same time the documented programs can be mechanically translated into a working software system that matches the documentation.

The CWEB system consists of two programs named CWEAVE and CTANGLE. When writing a CWEB program the user keeps the C code and the documentation in the same file, called the CWEB file and generally named `something.w`. The command '`cweave something`' creates an output file `something.tex`, which can then be fed to TeX, yielding a "pretty printed" version of `something.w` that correctly handles typographic details like page layout and the use of indentation, italics, boldface, and mathematical symbols. The typeset output also includes extensive cross-index information that is gathered automatically. Similarly, if you run the command '`ctangle something`' you will get a C file `something.c`, which can then be compiled to yield executable code.

Besides providing a documentation tool, CWEB enhances the C language by providing the ability to permute pieces of the program text, so that a large system can be understood entirely in terms of small sections and their local interrelationships. The CTANGLE program is so named because it takes a given web and moves the sections from their web structure into the order required by C; the advantage of programming in CWEB is that the algorithms can be expressed in "untangled" form, with each section explained separately. The CWEAVE program is so named because it takes a given web and intertwines the TeX and C portions contained in each section, then it knits the whole fabric into a structured document. (Get it? Wow.) Perhaps there is some deep connection here with the fact that the German word for "weave" is "*webe*", and the corresponding Latin imperative is "*texe*"!

**Figure 3**: Horizontal display mode.

## 4.3 Vertical and horizontal display modes

It is a steady trend: screens are getting wider and wider. In order to take full advantage of such hardware configurations, we have implemented two different page flows in ModernDvi: vertical and horizontal display modes.

The vertical display mode is the classic top-down document flow that can be found in almost all viewers or WYSIWYG editors. The horizontal display mode is an original left-right document flow that proves comfortable when working on wide displays. Effectively, multiple pages can be displayed side-by-side on a wide screen at the same time, enabling a seamless reading experience. An illustration is given in Figure 3.

In addition to a global setting defining the default display mode of ModernDvi, the user interface contains an easily accessible "Switch flow" button that enables the user to quickly (instantly) switch the document display mode, without having to reload or render anything.

## 4.4 Automatic zooming

In the current iteration of ModernDvi, we have implemented three zoom levels: window fit, page fit and thumbnails. The user can choose between these modes to render the DVI document by automatically adapting to the screen resolution. Because the document rendering process is repeated when changing the zoom level, the rendering quality remains crisp at any time. An illustration of these three zoom levels is given in Figure 4.

To adapt to the user's screen, our `DviFileView` class (see Figure 1) registers the `Loaded` event of the view port. When fired, this event signals the availability of the `ActualWidth` and `ActualHeight` properties of the view port, giving the user screen resolution in pixels (precisely the screen area allocated to the view port). Because of the asynchronous nature of applications and especially their user interface, failing to use the event model will most likely result in null values for these two properties. In practice, the UI thread may not have completed the interface setup work when reading these two properties.

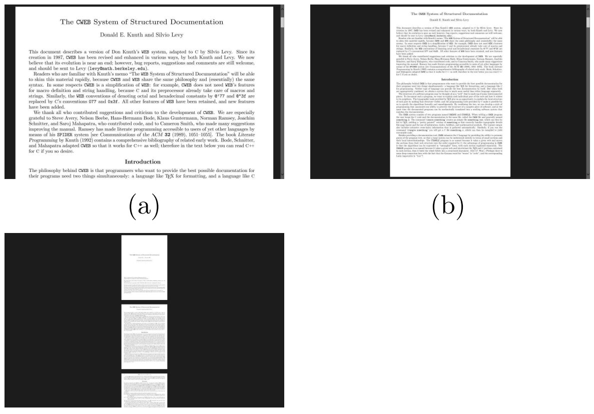


**Figure 4**: Vertical modes of (a) window fit, (b) page fit and (c) thumbnails zoom levels.

## 4.5 Archive formats and file types

DVI documents are often distributed as archives. For instance, most documents in the arXiv preprint repository are stored as tarballs. To facilitate display of such documents, we implemented routines in charge of uncompressing and extracting archives. Files are stored in the temporary folder of the app. Table 2 summarises the file formats ModernDvi supports.

Additionally, so as to correctly handle these different file formats, we implemented an accurate file type detection routine that can recognise each of the supported file formats given a file, regardless of its extension. In practice, one cannot be sure that a file will have an extension, or that it is accurate. And this is without mentioning that there often exist many different extensions for a single file format. Thus, we analyse a file's header data to determine its type.

**Table 2**: Supported file formats.

| Format (MIME type) | Usual extension |
|---|---|
| application/x-dvi | .dvi |
| application/x-tar | .tar |
| application/gzip<br>application/x-gzip-compressed | .gz |
| application/zip<br>application/x-zip-compressed | .zip |
| application/x-compressed | .tar.gz, .tgz |

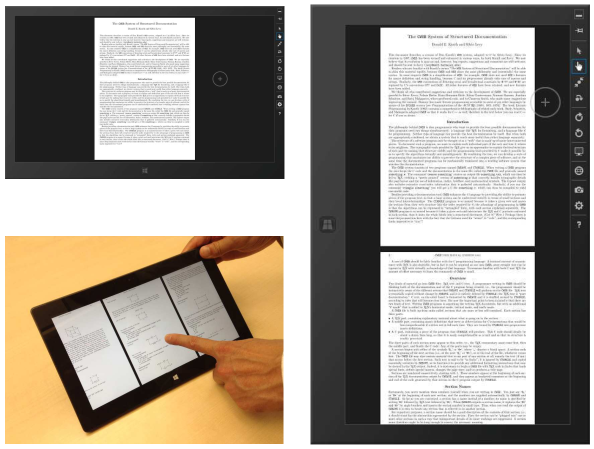


**Figure 5**: Running ModernDvi on a tablet: full touch and rotation support.

### 4.6 Modernity

One of the key aspects of ModernDvi is its recognition of new technologies, devices and interfaces. In recent years, touch screens have heavily changed our habits, and software design had to be significantly overhauled to adapt to such new interfaces. ModernDvi embraces this evolution by providing an innovative, touch-friendly user interface such that it can be similarly used with either a classic mouse-keyboard setup or a touch-enabled device such as a tablet. It is also worth mentioning that ModernDvi is by design compatible with multiple CPU architectures: x86, x64 and ARM. So, computers equipped with (at least) Windows 8 and devices running Windows RT, such as the Microsoft Surface, are all capable of natively running the application. Smartphones running the Windows Phone 8+ operating system can be expected to follow soon due to the common architecture with Windows 8 (NT kernel).

In addition to touch support, ModernDvi has full rotation support: no matter how the user holds the device, the application will update its layout so as to present correctly-oriented content. Figure 5 shows ModernDvi running on a tablet (emulation of an x86 tablet environment). By combining these two features (touch and rotation), the user can conveniently and naturally go through the document as if turning pages of a book by selecting the horizontal display mode and holding his tablet vertically; then, a simple finger sweep will then display the next page.

Finally, ModernDvi has docking support: the user can move ModernDvi onto the side of the screen to interact with another application. This is highly useful for the "Edit-and-preview" use case as detailed in Section 6.1.

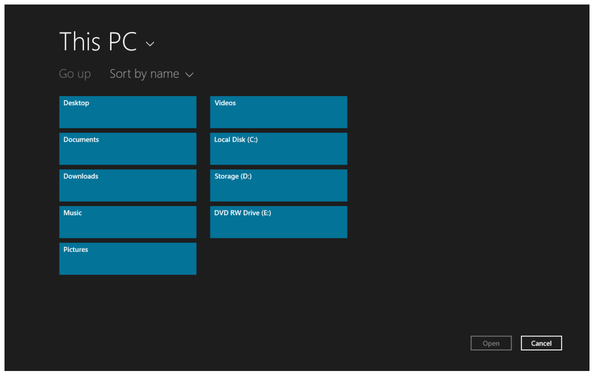


**Figure 6**: A file selection dialog: the only way to access a user's files.

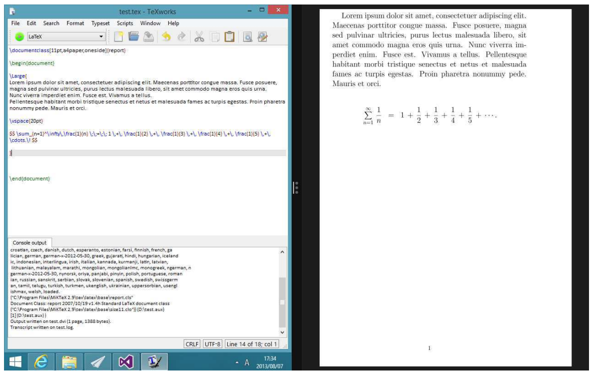
## 5 "App" considerations

Developing a Windows Store application has many advantages compared to a classic, legacy program. First, its distribution, deployment and promotion are greatly facilitated since everything is handled by the operating system through the official Windows Store application (installation, removal, etc.). Also, installing software via the Windows Store is a security guarantee for the user: applications are reviewed before they are added to the Store, and importantly, they run inside a protected environment, ensuring a minimum footprint on the operating system as detailed below.

By design, Windows Store applications (the same is true for Apple Store applications) run in a restricted (sandboxed) environment for security reasons. Thanks to this feature, the user need not worry about system modifications by the app: they are simply not allowed. The application footprint on the system is thus minimal, unlike legacy programs.

One of the main implications of this design is that applications have no direct access to the file system, except for the application's own data folder. To be precise, for an application to perform I/O operations on a file outside the application's data folder, the user must first manually select the file through a file selection dialog control such as a `FileOpenPicker` instance (see Figure 6).

This limitation has a major impact on an application such as ModernDvi. For example, there is no possibility of accessing external files, such as images, which may be referred to in the DVI document. And there is no possibility of running an external program, such as Ghostscript, to delegate PostScript work. A Store application is required to be completely independent. It has thus been a significant challenge to produce a fully functional DVI viewer application.



```
\documentclass[11pt,a4paper,oneside]{report}

\begin{document}

\Large{
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas porttitor congue massa. Fusce posuere,
magna sed pulvinar ultricies, purus lectus malesuada libero, sit amet commodo magna eros quis urna.
Nunc viverra imperdiet enim. Fusce est. Vivamus a tellus.
Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Proin pharetra
nonummy pede. Mauris et orci.

\vspace{20pt}

$$ \sum_{n=1}^\infty\,\frac{1}{n} \;\;=\;\; 1 \,+\, \frac{1}{2} \,+\, \frac{1}{3} \,+\, \frac{1}{4} \,+\, \frac{1}{5} \,+\,
\cdots.\! $$

}


\end{document}
```

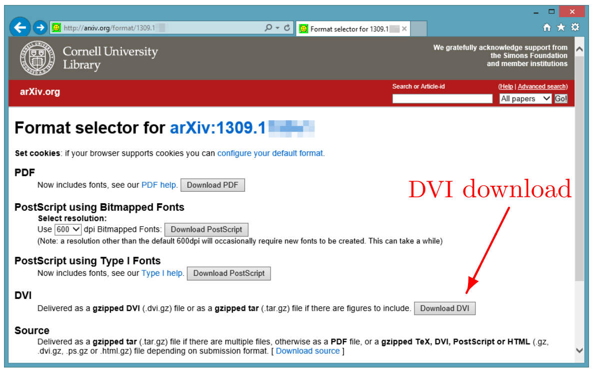
```
croatian, czech, danish, dutch, esperanto, estonian, farsi, finnish, french, ga
lician, german, german-x-2012-05-30, greek, gujarati, hindi, hungarian, iceland
ic, indonesian, interlingua, irish, italian, kannada, kurmanji, latin, latvian,
 lithuanian, malayalam, marathi, mongolian, mongolianlmc, monogreek, ngerman, n
german-x-2012-05-30, nynorsk, oriya, panjabi, pinyin, polish, portuguese, roman
ian, russian, sanskrit, serbian, slovak, slovenian, spanish, swedish, swissgerm
an, tamil, telugu, turkish, turkmen, ukenglish, ukrainian, uppersorbian, usengl
ishmax, welsh, loaded.
("C:\Program Files\MiKTeX 2.9\tex\latex\base\report.cls"
Document Class: report 2007/10/19 v1.4h Standard LaTeX document class
("C:\Program Files\MiKTeX 2.9\tex\latex\base\size11.clo")) (D:\test.aux)
[1] (D:\test.aux) )
Output written on test.dvi (1 page, 1388 bytes).
Transcript written on test.log.
```

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas porttitor congue massa. Fusce posuere, magna sed pulvinar ultricies, purus lectus malesuada libero, sit amet commodo magna eros quis urna. Nunc viverra imperdiet enim. Fusce est. Vivamus a tellus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Proin pharetra nonummy pede. Mauris et orci.

$$\sum_{n=1}^\infty \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots.$$

1

**Figure 7**: Editing (left) and preview (right) of LaTeX source via the Windows 8 screen splitting feature.


http://arxiv.org/format/1309.1[redacted]
Format selector for 1309.1[redacted]
Cornell University Library
arXiv.org
Search or Article-id
(Help | Advanced search)
All papers
Go!
DVI download

We gratefully acknowledge support from the Simons Foundation and member institutions

**Format selector for arXiv:1309.1**[redacted]

**Set cookies**: if your browser supports cookies you can configure your default format.

**PDF**
Now includes fonts, see our PDF help. Download PDF

**PostScript using Bitmapped Fonts**
**Select resolution:**
Use 600 dpi Bitmapped Fonts: Download PostScript
(Note: a resolution other than the default 600dpi will occasionally require new fonts to be created. This can take a while)

**PostScript using Type I Fonts**
Now includes fonts, see our Type I help. Download PostScript

**DVI**
Delivered as a **gzipped DVI** (.dvi.gz) file or as a **gzipped tar** (.tar.gz) file if there are figures to include. Download DVI

**Source**
Delivered as a **gzipped tar** (.tar.gz) file if there are multiple files, otherwise as a **PDF** file, or a **gzipped TeX, DVI, PostScript or HTML** (.gz, .dvi.gz, .ps.gz or .html.gz) file depending on submission format. [ Download source ]

**Figure 8**: Reading a preprint downloaded from arXiv.org.

As an example, let us consider the file monitoring feature of ModernDvi (see Section 4.2). The Windows runtime does have a folder monitoring capability, namely the `ContentsChanged` event of the `StorageFolderQueryResult` class, but because accessing a file object (`StorageFile`), say via a file picker, does not grant permission to access the containing folder (`StorageFolder`), this is unusable for us. So, we implemented a file monitor by using a timer object (`DispatcherTimer`) to check the value of the `DateModified` property of the `Basic Properties` class instance returned by a call to the `GetBasicPropertiesAsync` method on the file on each clock tick. Because of the high timer resolution, this file monitoring solution is fully satisfactory.

## 6 Use cases

This section presents two different use case examples for ModernDvi: edit-and-preview, and reading mode.

### 6.1 Edit-and-preview mode

The user is preparing an article to be submitted to a symposium, using the LaTeX typesetting system. The user opens his LaTeX source file in his favourite editor, say, TeXworks. A first compilation by `latex` is triggered by the user, generating a DVI file. The user double clicks this DVI file to start ModernDvi and load the DVI file into view. As the user wants to continue editing, s/he docks ModernDvi onto the side of the screen, and puts TeXworks on the other side. The screen is thus split into two areas as shown in Figure 7. The user continues to make changes in the LaTeX source file and recompiles the document, still using `latex`. The DVI file generated by the previous run is overwritten by the new one. ModernDvi automatically detects that changes have been made to the DVI file and refreshes its view. The user thus has an instant preview of the changed version.

### 6.2 Reading mode

The user finds a preprint on the online repository arXiv.org, and presses the link to request the file, which is a `.dvi.gz` compressed DVI document. The browser silently decompresses this gzip compressed content and serves the DVI file to the user. The user's device system asks what should be done with the file: open, save, etc.; the user presses the "Open" button, which automatically loads the document into ModernDvi. See Figure 8. The user selects the "window fit" zoom mode for improved readability and can start going through the document.

## 7 Comparing rendering quality

In this section, we compare ModernDvi with other DVI viewers regarding rendering quality. It is difficult to compare usability, including speed, since ModernDvi is targeting a different platform and interface. For each of the comparisons, default settings were used.

### 7.1 vs. YAP

First, let us compare the rendering quality with that of YAP [16]. In both cases, the zoom level is set to match the screen width. We can see in Figure 9 that the rendering quality of ModernDvi is better than that of YAP.

### 7.2 vs. dviout

Then, we compare the rendering quality with that of dviout [14]. In both cases, the zoom level is set to match the screen width. Again, we can see in Figure 9 that the rendering quality of ModernDvi is better than that of dviout.

### 7.3 vs. Microsoft Reader

Lastly, let us compare the rendering quality of ModernDvi with that of the PDF viewer included with

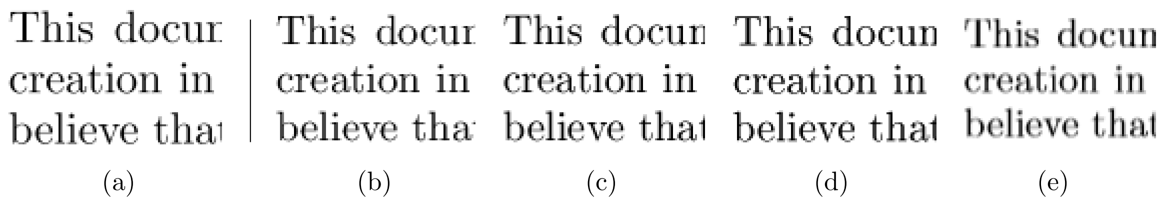


**Figure 9**: Rendering by (a) ModernDvi, (b) YAP (PostScript mode), (c) YAP (PK mode), (d) dviout and (e) Microsoft Reader (PDF).

Windows 8.1 (Microsoft Reader 6.3.9431.0), after converting the DVI to PDF format with the DVIPDFM$x$ utility. In both cases, the zoom level is set to match the screen width. As illustrated by Figure 9, the rendering quality of ModernDvi is significantly better.

## 8 Conclusions

We have presented in this paper ModernDvi, a new DVI document viewer targeting modern platforms and interfaces, such as tablets. After describing the software architecture, design and features, we compared the rendering quality of other viewers; ModernDvi is leading on that point too. Windows Store applications are sandboxed, and thus severely restricted regarding file system access. We have circumvented these challenges to produce a fully functional DVI viewer application.

Future work includes refining our rendering technique by going down to sub-pixels, as well as improving the rendering speed by continuing the work on glyph caching. Lastly, user interface improvements, such as being able to handle multiple documents at once, are also planned.

## A ModernDvi in the Windows Store

ModernDvi can be found in the Windows Store, or directly at `http://apps.microsoft.com/windows/app/moderndvi/bfa836ff-4eb0-454b-ad9e-a2405197f23b`.

## References

[1] Adobe Systems Incorporated. *Adobe Type 1 Font Format*. Reading, Massachusetts: Addison-Wesley, 1990.

[2] Matias Atria. MDVI — A DVI previewer. `http://mdvi.sourceforge.net/`. Accessed August 2013.

[3] Edward Barrett. Porting TEX Live to OpenBSD. *TUGboat*, 29(2):303–304, 2008.

[4] CTAN: The Comprehensive TEX Archive Network. Available fonts. `http://www.ctan.org/tex-archive/fonts`. Accessed August 2013.

[5] Hàn Thế Thành. *Micro-typographic extensions to the TEX typesetting system*. PhD thesis, Masaryk University Brno, October 2000.

[6] Allyn Jackson. From preprints to e-prints: The rise of electronic preprint servers in mathematics. *Notices of the American Mathematical Society*, 49(1):23–32, 2002.

[7] Donald E. Knuth. *The TEXbook*. Reading, Massachusetts: Addison-Wesley, 1984.

[8] Donald E. Knuth. *Metafont: The Program*. Reading, Massachusetts: Addison-Wesley, 1986.

[9] Donald E. Knuth. *The Metafontbook*. Reading, Massachusetts: Addison-Wesley, 1986.

[10] Leslie Lamport. *LATEX: A document preparation system: User's guide and reference*. Reading, Massachusetts: Addison-Wesley Professional, 1994.

[11] Microsoft. Windows Store. `http://windows.microsoft.com/en-us/windows-8/apps`. Accessed October 2013.

[12] MSDN. StorageFile.GetBasicPropertiesAsync. `http://msdn.microsoft.com/en-us/library/windows/apps/windows.storage.storagefile.getbasicpropertiesasync.aspx`. Accessed August 2013.

[13] Take-Yuki Nagao. Automatic recognition of theorem environments of mathematical papers in LATEX format. *Bulletin of Advanced Institute of Industrial Technology*, 7:81–87, 2013.

[14] Toshio Oshima, Yoshiki Otobe, and Kazunori Asayama. dviout — a DVI previewer for Windows. `http://www.ctan.org/pkg/dviout`. Accessed October 2013.

[15] Fabrice Popineau. windvi — MS-Windows DVI viewer. `http://www.ctan.org/pkg/windvi`. Accessed August 2013.

[16] Christian Schenk. Yet Another Previewer. `http://www.miktex.org`. Accessed October 2013.

[17] Paul Vojta. xdvi — A DVI previewer for the X Window System. `http://math.berkeley.edu/~vojta/xdvi.html`. Accessed October 2013.

◇ Antoine Bossard and Takeyuki Nagao
Big Data Laboratory
Advanced Institute of Industrial Technology
1-10-40 Higashiooi
Shinagawa-ku, 140-0011
Japan
`abossard (at) aiit dot ac dot jp`
`nagao-takeyuki (at) aiit dot ac dot jp`

## Selection in PDF viewers and a LuaTEX bug

Hans Hagen

In January 2014 a message was posted to the ConTEXt mailing list asking for clarification about the way PDF viewers select text. Let me give an example of that (inside a convenient ConTEXt wrapper):

```
\startTEXpage[offset=2cm]
  \hbox{$ x+y $}
\stopTEXpage
```

In figure 1 you can see how for instance Acrobat (which I use for proofing) and SumatraPDF (which I use in my edit–preview cycle) select this text. As reported in the mail, other viewers behave like SumatraPDF does, with excessive vertical space included in the selection.

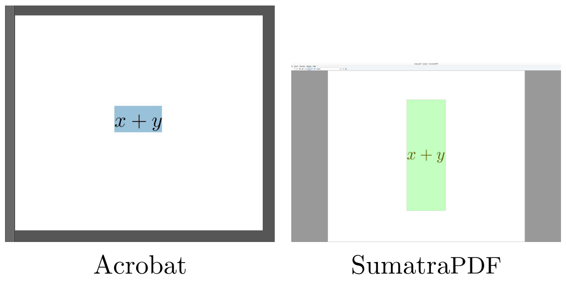


**Figure 1**: Some math selected in PDF viewers.

Part of the question was why wrapping a display formula in an \hbox doesn't have this effect on viewers. Think of:

```
\startTEXpage[offset=2cm]
  \hbox{\startformula x+y \stopformula}
\stopTEXpage
```

This can be reduced to the following primitive construct (which is rendered in figure 2):

```
\startTEXpage[offset=2cm]
  \hbox{$$ x+y $$}
\stopTEXpage
```

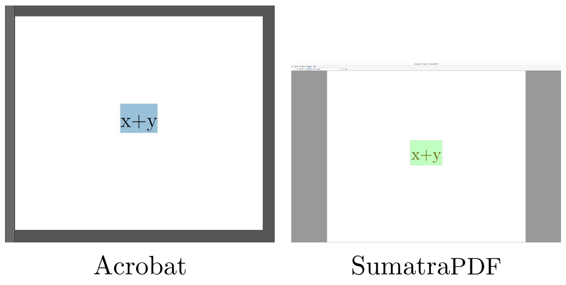


**Figure 2**: Some text selected in PDF viewers.

If you look closely you will see that we have text and not math. This is because in restricted horizontal mode (inside \hbox) TEX sees the $$ as a begin and immediate end of math mode, so in fact we have here some text surrounded by empty math. When I realized that using ConTEXt's \startformula could have this side effect in some situations I decided to catch this, but sometimes TEX can give surprises.

Already a while ago Taco and I decided that it would be handy to have primitives in LuaTEX for special characters (or more precisely: characters with certain catcodes) as used in math and alignments.

**Table 1**: Some of the extra LuaTEX primitives.

| token | primitive(s) | |
|---|---|---|
| # | \alignmark | |
| & | \aligntab | |
| $ | \Ustartmath | \Ustopmath |
| $$ | \Ustartdisplaymath | \Ustopdisplaymath |
| ^ | \Usuperscript | |
| _ | \Usubscript | |

As you can see in table 1, the dollars are somewhat special as in fact we don't alias characters (tokens) but have introduced primitives that change the mode from text to inline or display math and back. So, we can say:

```
\startTEXpage[offset=2cm]
 \hbox{\Ustartdisplaymath x+y \Ustopdisplaymath}
\stopTEXpage
```

This renders okay in LuaTEX 0.78.1, but when we tinker a bit like this (with an invalid \par inside the \hbox):

```
\startTEXpage[offset=2cm]
  \hbox{\Ustartdisplaymath x+y \Ustopdisplaymath
        \par}
\stopTEXpage
```

we get this:

```
Assertion failed!

Program: c:\tex\texmf-win64\bin\luatex.exe
File: web2c/luatexdir/tex/texnodes.w, Line 830

Expression: p> my_prealloc
```

Oops. Furthermore, if we add some text after the invalid \par:

```
\startTEXpage[offset=2cm]
  \hbox{\Ustartdisplaymath x+y \Ustopdisplaymath
        \par x}
\stopTEXpage
```

we get:

```
! This can't happen (vpack).
```

As an excursion from working on the Critical Editions project Luigi Scarso and I immediately started debugging this at the engine level and after some tracing we saw that it had to do with packaging. Taco joined in, and we decided that it made no sense at all to try to deal with this at that level simply because we ourselves had bypassed a natural boundary of TEX: caching the start of display math by seeing two successive `$` as an inline formula. So the solution is either to make `\Ustartdisplaymath` more clever, but better is to simply issue an error message when this state is entered. That way we stick to the original TEX point of view, an approach that has never failed us so far. The chosen solution is to issue error messages (broken onto two lines for *TUGboat*):

```
! You can't use `\Ustartdisplaymath'
  in restricted horizontal mode

! You can't use `\Ustopdisplaymath'
  in restricted horizontal mode
```

If you really want this you can redefine the primitive:

```
\let\normalUstartmath\Ustartmath
\let\normalUstopmath \Ustopmath

\let\normalUstartdisplaymath\Ustartdisplaymath
\let\normalUstopdisplaymath \Ustopdisplaymath

\unexpanded\def\Ustartdisplaymath % context way
 {\ifinner
   \ifhmode
     \normalUstartmath
     \let\Ustopdisplaymath
         \normalUstopmath
   \else
     \normalUstartdisplaymath
     \let\Ustopdisplaymath
         \normalUstopdisplaymath
   \fi
  \else
    \normalUstartdisplaymath
    \let\Ustopdisplaymath
        \normalUstopdisplaymath
  \fi}
```

As with many things in TEX there is often a way out, as long as things are open and accessible enough. Currently in ConTEXt we do something like the above for cases where confusion can happen.

With that fixed, it was time to return to the original question. Why do math selections have such large bounding boxes in some viewers? The answer to that is in the PDF file. Let's look at the font properties of a math font, Latin Modern Math here:

```
24 0 obj
  <<
    /Type        /FontDescriptor
    /FontName    /CXDZIF+LatinModernMath-Regular
    /Flags       4
    /FontBBox    [-1042 -3060 4082 3560]
    /Ascent      3560
    /CapHeight   683
    /Descent     -3060
    /ItalicAngle 0
    /StemV       93
    /XHeight     431
    /FontFile3   23 0 R
    /CIDSet      22 0 R
  >>
endobj
```

Compare these rather large values for `FontBBox`, `Ascent`, etc., with a text font (Latin Modern Roman):

```
24 0 obj
  <<
    /Type        /FontDescriptor
    /FontName    /TEKCPF+LMRoman12-Regular
    /Flags       4
    /FontBBox    [-422 -280 1394 1127]
    /Ascent      1127
    /CapHeight   683
    /Descent     -280
    /ItalicAngle 0
    /StemV       91
    /XHeight     431
    /FontFile3   23 0 R
    /CIDSet      22 0 R
  >>
endobj
```

So, it looks like Acrobat is using the actual heights and depths of glyphs (probably with some slack) while other viewers use the font's ascender and descender values. So in the end the answer is: there is nothing the user, ConTEXt or LuaTEX can do about it, apart from messing with the values above, which is probably not a good idea.

But trying to answer the question (by stripping down, etc.) had the side effect of identifying a bug in LuaTEX. A lesson learned is thus that even adding simple primitives like the ones above needs some studying of the source code in order to identify side effects. We should have known!

◇ Hans Hagen
Pragma ADE
http://pragma-ade.com

# Parsers in TEX and using CWEB for general pretty-printing

Alexander Shibakov

In this article I describe a collection of TEX macros and a few simple C programs called SPLinT that enable the use of the standard parser and scanner generator tools, bison and flex, to produce very general parsers and scanners coded as TEX macros. SPLinT is freely available from http://ctan.org/pkg/splint and http://math.tntech.edu/alex.

## Introduction

The need to process formally structured languages inside TEX documents is neither new nor uncommon. Several graphics extensions for TEX (and LATEX) have introduced a variety of small specialized languages for their purposes that depend on simple (and not so simple) interpreters coded as TEX macros. A number of *pretty-printing* macros take advantage of different parsing techniques to achieve their goals (see [Go], [Do], and [Wo]).

Efforts to create general and robust parsing frameworks inside TEX go back to the origins of TEX itself. A well-known BASIC subset interpreter, BASIX (see [Gr]) was written as a demonstration of the flexibility of TEX as a programming language and a showcase of TEX's ability to handle a variety of abstract data structures. On the other hand, a relatively recent addition to the LATEX toolbox, l3regex (see [La]), provides a powerful and very general way to perform regular expression matching in LATEX, which can be used (among other things) to design parsers and scanners.

Paper [Go] contains a very good overview of several approaches to parsing and tokenizing in TEX and outlines a universal framework for parser design using TEX macros. In an earlier article (see [Wo]), Marcin Woliński describes a parser creation suite paralleling the technique used by CWEB (CWEB's 'grammar' is hard-coded into CWEAVE, whereas Woliński's approach is more general). One commonality between these two methods is a highly customized tokenizer (or scanner) used as the input to the parser proper. Woliński's design uses a finite automaton as the scanner engine with a 'manually' designed set of states. No backing up mechanism was provided, so matching, say, the longest input would require some custom coding (it is, perhaps, worth mentioning here that a backup mechanism is all one needs to turn any regular language *scanner* into a general CWEB-type parser). The scanner in [Go] was designed mainly with efficiency in mind and thus relies on a number of very clever techniques that are highly language-specific (out of necessity).

Since TEX is a system well-suited for typesetting technical documents, *pretty-printing* texts written in formal languages is a common task and is also one of the primary reasons to consider a parser written in TEX.

The author's initial motivation for writing the software described in this article grew out of the desire to fully document a few embedded microcontroller projects that contain a mix of C code, Makefiles, linker scripts, etc. While the majority of code for such projects is written in C, superbly processed by CWEB itself, some crucial information resides in the kinds of files mentioned above and can only be handled by CWEB's verbatim output (with some minimal postprocessing, mainly to remove the #line directives left by CTANGLE).

## Parsing with TEX vs. others

Naturally, using TEX in isolation is not the only way to produce pretty-printed output. The CWEB system for writing structured documentation uses TEX merely as a typesetting engine, while handing over the job of parsing and preprocessing the user's input to a program built specifically for that purpose. Sometimes, however, a paper or a book written in TEX contains a few short examples of programs written in another programming language. Using a system such as CWEB to process these fragments is certainly possible (although it may become somewhat involved) but a more natural approach would be to create a parser that can process such texts (with some minimal help from the user) entirely inside TEX itself. As an example, pascal (see [Go]) was created to pretty-print Pascal programs using TEX. It used a custom scanner for a subset of standard Pascal and a parser, generated from an LL(1) Pascal grammar by a parser generator, called parTEX.

Even if CWEB or a similar tool is used, there may still be a need to parse a formal language inside TEX. One example would be the use of CWEB to handle a language other than C.

Before I proceed with the description of the tool that is the main subject of this paper, allow me to pause for just a few moments to discuss the *wisdom* (or *lack* thereof) of laying the task of parsing formal texts entirely on TEX's shoulders. In addition to using an external program to preprocess a TEX document, some recent developments allow one to implement a parser in a language 'meant for such tasks' inside an extension of TEX. We are speaking of course about LuaTEX (see [Ha]) that essentially implements an entirely separate interface to TEX's typesetting mechanisms and data structures in Lua (see [Lu]), 'grafted' onto a TEX extension.

Although I feel nothing but admiration for the LuaTEX developers, and completely share their desire to empower TEX by providing a general purpose programming language on top of its internal mechanisms, I would like to present three reasons to avoid taking advantage of LuaTEX's impressive capabilities for this particular task.

First, I am unaware of any standard tools for generating parsers and scanners in Lua (of course, it would be just as easy to use the approach described here to create such tools). At this point in time, it is just as easy to coax standard parser generators into outputting parsers in TEX as it is to make them output Lua.

Second, I am a great believer in generating 'archival quality' documents: standard TEX has been around for almost three decades in a practically unchanged form, an eternity in the software world. The parser produced using the methods outlined in this paper uses standard (plain) TEX exclusively. Moreover, if the grammar is unchanged, the parser code itself (i.e. its semantic actions) is very readable, and can be easily modified without going through the whole pipeline (`bison`, `flex`, etc.) again. A full record of the grammar is left with the generated parser and scanner so even if the more 'volatile' tools, such as `bison` and `flex`, become incompatible with the package, the parser can still be utilized with TEX alone. Perhaps the following quote by D. Knuth (see [DEK2]) would help to reinforce this point of view: "*Of course I do not claim to have found the best solution to every problem. I simply claim that it is a great advantage to have a fixed point as a building block.*"

Finally, the idea that TEX is somehow unsuitable for such tasks may have been overstated. While it is true that TEX's macro language lacks some of the expressive ability of its 'general purpose' brethren, it does possess a few qualities that make it quite adept at processing text (it is a typesetting language after all!). Among these features are: a built-in hashing mechanism (accessible through `\csname...\endcsname` and `\string` primitives) for storing and accessing control sequence names and creating associative arrays, a number of tools and data structures for comparing and manipulating strings (token registers, the `\ifx` primitive, various expansion primitives: `\edef`, `\expandafter` and the like), and even string matching and replacement (using delimited parameters in macros). TEX notoriously lacks a good (i.e. efficient *and* easy to use) framework for storing and manipulating *arrays* and *lists* (see the discussion of list macros in Appendix D of *The TEXbook* and in [Gr]) but this limitation is readily overcome by putting some extra care into one's macros.

## Languages, grammars, parsers, and TEX

Or ...

*Tokens and tables keep macros in check.*
*Make 'em with* `bison`*, use* `WEAVE` *as a tool.*
*Add TEX and* `CTANGLE`*, and* C *to the pool.*
*Reduce 'em with actions, look forward, not back.*
*Macros, productions, recursion and stack!*

Computer generated (most likely)

The goal of the software described in this article, `SPLinT` (Simple Parsing and Lexing in TEX, or, in the tradition of GNU, `SPLinT` Parses Languages in TEX) is to give a macro writer the ability to use standard parser/scanner generator tools to produce TEX macros capable of parsing formal languages.

Let me begin by presenting a 'bird's eye view' of the setup and the workflow one would follow to create a new parser with this package. To take full advantage of this software, two external programs (three if one counts a C compiler) are required: `bison` and `flex` (see [Bi] and [Pa]), the parser and scanner generators, respectively. Both are freely available under the terms of the General Public License version 3 or higher and are standard tools included in practically every modern GNU/Linux distribution. Versions that run under a number of other operating systems exist as well.

While the software allows the creation of both parsers and scanners in TEX, the steps involved in making a scanner are very similar to those required to generate a parser, so only the parser generation will be described below.

Setting the semantic actions aside for the moment, one begins by preparing a generic `bison` input file, following some simple guidelines. Not all `bison` options are supported (the most glaring omission is the ability to generate a general LR (glr) parser but this may be added in the future) but in every other respect it is an ordinary `bison` grammar. In some cases, a `bison` grammar may already exist and can be turned into a TEX parser with just a few (or none!) modifications and a new set of semantic actions (written in TEX of course). As a matter of example, the grammar used to pretty-print `bison` grammars in `CWEB` that comes with this package was adopted (with very minor modifications, mainly to create a more logical presentation in `CWEB`) from the original grammar used by `bison` itself.

Once the grammar has been debugged (using a combination of `bison`'s own impressive debugging facilities and the debugging features supported by the macros in the package), it is time to write the semantic actions for the *syntax-directed translation* (see [Ah]). These are ordinary TEX macros written using a few simple conventions listed below. First, the actions themselves will be executed inside a large `\ifcase` statement (this is not *always* the case, see the discussion of 'optimization' below, but it would be better to assume that it is); thus, care must be taken to write the macros so that they can be 'skipped' by TEX's scanning mechanism. Second, instead of using `bison`'s `$`*n* syntax to access the value stack, a variety of `\yy`*p* macros are provided. Finally, the 'driver' (a small C program, see below) provided with the package merely cycles through the actions to output TEX macros, so one has to use one of the C macros provided with the package to output TEX in a proper form. One such macro is `TeX_`, used as `TeX_("{`TEX tokens`}");`.

The next step is the most technical, and the one most in need of automation. A `Makefile` provided with the package shows how such automation can be achieved. The newly generated parser (the '`.c`-file' produced by `bison`) is `#include`'d in (yes, included, not merely linked to) a special 'driver' file. No modifications to the driver file or the `bison` produced parser are necessary; all one has to do is call a C compiler with an appropriately defined macro (see the `Makefile` for details). The resulting executable is then run which produces a `.tex` file that contains the macros necessary to use the freshly-minted parser in TEX. This short brush with a C compiler is the only time one ventures outside of the world of pure TEX to build a parser with this software (not counting the one needed to create the accompanying scanner if one is desired). It is possible to add a 'plugin' to `bison` to create a 'TEX output mode' but at the moment the 'lazy' version seems to be sufficient.

Now `\input` this file into your TEX document along with the macros that come with the package and voilà! You have a brand new parser in TEX! A full-featured parser for the `bison` input file format is included, and can be used as a template. For smaller projects, it might help to take a look at the examples portion of the package.

The discussion above glosses over a few important details that anybody who has experience writing 'ordinary' (i.e. non-TEX) parsers in `bison` would be eager to find out. Let us now discuss some of these details.

## Data structures for parsing

A surprisingly modest amount of machinery is required to make a `bison`-generated parser 'tick'. In addition to the standard arithmetic 'bag of tricks' (integer addition, multiplication and conditionals), some basic integer and string array (or closely related list and stack) manipulation is all that is needed.

Parser tables and stack access 'in the raw' are normally hidden from the parser designer but creating lists and arrays is standard fare for most semantic actions. The `bison` parser supplied with the package does not use any techniques that are more sophisticated than simple token register operations. Representing and accessing arrays this way (see Appendix D of *The TEXbook* or the `\concat` macro in the package) is simple and intuitive but computationally expensive. The computational costs are not prohibitive though, as long as the arrays are kept short. In the case of large arrays that are read often, it pays to use a different mechanism. One such technique (used also in [Go], [Gr], and [Wo]) is to 'split' the array into a number of control sequences (creating an *associative array* of token sequences called, for example `\array[`*n*`]`, where *n* is an index value). This approach is used with the parser and scanner tables (which tend to be quite large) when the parser is 'optimized' (more about this later). Once again, it is possible to write the parser semantic actions without this (slightly unintuitive and cumbersome to implement) machinery.

This covers most of the routine computations inside semantic actions; all that is left is a way to 'tap' into the stack automaton built by `bison` using an interface similar to the special `$`*n* variables utilized by the 'genuine' `bison` parsers (i.e. written in C or any other target language supported by `bison`).

This role is played by the several varieties of `\yy`*p* command sequences (for the sake of completeness, *p* stands for one of `(`*n*`)`, `[`name`]`, `]`name`[` or *n*; here *n* is a string of digits; and a 'name' is any name acceptable as a symbolic name for the term in `bison`). Instead of going into the minutiae of various flavors of `\yy`-macros, let me just mention that one can get by with only two 'idioms' and still be able to write parsers of arbitrary sophistication: `\yy(`*n*`)` can be treated as a token register containing the value of the *n*-th term of the rule's right hand side, $n > 0$. The left hand side of a production is accessed through `\yyval`. A convenient shortcut is `\yy0{`⟨*TEX material*⟩`}` which will expand the ⟨*TEX material*⟩ inside the braces. Thus, a simple way to concatenate the values of the first two production terms is `\yy0{\the\yy(1)\the\yy(2)}`. The included `bison` parser can also be used to provide support for 'symbolic names', analogous to `bison`'s `$[name]` syntax but this requires a bit more effort on the user's part in order to initialize such support. It could make the parser more readable and maintainable, however.

Naturally, a parser writer may need a number of other data abstractions to complete the task. Since these are highly dependent on the nature of the processing the parser is supposed to provide, we refer the interested reader to the parsers included in the package as a source of examples of such specialized data structures.

## Pretty-printing support with formatting hints

The scanner 'engine' is propelled by the same set of data structures and operations that drive the parser automaton: stacks, lists and the like. Table manipulation happens 'behind the scenes' just as in the case of the parser. There is also a stack of 'states' (more properly called *subautomata*) that is manipulated by the user directly, where the access to the stack is coded as a set of macros very similar to the corresponding C functions in the 'real' `flex` scanners. The 'handoff' from the scanner to the parser is implemented through a pair of registers: `\yylval`, a token register containing the value of the returned token and `\yychar`, a `\count` register that contains the numerical value of the token to be returned.

Upon matching a token, the scanner passes one crucial piece of information to its user: the character sequence representing the token just matched (`\yytext`). This is not the whole story, though: three more token sequences are made available to the parser writer whenever a token is matched.

The first of these is simply a 'normalized' version of `\yytext` (called `\yytextpure`). In most cases it is a sequence of TeX tokens with the same character codes as the one in `\yytext` but with their category codes set to 11. In cases when the tokens in `\yytext` are *not* (character code, category code) pairs, a few simple conventions are followed, explained elsewhere. This sequence is provided merely for convenience and its typical use is to generate a key for an associative array.

The other two sequences are special 'stream pointers' that provide access to the extended scanner mechanism in order to implement passing of 'formatting hints' to the parser without introducing any changes to the original grammar, as explained below.

Unlike strict parsers employed by most compilers, a parser designed for pretty-printing cannot afford being too picky about the structure of its input ([Go] calls such parsers 'loose'). As a way of simple illustration, an isolated identifier, such as '`lg_integer`' can be a type name, a variable name, or a structure tag (in a language like C for example). If one expects the pretty-printer to typeset this identifier in a correct style, some context must be supplied, as well. There are several strategies a pretty-printer can employ to get hold of the necessary context. Perhaps the simplest way to handle this, and to reduce the complexity of the pretty-printing algorithm, is to insist on the user providing enough context for the parser to do its job. For short examples like the one above, this is an acceptable strategy. Unfortunately, it is easy to come up with longer snippets of grammatically deficient text that a pretty-printer should be expected to handle. Some pretty-printers, such as the one employed by `CWEB` and its ilk (`WEB`, `FWEB`), use a very flexible bottom-up technique that tries to make sense of as large a portion of the text as it can before outputting the result (see also [Wo], which implements a similar algorithm in LaTeX).

The expectation is that this algorithm will handle the majority of the cases with the remaining few left for the author to correct. The question is, how can such a correction be applied?

`CWEB` itself provides two rather different mechanisms for handling these exceptions. The first uses direct typesetting commands (for example, `@+` and `@*` for cancelling and introducing a line break, resp.) to change the typographic output.

The second (preferred) way is to supply *hidden context* to the pretty-printer. Two commands, `@;` and `@[...@]` are used for this purpose. The former introduces a 'virtual semicolon' that acts in every way like a real one except it is not typeset (it is not output in the source file generated by `CTANGLE`, either but this has nothing to do with pretty-printing, so I will not mention `CTANGLE` anymore). For instance, from the parser's point of view, if the preceding text was parsed as a 'scrap' of type *exp*, the addition of `@;` will make it into a 'scrap' of type *stmt* in `CWEB`'s parlance. The latter construct (`@[...@]`), is used to create an *exp* scrap out of whatever happens to be inside the brackets.

This is a powerful tool at one's disposal. Stylistically, this is the right way to handle exceptions as it forces the writer to emphasize the *logical* structure of the formal text. If the pretty-printing style is changed extensively later, the texts with such hidden contexts should be able to survive intact in the final document (as an example, using a break after every statement in C may no longer be considered appropriate, so any forced break introduced to support this convention would now have to be removed, whereas `@;`'s would simply quietly disappear into the background).

The same hidden context idea has another important advantage: with careful grammar fragmenting (facilitated by `CWEB`'s or any other literate programming tool's 'hypertext' structure) and a more diverse hidden context (or even arbitrary hidden text) mechanism, it is possible to use a strict parser to parse incomplete language fragments. For example, the productions that are needed to parse C's expressions form a complete subset of the parser. If the grammar's 'start' symbol is changed to *expression* (instead of the *translation-unit* as it is in the full C grammar), a variety of incomplete C fragments can now be parsed and pretty-printed. Whenever such granularity is still too 'coarse', carefully supplied hidden context will give the pretty-printer enough information to adequately process each fragment. A number of such *sub*-parsers can be tried on each fragment (this may sound computationally expensive, however, in practice, a carefully chosen hierarchy of parsers will finish the job rather quickly) until a correct parser produced the desired output.

This somewhat lengthy discussion brings us to the question directly related to the tools described in this article: how does one provide typographical hints or hidden context to the parser?

One obvious solution is to build such hints directly into the grammar. The parser designer can, for instance, add new tokens (terminals, say, `BREAK_LINE`) to the grammar and extend the production set to incorporate the new additions. The risk of introducing new conflicts into the grammar is low (although not entirely non-existent, due to the lookahead limitations of LR(1) grammars) and the changes required are easy, although very tedious, to incorporate.

In addition to being labor intensive, this solution has two other significant shortcomings: it alters the original grammar and hides its logical structure, and it 'bakes in' the pretty-printing conventions into the language structure (making 'hidden' context much less 'stealthy').

A much better approach involves inserting the hints at the lexing stage and passing this information to the parser as part of the token 'values'. The hints themselves can masquerade as characters ignored by the scanner (white space, for example) and preprocessed by a specially designed input routine. The scanner then simply passes on the values to the parser.

The difficulty lies in synchronizing the token production with the parser. This subtle complication is very familiar to anyone who has designed TeX's output routines: the parser and the lexer are not synchronous, in the sense that the scanner might be reading several (in the case of the general LR($n$) parsers) tokens ahead of the parser before deciding on how to proceed (the same way TeX can consume a whole paragraph's worth of text before exercising its page builder).

If we simple-mindedly let the scanner return every hint it has encountered so far, we may end up feeding the parser the hints meant for the token that appears *after* the fragment the parser is currently working on. In other words, when the scanner 'backs up' it must correctly back up the hints as well.

This is exactly what the scanner produced by the tools in this package does: along with the main stream of tokens meant for the parser, it produces two hidden streams (called the `\format` stream and the `\stash` stream) and provides the parser with two strings (currently only strings of digits are used although arbitrary sequences of TeX tokens can be used as pointers) with the promise that *all the 'hints' between the beginning of the corresponding stream and the point labelled by the current stream pointer appeared among the characters up to and, possibly, including the ones matched as the current token.* The macros to extract the relevant parts of the streams (`\yyreadfifo` and its cousins) are provided for the convenience of the parser designer. The interested reader can consult the input routine macros for the details of the internal representation of the streams.

In the interest of full disclosure, let me point out that this simple technique introduces a significant strain on TeX's computational resources: the lowest level macros, the ones that handle character input and are thus executed (sometimes multiple times), for *every* character in the input stream are rather complicated and therefore, slow. Whenever the use of such streams is not desired a simpler input routine can be written to speed up the process (see `\yyinputtrivial` for a working example of such macro).

## The parser function

To achieve such a tight integration with `bison`, its parser template, `yyparse()` was simply translated into TeX using the following well known method.

Given the code (where `goto`'s are the only means of branching but can appear inside conditionals):

```
label A: ...
     [more code ...]
          goto C;
     [more code ...]
label B: ...
     [more code ...]
          goto A;
     [more code ...]
label C: ...
     [more code ...]
```

one way to translate it into TeX is to define a set of macros (call them `\labelA`, `\labelAtail` and so forth for clarity) that end in `\next` (a common name for this purpose). Now, `\labelA` will implement the code that comes between `label A:` and `goto C;`, whereas `\labelAtail` is responsible for the code after `goto C;` and before `label B:` (provided no other `goto`'s intervene which can always be arranged). The conditional preceding `goto C;` can now be written in TeX as

```
\if⟨condition⟩
    \let\next=\labelC
\else
    \let\next=\labelAtail
```

where ⟨condition⟩ is an appropriate translation of the corresponding condition in the code being translated (usually, one of '$=$' or '$\neq$'). Further details can be extracted from the TeX code that implements these functions where the corresponding C code is presented alongside the macros that mimic its functionality.

## Debugging

If the tools in the package are used to create medium to high complexity parsers, the question of debugging will come up sooner or later. The grammar design stage of this process can utilize all the excellent debugging facilities provided by `bison` and `flex` (reporting of conflicts, output of the automaton, etc.). The `Makefile`s supplied with the package will automatically output all the debugging information the corresponding tool can provide. Eventually, when all the conflicts are ironed out and the parser begins to process input without performing any actions, it becomes important to have a way to see 'inside' the parsing process. Since the processing performed by the generated parser is done in several stages, the debugging may become rather involved.

All the debugging features are activated by using various `\iftrace`... conditionals, as well as `\ifyyinputdebug` and `\ifyyflexdebug` (for example, to look at the parser stack, one would set `\tracestackstrue`). When all of the conditionals are activated, *a lot* of output is produced. At this point it is important to narrow down the problem area and only activate the debugging features relevant to any errant behaviour exhibited by the parser. Most of the debugging features built into ordinary `bison` parsers (and `flex` scanners) are available.

In general, debugging parsers and scanners (and debugging in general) is a very deep topic that may require a separate paper (or maybe a book!) all by itself, so I will simply leave it here and encourage the reader to experiment with the included parsers to learn the general operational principles behind the parsing automaton. One needs to be aware that, unlike the 'real' C parsers, the TeX parser has to deal with more than simply straight text. So if it looks like the parser (or the scanner) absolutely has to accept the (rejected) input displayed on the screen, just remember that an 'a' with a category code 11 and an 'a' with a category code 12 look the same on the terminal while TeX and the parser/scanner may treat them as completely different characters (this behavior itself can be fine tuned by changing `\yyinput`).

## Speeding up the parser

By default, the generated parser and scanner keep all of their tables in separate token registers. Each stack is kept in a single macro. Thus, every time a table is accessed, it has to be expanded making the table access latency linear in *the size of the table*. The same holds for stacks and the action 'switches', of course. While keeping the parser tables (that are constant) in token registers does not have any better rationale than saving control sequence memory (the most abundant memory in TeX), this way of storing *stacks* does have an advantage when multiple parsers come into play simultaneously. All one has to do to switch from one parser to another is to save the state by renaming the stack control sequences accordingly.

When the parser and scanner are 'optimized' (by saying `\def\optimization{5}`, for example), all these control sequences are 'spread over' the appropriate associative arrays (by creating a number of control sequences that look like `\array[`$n$`]`, where *n* is the index, as explained above). While it is certainly possible to optimize only some of the parsers (if your document uses multiple) or even only some *parts* of a given parser (or scanner), the details of how to do this are rather technical and are left for the reader to discover by reading the examples supplied with the package. At least at the beginning it is easier to simply set the highest optimization level and use it consistently throughout the document.

### Use with CWEB

Since the macros in the package were designed to support pretty-printing of languages other than C in CWEB it makes sense to spend a few paragraphs on this particular application. The CWEB system consists of two weakly related programs: CWEAVE and CTANGLE. The latter extracts the C portion of the users input, and outputs a C file after an appropriate rearrangement of the various sections of the code. The task of CWEAVE is very different and arguably more complex: not only does it have to be aware of the general 'hierarchy' of various subsections of the program to create cross references, an index, etc., it also has to understand enough of the C code in order to pretty-print it. Whereas CTANGLE simply separates the program code from the programmer's documentation, rearranges it and outputs the original program text (with added #line directives and simple C comments that can be easily removed in postprocessing if necessary), the output of CWEAVE bears very little resemblance to the original program. It might sound a bit exaggerated but CWEAVE's processing is 'one-way': it would be difficult or even impossible to write software that 'undoes' the pretty-printing performed by CWEAVE.

There is, however, a loophole that allows one to use CWEB with practically any language, *and* pretty-print the results, if an appropriate 'filter' is available. The saving grace comes in the form of CWEB's *verbatim output*: any text inside @= and @> undergoes some minimal processing (mainly to 'escape' dangerous TEX characters such as '$') and is put inside \vb{...} by CWEAVE.

The macros in the package take advantage of this feature by collecting all the text inside \vb groups and trying to parse it. If the parsing pass is successful, pretty-printed output is produced, if not, the text is output in 'typewriter' style.

With languages such as bison's input script, an additional complication has to be dealt with: most of the time the actions are written in C so it makes sense to use CWEAVE's C pretty-printer to typeset the action code. Most material outside of \vb groups is therefore assumed to be C code and is carefully collected and 'cleaned up' by the macros included in the package.

For the purposes of documenting the TEX parser, one additional feature of CWEAVE is taken advantage of: the text inside double quotes, "..." is treated similarly to the verbatim portion of the input (this can be viewed as a 'local' version of the verbatim sections). Moreover, CWEAVE allows one to treat a function name (or nearly any identifier) as a TEX macro. These two features are enough to implement pretty-printing of semantic actions in TEX. The macros will turn an input string such as, e.g. 'TeX_( "\\relax" );' into '∘' (for the sake of convenience, the string above would actually be written as 'TeX_( "/relax" );' as explained in the manual for the package). See the documentation that comes with the package and the bison language pretty-printer implementation for any additional details.

### An example

As an example, let us walk through the development process of a simple parser. Since the language itself is not of any particular importance, a simple grammar for additive expressions was chosen. The example, with a detailed description, and all the necessary files, is included in the examples directory. The Makefile there allows one to type, say, make step1 to produce all the files needed in the first step of this short walk-through. Finally, make docs will produce a pretty-printed version of the grammar, the regular expressions, and the test TEX file along with detailed explanations of every stage.

As the first step, one creates a bison input file (expp.y) and a similar input for flex (expl.l). A typical fragment of expp.y looks like the following:

```
value:
  expression {TeX_("/yy0{/the/yy(1)}");}
;
```

The scanner's regular expression section, in its entirety is:

```
[ \f\n\t\v]  {TeX_("/yylexnext");}
{id}         {
    TeX_("/yylexreturnval{IDENTIFIER}");}
{int}        {
    TeX_("/yylexreturnval{INTEGER}");}
[+*()]       {TeX_("/yylexreturnchar");}
.            {
TeX_("/iftracebadchars");
TeX_("   /yycomplain{%%");
TeX_("     invalid character(s): %%");
```

```
  TeX_("      /the/yytext}");
  TeX_("/fi");
  TeX_("/yylexreturn{$undefined}");
  }
```

Once the files have been prepared and debugged, the next step is to generate the 'driver' files, `ptabout` and `ltabout`. For the parser 'driver', this is done with

```
bison expp.y -o expp.c
gcc -DPARSER_FILE=\
    \"examples/expression/expp.c\" \
    -o ptabout ../../mkeparser.c
```

The first line generates the parser from the `bison` input file that was prepared in the first step and the next line uses this file to produce a 'driver'. If the included `Makefile` is used, the file `mkeparser.c` is generated automatically, otherwise one has to make sure that it exists and resides in the appropriate directory first. It has no external dependencies and can be freely moved to any place that is convenient.

Next, run `ptabout` and `ltabout` to produce the automata tables:

```
ptabout --optimize-actions ptab.tex
ltabout --optimize-actions ltab.tex
```

Now, look inside `expression.sty` for a way to include the parser in your own documents, or simply `\input` it in your own TeX file. Executing `make test.tex` will produce a test file for the new parser. This is it!

## Acknowledgment

The author would like to thank the editors, Barbara Beeton and Karl Berry, for a number of valuable suggestions and improvements to this article.

## References

[Ah] Alfred V. Aho et al., *Compilers: Principles, Techniques, and Tools*, Pearson Education, 2006.

[Bi] Charles Donnelly and Richard Stallman, *Bison, The Yacc-compatible Parser Generator*, The Free Software Foundation, 2013. `http://www.gnu.org/software/bison/`

[DEK1] Donald E. Knuth, *The TeXbook*, Addison-Wesley Reading, Massachusetts, 1984.

[DEK2] Donald E. Knuth *The future of TeX and METAFONT*, *TUGboat* **11** (4), p. 489, 1990. `http://tug.org/TUGboat/tb11-4/tb30futuretex.pdf`

[Do] Jean-luc Doumont, *Pascal pretty-printing: An example of "preprocessing with TeX"*, *TUGboat* **15** (3), 1994 — Proceedings of the 1994 TUG Annual Meeting. `http://tug.org/TUGboat/tb15-3/tb44doumont.pdf`

[Er] Sebastian Thore Erdweg and Klaus Ostermann, *Featherweight TeX and Parser Correctness*, Proceedings of the Third International Conference on Software Language Engineering, pp. 397–416, Springer-Verlag Berlin, Heidelberg, 2011.

[Fi] Jonathan Fine, *The* `\CASE` *and* `\FIND` *macros*, TUGboat **14** (1), pp. 35–39, 1993. `http://tug.org/TUGboat/tb14-1/tb38fine.pdf`

[Go] Pedro Palao Gostanza, *Fast scanners and self-parsing in TeX*, *TUGboat* **21** (3), 2000 — Proceedings of the 2000 Annual Meeting. `http://tug.org/TUGboat/tb21-3/tb68gost.pdf`

[Gr] Andrew Marc Greene, *BASIX — An interpreter written in TeX*, *TUGboat* **11** (3), 1990 — Proceedings of the 1990 TUG Annual Meeting. `http://tug.org/TUGboat/tb11-3/tb29greene.pdf`

[Ha] Hans Hagen, *LuaTeX: Halfway to version 1*, *TUGboat* **30** (2), pp. 183–186, 2009. `http://tug.org/TUGboat/tb30-2/tb95hagen-luatex.pdf`

[Ie] R. Ierusalimschy et al., *Lua 5.1 Reference Manual*, `Lua.org`, August 2006. `http://www.lua.org/manual/5.1/`

[La] *The* `l3regex` *package: Regular expressions in TeX*, The LaTeX3 Project. `http://www.ctan.org/pkg/l3regex`

[Pa] Vern Paxson et al., *Lexical Analysis With Flex, for Flex 2.5.37*, July 2012. `http://flex.sourceforge.net/manual/`

[Wo] Marcin Woliński, `Pretprin` — *A LaTeX2ε package for pretty-printing texts in formal languages*, *TUGboat* **19** (3), 1998 — Proceedings of the 1998 TUG Annual Meeting. `http://tug.org/TUGboat/tb19-3/tb60wolin.pdf`

◇ Alexander Shibakov
Dept. of Mathematics
Tennessee Tech. University
Cookeville, TN
`http://math.tntech.edu/alex`

## Entry-level MetaPost 4: Artful lines

Mari Voipio

For basic information on running MetaPost, either standalone or within a ConTEXt document, see `http://tug.org/metapost/runningmp.html`. For previous installments of this tutorial series, see `http://tug.org/TUGboat/intromp`.

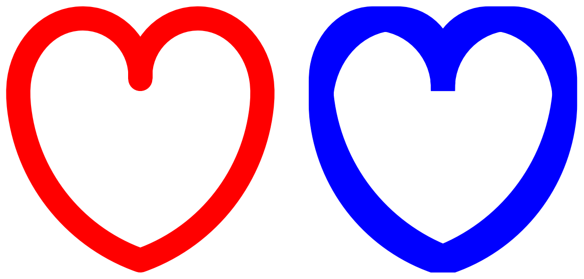
Thus far, we've done very little with lines except change their color. Other than that, we have used the built-in default settings for things like line width and line joins. In this tutorial we learn to tweak lines and use some special effects to put lines to work.

### 1 Line width

Lines are drawn with a pen. The default pen is `pencircle` with a line similar to what e.g. a ballpoint pen produces. When we want to change line width, we always need to specify both the pen "nib" and the width desired:

```
numeric u; u := 4mm; % measurement unit

path heart;
heart := (4u,0u) .. (0u,5u) .. (0u,6u)
  .. (2u,8u) .. (4u,6u) -- (4u,6u) .. (6u,8u)
  .. (8u,6u) .. (8u,5u) .. (4u,0u) -- cycle;

draw heart withpen pencircle scaled .8u
  withcolor red;
draw heart shifted (10u,0)
  withpen pensquare scaled .8u withcolor blue;
```

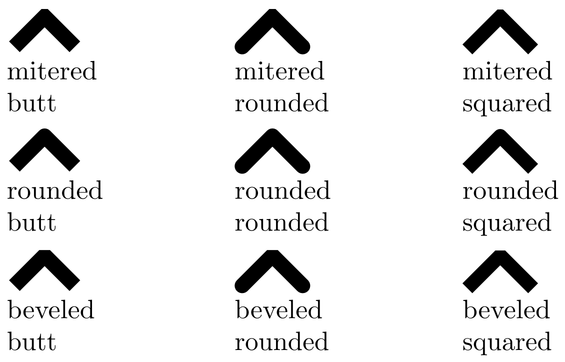
(Grayscaled for printed *TUGboat* output.)



For more information about pens, e.g. calligraphic pens, see the MetaFun manual [3, p. 36–38].

### 2 Line joins and line caps

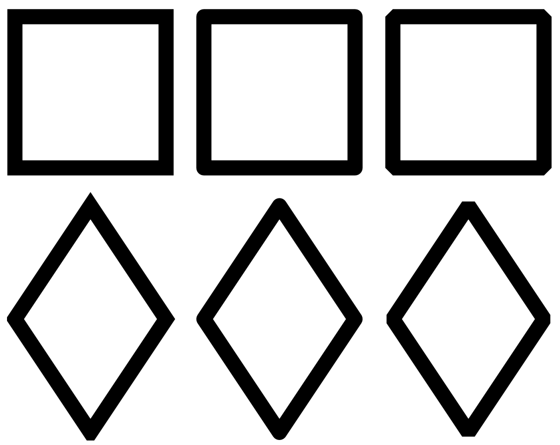
Where lines are joined together, the corner can be turned in several different ways: "sharp" (`mitered`), rounded (`rounded`) or "cut off" (`beveled`). When a line is not cycled into a closed object, it also has two ends that can be "capped" in different ways: `butt` means straight end without any line cap, `rounded` adds a rounded line cap and `squared` adds a square cap to the end. If you look carefully at the example below, you can see that the butted line is shorter than the one with squared caps.



Linejoins and linecaps

How a linejoin looks depends also on the angle of the corner:

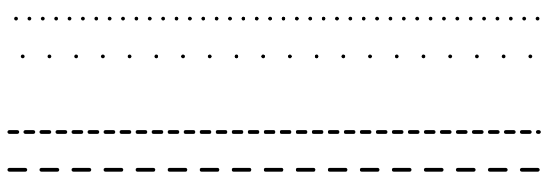
```
pickup pencircle scaled .2cm; % pen width

path rectangle; rectangle := (0,0) -- (2cm,0)
  -- (2cm,2cm) -- (0,2cm) -- cycle;
path diamond;   diamond := (1cm,0) -- (2cm,1.5cm)
  -- (1cm,3cm) -- (0,1.5cm) -- cycle;

linejoin := mitered;
draw rectangle shifted (0,3.5cm);
draw diamond;

linejoin := rounded;
draw rectangle shifted (2.5cm,3.5cm);
draw diamond shifted (2.5cm,0);

linejoin := beveled;
draw rectangle shifted (5cm,3.5cm);
draw diamond shifted (5cm,0);
```



### 3 Dashed lines

In addition to a solid line, we can create dotted and dashed lines. The distance between the dots/dashes is adjusted with the setting `scaled`; the bigger the number, the more space there is between the dots or dashes.

```
pickup pencircle scaled .5mm; % pen width

% dotted lines
draw (0,20mm) -- (70mm,20mm) dashed withdots;
draw (0,15mm) -- (70mm,15mm) dashed withdots
  scaled 2;

% dashed lines
draw (0,5mm) -- (70mm,5mm) dashed evenly;
draw (0,0) -- (70mm,0) dashed evenly scaled 2;
```



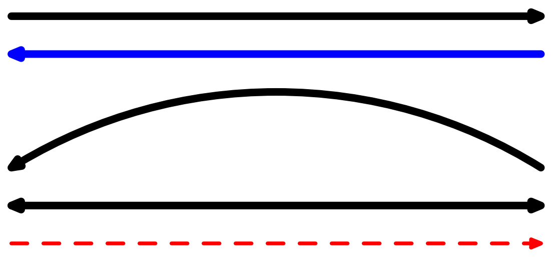
It is not possible to fill paths that have a dashed (out)line. It is also advisable to use only `pencircle` in combination with dashed lines.

For more information on adjusting dashed lines, see the MetaPost manual [2, pp. 37–40] and the MetaFun manual [3, pp. 40–41].

## 4 Arrows

There are separate commands for drawing arrows, but they have the same settings as the plain `draw` command: pen, dashing and color. Arrows go from left to right by default; an arrow with the arrowhead on the left can be created either by giving the coordinates right-to-left or by using the `drawarrow reverse()` command.

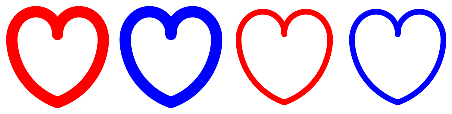
```
pickup pencircle scaled .1cm;

drawarrow (0,3.5cm) -- (7cm,3.5cm);
drawarrow (7cm,3cm) -- (0,3cm) withcolor blue;
drawarrow reverse((0,1.5cm) .. (3.5cm,2.5cm)
 .. (7cm,1.5cm));
drawdblarrow (0,1cm) -- (7cm,1cm);

drawarrow (0,0.5cm) -- (7cm,0.5cm)
  withpen pencircle scaled .05cm
  dashed evenly scaled 2
  withcolor red;
```



## 5 Applying settings for multiple paths

In the examples above we have already used `pickup` to set the pen width for multiple paths. I think of this as having a bunch of pens on the table and picking up one after another to draw with; the drawn lines have the same width until I switch to a different pen — except that we can override the pickup settings for an individual path by using the `withpen` command, as in the arrow example above. The following example picks up two pens in turn:

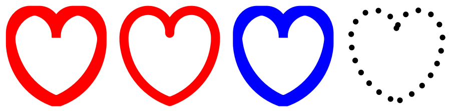
```
numeric u; u := 1.5mm;
heart := (4u,0u) .. (0u,5u) .. (0u,6u)
  .. (2u,8u) .. (4u,6u) -- (4u,6u) .. (6u,8u)
  .. (8u,6u) .. (8u,5u) .. (4u,0u) -- cycle;

pickup pencircle scaled u;
draw heart withcolor red;
draw heart shifted (10u,0) withcolor blue;

pickup pencircle scaled .5u;
draw heart shifted (20u,0) withcolor red;
draw heart shifted (30u,0) withcolor blue;
```



A more versatile command is `drawoptions(`*pen, color, withcolor*`)` which allows us to set default line properties: pen `withpen`), color (`withcolor`) and dashing (`dashed`). This command can be used to set defaults for the whole drawing or just part of it, and is valid until the next `drawoptions()` command. You can also reset everything by giving the `drawoptions` command with nothing within the parentheses.

The `drawoptions` are overridden by setting pen and/or color and/or dashing individually for a path, as usual:

```
numeric u; u := 1.5mm;
heart := (4u,0u) .. (0u,5u) .. (0u,6u)
  .. (2u,8u) .. (4u,6u) -- (4u,6u) .. (6u,8u)
  .. (8u,6u) .. (8u,5u) .. (4u,0u) -- cycle;

drawoptions(withpen pensquare scaled .8u
            withcolor red);

draw heart; % default settings from drawoptions
draw heart shifted (10u,0)
  withpen pencircle scaled .8u; % pen override
draw heart shifted (20u,0)
  withcolor blue; % color override
draw heart shifted (30u,0)
  dashed withdots
  withpen pencircle scaled .5u
  withcolor black; % dash, pen, color override
```



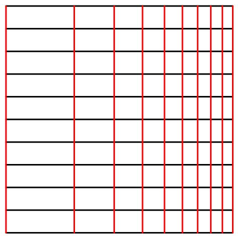
## 6 MetaFun bonus: Grids

With the MetaFun package we can easily create evenly spaced and logarithmic grids. The syntax is:

```
% horizontal/vertical linear:
hlingrid (Min, Max, Step, Length, Width)
vlingrid (Min, Max, Step, Length, Height)

% horizontal/vertical logarithmic:
hloggrid (Min, Max, Step, Length, Width)
vloggrid (Min, Max, Step, Length, Height)
```

The grid settings are used in combination with the `draw` command:

```
pickup pencircle scaled .2mm;

draw hlingrid (0, 10, 1, 3cm, 3cm);
draw vloggrid (0, 10, 1, 3cm, 3cm)
  withcolor red;
```

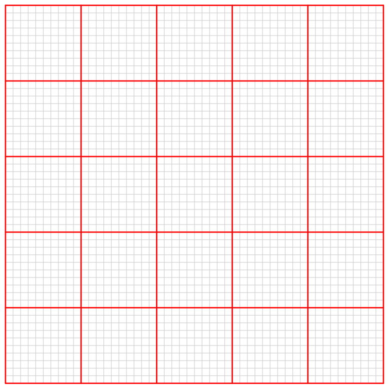


We can create gridded paper with the right grid settings:

```
width  := 5cm; height := 5cm; unit := cm;

drawoptions(withpen pencircle scaled .2pt
            withcolor .8white);
draw vlingrid(0, width /unit, 1/10, width,  height);
draw hlingrid(0, height/unit, 1/10, height, width );

drawoptions(withpen pencircle scaled .5pt
            withcolor red);
draw vlingrid(0, width /unit, 1,    width,  height);
draw hlingrid(0, height/unit, 1,    height, width );
```

We haven't seen the `unit` assignment (in the first line) before: in MetaFun, it applies to numbers without any other unit.

See the MetaFun manual [3, pp. 213–215] for further examples.



## 7 References

[1] Running MetaPost and Metafun.
`http://tug.org/metapost/runningmp.html`

[2] MetaPost manual.
`http://tug.org/docs/metapost/mpman.pdf`

[3] MetaFun manual.
`http://www.pragma-ade.com/general/manuals/metafun-p.pdf`

◇ Mari Voipio
`mari dot voipio (at) lucet dot fi`
`http://www.lucet.fi`

## drawdot in MetaPost: A bug, a fix

Hans Hagen

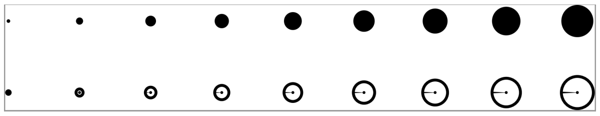
It is no secret that Don Knuth uses MetaPost for graphics in his books nowadays. This has the nice side effect of large-scale testing of MetaPost stability. Recently he uncovered a bug in the `drawdot` macro, which plain MetaPost has always defined like this:

```
def drawdot expr z =     % original definition
  addto currentpicture
    contour makepath currentpen shifted z _op_
enddef;
```

The submitted test was this:

```
for j = 1 upto 9 :
  pickup pencircle scaled .4;
  drawdot (10j,0) withpen pencircle scaled .5j;
  pickup pencircle scaled .5j;
  drawdot (10j,10);
endfor;
```

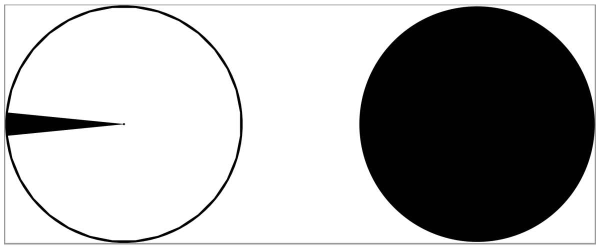
which visualizes as:



Let's simplify and exaggerate this a bit:

```
drawdot origin withpen pencircle scaled 2cm;
pickup pencircle scaled 2cm;
drawdot origin shifted (3cm,0);
```

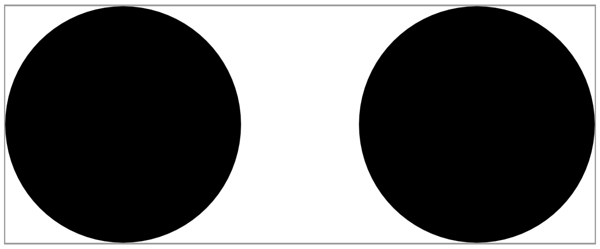
The left-hand variant demonstrates that the old definition of the macro uses the current pen (which by default is one base unit, $\frac{1}{65536}$ of a pixel) to calculate a contour (a.k.a. outline) that then is drawn with a larger pen. The opened up dot is a side effect of the exported PostScript code. The right-hand version shows that picking up the larger pen first and then drawing has a different (and correct) effect.



The two formulations should be equivalent. So the version of MetaPost that will ship with TeX Live 2014 has a new definition of `drawdot`. While the original definitions followed a METAFONT approach, the new definition relies on PostScript doing the work:

```
def drawdot expr p =      % new definition
  if pair p :
    addto currentpicture
      doublepath p withpen currentpen _op_
  else :
    errmessage("drawdot needs a pair ...")
  fi
enddef;


drawdot origin withpen pencircle scaled 2cm;
pickup pencircle scaled 2cm;
drawdot origin shifted (3cm,0);
```

Now our simplified example comes out the same:



This definition is more or less the same as:

```
let drawdot = draw;
```

But our more extensive variant has the advantage that it behaves a bit like a primitive operation: a dot is supposed to be a `pair` and if not, we get an error.

We believe that most users will not notice this change. First of all we have never received a complaint before, which might be an indication that users already used `draw` instead of `drawdot`. Second, dots are normally drawn small, so users might never have noticed such artifacts.

Of course the MetaPost team is curious about what bug Don will come up with next, especially when he needs very large graphics that rely on the new `double` (floating-point) mode of MetaPost.

In the original message, available at `tug.org/pipermail/tex-k/2014-January.txt`, a few more observations were made and testing revealed that there is room for improvement for paths that consist of a point cycling to itself. We will look into these some time in the future.

◇ Hans Hagen
Pragma ADE
http://metapost.org

Editor's note: The figures here are bitmaps, extracted from screenshots, because the conversion of EPS to PDF for production also filled in the open dots! Clearly the effects are dependent on the particular software involved.

# HTML to LaTeX transformation

Frederik R. N. Schlupkothen

## Abstract

(LA)TeX was created as an authoring language that enables authors to keep typesetting quality standards while preparing their printed matter, assuming that the output context is known from the very beginning of the writing process. As a counter-concept, XML is focused on keeping output flexible, providing mechanisms to manage and control the logical structure of documents. Combining the strengths of both ecosystems has been discussed frequently in the past. This article aims to contribute to this discussion by introducing a mapping from HTML to LaTeX, the two most widespread document description languages in their respective fields of application.

## 1 Introduction

The Extensible Markup Language (XML) has become the native markup language for structured information in (distributed) document processing architectures. It provides a core syntax that has been adopted by many document description languages. A broad software ecosystem and further standards have emerged to realize and facilitate the processing of XML-based documents. Among them, the Extensible Stylesheet Language Transformations (XSLT) described in [8] offers a standardized mechanism to translate different XML-based languages into one another. This has made XML the language of choice for cross-media publishing workflows.

However, while the XML processing is fully covered by viable tools, the final document production of printed matter from XML may still be a problem. A potential solution is to integrate the TeX typesetting engine (described in [10]) in XML-based processing chains. For this task, TeXML, an XML representation of TeX commands, was introduced in [12] and its "production proof" implementation discussed in [14]. As any other XML language, TeXML documents can be produced via XSLT. So the last gap to fill in TeXML-based workflows is to define XSL stylesheets that realize the transformation between specific XML and TeX document description languages (see [14]).

Here we introduce a mapping between two document description languages that are well known in their respective fields of application: The HyperText Markup Language (HTML), as the transformation's source language, is the core language of the World Wide Web and has a history that is closely tied to XML. It offers layout-oriented markup semantics primarily for textual content and is used amongst others to describe web pages (see [7]), electronic books (see [4]), and printed matter (see e.g. [9, 13]). LaTeX, as the transformation's output, is the abstraction of TeX's typesetting commands to logical markup.

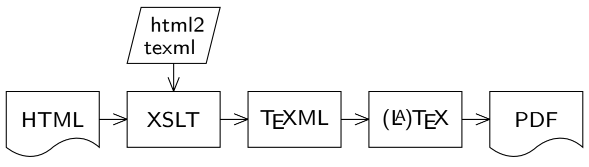


**Figure 1**: HTML to PDF processing

Figure 1 shows the XML-based production workflow that produces the Portable Document Format (PDF) from HTML by the use of the TeX typesetting engine. The particular processing steps are described via an example of emphasized text:

1. The HTML source document describes the emphasized text "dolor" as: `<em>dolor</em>`
2. The XSLT processor queries the stylesheet "html 2 texml" for a matching transformation template. The matching template defines the transformation of the HTML's "em" element to the corresponding LaTeX command in its TeXML representation:
```
<xsl:template match="html:em">
  <tex:cmd name="emph">
    <tex:parm>
      <xsl:apply-templates />
    </tex:parm>
  </tex:cmd>
</xsl:template>
```
3. The result of the XSLT transformation is the following TeXML element:
```
<cmd name="emph">
  <parm>dolor</parm>
</cmd>
```
4. The TeXML processor converts the TeXML element to a LaTeX command: `\emph{dolor}`
5. The LaTeX processor typesets: *dolor*

The prior example shows that defining the transformation between two languages needs insight into the differences in conceptual syntax and semantic coverage of source and destination language. While HTML is native to the XML syntax, TeXML is reproducing the syntax logic inherited from (LA)TeX. While LaTeX is native to printed matter, HTML was initially designed for electronic resources. Section 2 introduces the underlying concepts of the HTML and LaTeX markup syntax. Section 3 introduces the mapping between HTML and LaTeX markup semantics. Finally, section 4 shows a complete document with its corresponding representations in HTML and LaTeX.

## 2 Markup syntax

The following two subsections give an overview of the very basic HTML and LaTeX syntax. They do not introduce the full syntax but focus on the aspects needed within this article. Full descriptions can be found in [3] for XML, the underlying markup language of HTML as described in [6], and in [10, 11] for LaTeX.

### 2.1 Basic syntax of HTML

An HTML document consists of *elements* that are either *empty* or *non-empty*. The boundaries of a non-empty element is marked by a *start-tag* and a *end-tag*. Tag delimiters are the `<` and `>` characters. The element *type* is defined in the start-tag by its *name*. The end-tag repeats the element name preceded by a `/` character. The element's *content* is enclosed between the start- and end-tag. The content consists of *character data* (i.e. text), subordinated *child* elements, or both. An element without content is called empty and is either described by a start-tag that is directly followed by its end-tag or by an *empty-element-tag*. An empty-element-tag has the same form as a start-tag but ends with a `/` character. The following example shows a non-empty 'p'-element with mixed content:

$$\underbrace{\texttt{<p>}}_{\text{start-tag}}\underbrace{\texttt{Lorem ipsum }\overbrace{\texttt{<em>dolor</em>}}^{\text{child element}}\texttt{ sit amet.}}_{\text{content}}\underbrace{\texttt{</p>}}_{\text{end-tag}}$$

An element can possess *attributes*. Attributes are noted in the start-tag or empty-element-tag behind the element name. Attributes consist of a *name-value*-pair. The attribute-value is given between two `'` or `"` characters and is assigned to its name by a preceding `=` character. The following example shows an empty 'img' element with a 'src' attribute of the value 'uri':

$$\texttt{<img }\overbrace{\underbrace{\texttt{src}}_{\text{name}}\texttt{=}\underbrace{\texttt{"uri"}}_{\text{value}}}^{\text{attribute}}\texttt{ />}$$

There is exactly one *root* element that includes all the document's content. The tag placement within the document follows the rules of mathematical brackets. The examples below show possible tag placements by means of 'a' and 'b' elements:

| | |
|---|---|
| sequence | `<a></a><b></b>` |
| subordination | `<a><b></b></a>` |
| *syntax error* | `<a><b></a></b>` |

### 2.2 Basic syntax of LaTeX

A LaTeX document consists of *commands* that describe either output characters (i.e. characters to typeset), special characters (e.g. the `~` character for a non-breaking space), or *control sequences*. There are two types of control sequences: *control words* and *control symbols*. A control word starts with a `\` character followed by its *name* that consists of one or more letters (i.e. lower- or uppercase letters 'a' to 'z') and is terminated by either a space or another non-letter. A control symbol starts with a `\` character followed by one non-letter. A command can possess optional and required *parameters* that are set by *arguments*. Optional parameter arguments are noted after the command name between square brackets, and required parameter arguments between curly braces. The following example shows a 'usepackage'-command with an optional parameter set to 'utf8' and a required parameter set to 'inputenc':

$$\texttt{\textbackslash usepackage }\overbrace{\underbrace{\texttt{[utf8]}}_{\text{optional}}\underbrace{\texttt{\{inputenc\}}}_{\text{required}}}^{\text{parameters}}$$

Furthermore there are two special types of commands: *environments* and *declarations*. Environments are pairs of 'begin'- and 'end'-commands that enclose the environment's content. The environment name is provided as the first required argument of the corresponding 'begin'- and 'end'-commands. The arguments of the environment are noted as further arguments of the 'begin'-command. Declarations influence the behavior of following commands. The *scope* (i.e. range of effect) of most declarations is limited to its enclosing environment or *group*. The group delimiters are the `{` and `}` characters. The placement of group delimiters and environment commands follows the rules of mathematical brackets. The examples below show possible placements by example of a group and an 'x'-environment:

| | |
|---|---|
| sequence | `{ } \begin{x} \end{x}` |
| subordination | `{ \begin{x} \end{x} }` |
| *syntax error* | *`{ \begin{x} } \end{x}`* |

## 3 Markup correspondence

The following sections introduce a possible mapping between HTML elements and LaTeX commands in the order of the HTML module descriptions in [1]. For an XSLT implementation transforming HTML to TeXML, the following mapping tables show the resulting LaTeX commands for expository purposes.

## 3.1 Core Modules

The HTML Core Modules assemble the markup that is common to all HTML dialects that are derived from module-based HTML. This core markup for high level structures, basic text, hyperlinks, and lists of HTML documents and its corresponding LaTeX commands are described in the following subsections.

### 3.1.1 Structure Module

The HTML Structure Module defines the high level markup of a document. The `html`-element is the document's root containing the meta-information (`head`) and the actual content (`body`) of a document. LaTeX follows a similar separation with its preamble and `document`-environment. Table 1 below shows the corresponding commands.

**Table 1**: HTML to LaTeX structure mapping

| HTML | LaTeX |
|---|---|
| `<html>`⟨...⟩ | `\documentclass{report}`⟨...⟩ |
| `<head>`⟨...⟩ | ⟨...⟩ |
| `<title>`⟨...⟩ | `\title{`⟨...⟩`}` |
| `<body>`⟨...⟩ | `\begin{document}`⟨...⟩ |

### 3.1.2 Text Module

The HTML Text Module defines the basic text markup to describe heading, block, and inline elements. Most of these elements have equivalent commands in LaTeX, but not all. In these cases the '↦' symbol indicates the default formatting in HTML where the Presentation Module described in section 3.2.1 might be used for an alternative, not corresponding semantically, mapping.

**Headings** The HTML Text Module defines six levels of headings (`h1` to `h6`). LaTeX offers a specific heading hierarchy that depends on the given document class. Table 2 below shows the corresponding heading commands for the `report` document class.

**Table 2**: HTML to LaTeX heading mapping

| HTML | LaTeX |
|---|---|
| `<h1>`⟨...⟩ | `\chapter{`⟨...⟩`}` |
| `<h2>`⟨...⟩ | `\section{`⟨...⟩`}` |
| `<h3>`⟨...⟩ | `\subsection{`⟨...⟩`}` |
| `<h4>`⟨...⟩ | `\subsubsection{`⟨...⟩`}` |
| `<h5>`⟨...⟩ | `\paragraph{`⟨...⟩`}` |
| `<h6>`⟨...⟩ | `\subparagraph{`⟨...⟩`}` |

**Blocks** The HTML Text Module defines elements to mark text groups as paragraphs (`p`), contact information (`address`), quotations (`blockquote`), generic groups (`div`), and preformatted text (`pre`). Table 3 shows the corresponding LaTeX commands (using the LaTeX core package alltt for preformatted text).

**Table 3**: HTML to LaTeX block mapping

| HTML | LaTeX |
|---|---|
| `<p>`⟨...⟩ | ⟨...⟩ `\par` |
| `<address>`⟨...⟩ | ↦ *italic* |
| `<blockquote>`⟨...⟩ | `\begin{quote}`⟨...⟩ |
| `<div>`⟨...⟩ | ⟨...⟩ |
| `<pre>`⟨...⟩ | `\begin{alltt}`⟨...⟩ |

**Inlines** The HTML Text Module defines markup for text fragments. This includes abbreviations (`abbr`) and acronyms (`acronym`), citations (`cite`), quotations (`q`), and definitions (`dfn`), program code (`code`), sample output (`samp`), arguments (`var`), and input (`kbd`), regular (`em`) and strong (`strong`) emphases, generic fragments (`span`), and forced line breaks (`br`). Table 4 below shows the corresponding LaTeX commands (using the glossaries package for abbreviations and acronyms, the csquotes package for quotations, and the listings package for code).

**Table 4**: HTML to LaTeX inline mapping

| HTML | LaTeX |
|---|---|
| `<abbr>`⟨...⟩ | `\acrshort{`⟨...⟩`}` |
| `<acronym>`⟨...⟩ | `\ac{`⟨...⟩`}` |
| `<cite>`⟨...⟩ | `\cite{`⟨*gen-id*⟩`}` `\bibitem{`⟨*gen-id*⟩`}`⟨...⟩ |
| `<q>`⟨...⟩ | `\enquote{`⟨...⟩`}` |
| `<dfn>`⟨...⟩ | ↦ *italic* |
| `<code>`⟨...⟩ | `\lstinline`\|⟨...⟩\| |
| `<samp>`⟨...⟩ | ↦ *teletype* |
| `<var>`⟨...⟩ | ↦ *italic* |
| `<kbd>`⟨...⟩ | ↦ *teletype* |
| `<em>`⟨...⟩ | `\emph{`⟨...⟩`}` |
| `<strong>`⟨...⟩ | ↦ *bold* |
| `<span>`⟨...⟩ | ⟨...⟩ |
| `<br />` | `\newline` |

### 3.1.3 Hypertext Module

The HTML Hypertext Module defines markup to describe hyperlinks. They are described by source anchors (`a`) that reference to contents inside or outside of the document via Unified Resource Identifiers (URIs). Referenceable document fragments are marked by common 'id' attributes that can be applied to all elements. The use of traversable hyperlinks is an adequate solution in the context of electronic documents; its mapping to corresponding

LaTeX commands by means of the hyperref package is shown in Table 5. However, in the context of printed matter a solution with references by e.g. visual key or page numbers might be more appropriate.

**Table 5**: HTML to LaTeX hypertext mapping

| HTML | LaTeX |
|---|---|
| `<a href="`⟨*uri*⟩`">`⟨...⟩ | `\href{`⟨*uri*⟩`}{`⟨...⟩`}` |
| `<a href="`⟨*id*⟩`">`⟨...⟩ | `\hyperref[`⟨*id*⟩`]{`⟨...⟩`}` |
| `id="`⟨*id*⟩`"` | `\label{`⟨*id*⟩`}` |

### 3.1.4 List Module

The HTML List Module defines markup to describe ordered (`ol`) and unordered (`ul`) lists as sequences of list items (`li`) and furthermore markup to describe definition lists (`dl`) that are composed of sequences of term (`dt`) and description (`dd`) pairs. Table 6 below shows corresponding LaTeX commands.

**Table 6**: HTML to LaTeX list mapping

| HTML | LaTeX |
|---|---|
| `<ol>`⟨...⟩ | `\begin{enumerate}`⟨...⟩ |
| `<ul>`⟨...⟩ | `\begin{itemize}`⟨...⟩ |
| `<li>`⟨...⟩ | `\item` ⟨...⟩ |
| `<dl>`⟨...⟩ | `\begin{description}`⟨...⟩ |
| `<dt>`⟨...⟩ | `\item[`⟨...⟩`]` |
| `<dd>`⟨...⟩ | ⟨...⟩`\\` |

## 3.2 Text Extension Modules

The HTML Text Extension Modules assemble additional text markup to control text rendering, maintenance, and direction for HTML documents. These and the corresponding LaTeX commands are described in the following subsections.

### 3.2.1 Presentation Module

The HTML Presentation Module defines markup to control the text rendering. It provides elements to render text in/as bold (`b`) and italic (`i`) style, typewriter (`tt`), super- (`sup`) or subscripted (`sub`), larger (`big`) or smaller font (`small`). Additionally the module provides an element to render horizontal rules (`hr`). LaTeX offers corresponding commands with the exception of 'textsubscript' that relies on the subscript package. The relsize package offers commands to realize relative font sizes (as intended by the 'big' and 'small' elements in HTML). Table 7 shows a possible mapping.

### 3.2.2 Edit Module

The HTML Edit Module defines editing-related markup. It provides elements to mark content as deleted (`del`) or inserted (`ins`). The changes package offers semantically corresponding LaTeX commands as shown in Table 8 below. However, if LaTeX is used as final output format only, a more stable solution might be to simply output contents of 'ins'-elements, but not those of 'del'-elements.

**Table 7**: HTML to LaTeX presentation mapping

| HTML | LaTeX |
|---|---|
| `<b>`⟨...⟩ | `\textbf{`⟨...⟩`}` |
| `<i>`⟨...⟩ | `\textit{`⟨...⟩`}` |
| `<tt>`⟨...⟩ | `\texttt{`⟨...⟩`}` |
| `<sup>`⟨...⟩ | `\textsuperscript{`⟨...⟩`}` |
| `<sub>`⟨...⟩ | `\textsubscript{`⟨...⟩`}` |
| `<big>`⟨...⟩ | `{\larger` ⟨...⟩`}` |
| `<small>`⟨...⟩ | `{\smaller` ⟨...⟩`}` |
| `<hr />` | `\hrulefill` |

**Table 8**: HTML to LaTeX edit mapping

| HTML | LaTeX |
|---|---|
| `<del>`⟨...⟩ | `\deleted{`⟨...⟩`}` |
| `<ins>`⟨...⟩ | `\added{`⟨...⟩`}` |

### 3.2.3 Bi-directional Text Module

The HTML Bi-directional Text Module defines markup to declare text direction changes. It provides an attribute to control the direction of text (`dir`) that can be applied to all elements including a special element (`bdo`) to override the current text direction. The bidi package offers corresponding LaTeX commands. Table 9 below shows the corresponding commands for inline text. However, the bidi package defines a set of new environments which replace common LaTeX commands (e.g. lists and footnotes) which makes the general mapping between elements and commands more complex. Furthermore the combination with other common packages (e.g. hyperref or longtable) remains problematic. So a more stable solution might be to omit bi-directional text controls during the transformation process and to apply such changes manually in the LaTeX document.

**Table 9**: HTML to LaTeX bidi mapping

| HTML | LaTeX |
|---|---|
| `<bdo dir="ltr">`⟨...⟩ | `\LR{`⟨...⟩`}` |
| `<bdo dir="rtl">`⟨...⟩ | `\RL{`⟨...⟩`}` |

## 3.3 Forms Modules

The HTML Forms Modules define markup to describe interactive forms that can define, organize, and receive (textual) input and selections. The hyperref package implements most HTML form elements for LaTeX. As with hyperlinks, the use of interactive forms is adequate for electronic documents; their mapping by means of the hyperref package is shown in Table 10 below. However, in the context of printed matter, an alternative solution as given e.g. by the formular package might be more suitable.

**Table 10**: HTML to LaTeX forms mapping

| HTML | LaTeX |
|---|---|
| `<form>`⟨...⟩ | `\begin{Form}`⟨...⟩ |
| `<input />` | `\TextField{`⟨*label*⟩`}` |
| `type="password"` | `\TextField[password]{`⟨*label*⟩`}` |
| `type="checkbox"` | `\CheckBox{`⟨*label*⟩`}` |
| `type="button"` | `\PushButton{`⟨*label*⟩`}` |
| `type="radio"` | `\ChoiceMenu[radio]{`⟨*label*⟩`}{=}` |
| `type="submit"` | `\Submit{`⟨*label*⟩`}` |
| `type="reset"` | `\Reset{`⟨*label*⟩`}` |
| `type="file"` | `\TextField[fileselect]{`⟨*label*⟩`}` |
| `type="hidden"` | `\TextField[hidden]{`⟨*label*⟩`}` |
| `type="image"` | `\Submit[submitcoordinates]{`⟨*img*⟩`}` |
| `<select>`⟨...⟩ | `\ChoiceMenu{`⟨*label*⟩`}{`⟨*options*⟩`}` |
| `<option>`⟨...⟩ | ⟨...⟩ |
| `<textarea>`⟨...⟩ | `\TextField[multiline]{`⟨*label*⟩`}` |
| `<button>`⟨...⟩ | `\Submit{`⟨...⟩`}` |
| `type="button"` | `\PushButton{`⟨...⟩`}` |
| `type="reset"` | `\Reset{`⟨...⟩`}` |
| `<fieldset>`⟨...⟩ | ⟨...⟩ |
| `<label>`⟨...⟩ | ⟨...⟩ |
| `<legend>`⟨...⟩ | ⟨...⟩ |
| `<optgroup>`⟨...⟩ | ⟨...⟩ |

## 3.4 Table Modules

The HTML Table Modules define markup to describe tables (`table`) by organizing their data (`td`) and header (`th`) cells in rows (`tr`). These rows can be grouped into table headers (`thead`), footers (`tfoot`), and bodies (`tbody`). Column-based markup is realized by standoff elements (`col` and `colgroup`). A table caption (`caption`) can provide a short description of the table contents.

LaTeX table definitions differ in two essential aspects from HTML: *(i)* The total number of table columns has to be given explicitly to a LaTeX table environment. This is not necessary in HTML but calculated continuously by the rendering engine at processing time. *(ii)* LaTeX table cells that span several rows (by means of the multirow package) cover the adjacent cells in the following rows; therefore empty cells need to be inserted in the following rows. This is not necessary in HTML but the rendering engine automatically shifts the cells of the following rows according to the reading direction.

```
$table = \{row_i \mid row_i = \{cell_{ij}\}\}$
$grid = \{slot_i \mid slot_i = \langle x, y \rangle\}$
procedure TABLE($table$)
    $grid \leftarrow \{\emptyset\}$

    for all $row_i \mid i = 1 \,..\, n$ do
        $y \leftarrow i$
        $x \leftarrow 1$

        for all $cell_{ij} \mid j = 1 \,..\, m$ do
            while $grid \ni \langle x, y \rangle$ do
                EMPTY CELL($x, y$)
                $x \leftarrow x + 1$
            end while

            for $y_{\text{cell}} \leftarrow 0 \,..\, \text{rowspan}(cell_{ij}) - 1$ do
                for $x_{\text{cell}} \leftarrow 0 \,..\, \text{colspan}(cell_{ij}) - 1$ do
                    $grid \leftarrow grid \cup \langle x + x_{\text{cell}}, y + y_{\text{cell}} \rangle$
                end for
            end for

            $x \leftarrow x + \text{colspan}(cell_{ij})$
        end for
    end for
end procedure
```

**Figure 2**: HTML table cell positioning algorithm

Hence for the transformation of HTML tables to LaTeX this information (total number of table columns and position of additional empty cells) need to be precalculated. Therefore the transformation process has to include parts of the HTML table processing model described in [2]. This model describes an HTML table as a set of cells that are positioned on a two-dimensional grid of slots. The algorithm shown in Figure 2 calculates the cell positioning and illustrates how the additional empty cells are inserted; hence the total number of table columns is given by the maximum $x$-coordinate within the final grid. Table 11 shows the mapping of HTML table elements to corresponding LaTeX commands by means of the longtable package.

## 3.5 Image Module

The HTML Image Module defines markup to embed external images. The graphicx package offers a corresponding LaTeX command as shown in Table 12.

## 3.6 Further Modules

The HTML specification describes further modules that define markup to realize dynamic and interactive document content, mechanisms to control layout, and deprecated markup for backwards compatibility with legacy HTML. Due to the focus of this article on the transfer of the logical structure of HTML documents

**Table 11**: HTML to LaTeX table mapping

| HTML | LaTeX |
|---|---|
| `<caption>`⟨...⟩ | `\caption{`⟨...⟩`}` |
| `<table>`⟨...⟩ | `\begin{longtable}{`⟨*col*⟩`}`⟨...⟩ |
| `<td>`⟨...⟩ | ⟨...⟩`&` |
| `<th>`⟨...⟩ | `\bf{}`⟨...⟩`&` |
| `colspan="`⟨*span*⟩`"` | `\multicolumn{`⟨*span*⟩`}{l}{`⟨...⟩`}` |
| `rowspan="`⟨*span*⟩`"` | `\multirow{`⟨*span*⟩`}{*}{`⟨...⟩`}` |
| `<tr>`⟨...⟩ | ⟨...⟩`\\` |
| `<col />` | |
| `<colgroup>`⟨...⟩ | |
| `<tbody>`⟨...⟩ | ⟨...⟩ |
| `<thead>`⟨...⟩ | ⟨...⟩`\endhead` |
| `<tfoot>`⟨...⟩ | ⟨...⟩`\endfoot` |

**Table 12**: HTML to LaTeX image mapping

| HTML | LaTeX |
|---|---|
| `<img src="`⟨*uri*⟩`"/>` | `\includegraphics{`⟨*uri*⟩`}` |

to LaTeX, the mapping of these specialized modules is not described in detail. However, in specific use cases the support of these modules might be desired. The following hints might serve as a starting point to implement a transformation of these modules' features to LaTeX.

The HTML Applet, Object, Scripting, and Intrinsic Events modules define markup that introduces scripting facilities to manipulate dynamically the document content. At present this is notably realized through the JavaScript programming language, which is partially integrated with LaTeX by means of the insdljs package.

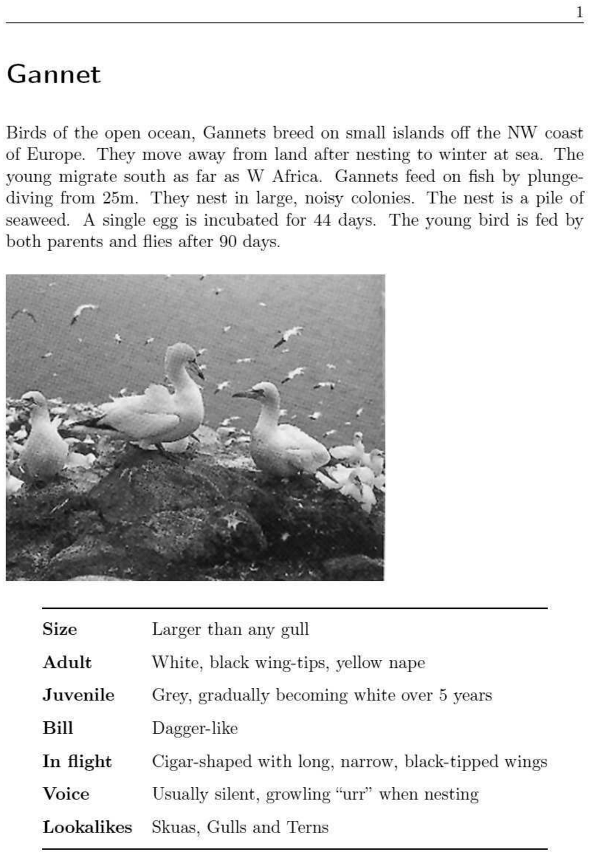
The HTML Client- and Server-side Image Map modules define markup for interactive and hyperlinked images. This functionality can be potentially realized in LaTeX by means of the Ti*k*Z package.

The HTML Frames and Iframe modules define markup to insert one document into another. This can be realized in LaTeX with the `\input` and/or `\include` commands.

The HTML Style Sheet and Style Attribute modules define markup to integrate layout definitions realized through Cascading Style Sheets (CSS). CSS has its own syntax and description logic — its transformation to LaTeX is a topic all its own, which has been outlined e.g. in [15].

## 4 An example

Figure 3 shows an example page taken from a bird guide. On the next page, Figure 4 shows a possible coding of this page using HTML, and Figure 5 its corresponding representation in LaTeX.


1

Gannet

Birds of the open ocean, Gannets breed on small islands off the NW coast of Europe. They move away from land after nesting to winter at sea. The young migrate south as far as W Africa. Gannets feed on fish by plunge-diving from 25m. They nest in large, noisy colonies. The nest is a pile of seaweed. A single egg is incubated for 44 days. The young bird is fed by both parents and flies after 90 days.

| | |
|---|---|
| **Size** | Larger than any gull |
| **Adult** | White, black wing-tips, yellow nape |
| **Juvenile** | Grey, gradually becoming white over 5 years |
| **Bill** | Dagger-like |
| **In flight** | Cigar-shaped with long, narrow, black-tipped wings |
| **Voice** | Usually silent, growling "urr" when nesting |
| **Lookalikes** | Skuas, Gulls and Terns |

**Figure 3**: An example document, derived from [5]

## 5 Conclusion

While HTML is increasingly becoming the common document description language for different output media (web, print, e-books, ...), the problem of creating well-typeset documents from HTML is not yet fully solved within the XML ecosystem. The article at hand has introduced a mapping from HTML elements to corresponding LaTeX commands, in order to use the TeX typesetting engine for this task.

With the multitude of existing LaTeX extensions released as packages, almost any HTML description can be ported to LaTeX and typeset according to its original logic. Unfortunately, the use of LaTeX packages often comes with a catch: while many HTML structures can be used recursively (e.g. nested lists or tables), LaTeX packages tend to override existing commands giving them a new meaning (e.g. the newline command is redefined in table environments to end a row). These context-dependent syntax-changes can make a mapping potentially error-prone for deep document structures.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>Gannet</title></head>
  <body>
    <h1>Gannet</h1>
    <p>Birds of the open ocean, Gannets breed on small islands off the <abbr title="northwest"
      >NW</abbr> coast of Europe. They move away from land after nesting to winter at sea. The
      young migrate south as far as <abbr title="west">W</abbr> Africa. Gannets feed on fish by
      plunge-diving from 25<abbr title="meters">m</abbr>. They nest in large, noisy colonies. The
      nest is a pile of seaweed. A single egg is incubated for 44 days. The young bird is fed by
      both parents and flies after 90 days.</p>
    <div><img src="gannet.jpg" alt="Gannet"/></div>
    <table>
      <tr><th>Size</th><td>Larger than any gull</td></tr>
      <tr><th>Adult</th><td>White, black wing-tips, yellow nape</td></tr>
      <tr><th>Juvenile</th><td>Grey, gradually becoming white over 5 years</td></tr>
      <tr><th>Bill</th><td>Dagger-like</td></tr>
      <tr><th>In flight</th><td>Cigar-shaped with long, narrow, black-tipped wings</td></tr>
      <tr><th>Voice</th><td>Usually silent, growling <q>urr</q> when nesting</td></tr>
      <tr><th>Lookalikes</th><td>Skuas, Gulls and Terns</td></tr>
    </table>
  </body>
</html>
```

**Figure 4**: HTML source, describing the document shown in Figure 3

```
\documentclass{report}
% preamble ...

\begin{document}
\chapter{Gannet}

Birds of the open ocean, Gannets breed on small islands off the \acrshort{NW}coast of Europe. They
move away from land after nesting to winter at sea. The young migrate south as far as \acrshort{W}
Africa. Gannets feed on fish by plunge-diving from 25\acrshort{m}. They nest in large, noisy
colonies. The nest is a pile of seaweed. A single egg is incubated for 44 days. The young bird is
fed by both parents and flies after 90 days.\par

\includegraphics{gannet.jpg}

\begin{longtable}{ll}
\toprule
\bf{}Size       & Larger than any gull \\
\bf{}Adult      & White, black wing-tips, yellow nape \\
\bf{}Juvenile   & Grey, gradually becoming white over 5 years \\
\bf{}Bill       & Dagger-like \\
\bf{}In flight  & Cigar-shaped with long, narrow, black-tipped wings \\
\bf{}Voice      & Usually silent, growling \enquote{urr} when nesting \\
\bf{}Lookalikes & Skuas, Gulls and Terns \\
\bottomrule
\end{longtable}

\end{document}
```

**Figure 5**: LaTeX output, exported from HTML shown in Figure 4

The mappings introduced in this article have been developed in the context of an XSLT implementation within a TeXML-based workflow, but do not rely on it and can be implemented through other approaches as well. However, the principle of TeXML, to provide a processor that transforms specific TeX commands from a generic XML representation to the TeX format, realizes a separation between the task of format transformation and the task of defining appropriate mappings. This facilitates the definition and adaption of markup correspondences as has e.g. been done by extending the HTML mapping with a third party stylesheet that defines the transformation from MathML to LaTeX.

**Acknowledgment**

The author would like to thank Prof. Dr. Karl-Heinrich Schmidt for giving valuable advice and Gilles Bülow for integrating this document processing workflow into our daily tasks for testing purposes.

**References**

[1] Daniel Austin, Shane McCarron, Subramanian Peruvemba, Masayasu Ishikawa, and Mark Birbeck. XHTML modularization 1.1 — second edition. W3C recommendation, W3C, July 2010. `http://www.w3.org/TR/2010/REC-xhtml-modularization-20100729/`.

[2] Robin Berjon, Steve Faulkner, Travis Leithead, Erika Doyle Navara, Edward O'Connor, Silvia Pfeiffer, and Ian Hickson. HTML 5. W3C candidate recommendation, W3C, August 2013. `http://www.w3.org/TR/2013/CR-html5-20130806/`.

[3] Tim Bray, François Yergeau, C. M. Sperberg-McQueen, Jean Paoli, and Eve Maler. Extensible markup language (XML) 1.0 (fourth edition). W3C recommendation, W3C, August 2006. `http://www.w3.org/TR/2006/REC-xml-20060816/`.

[4] Markus Gylling, William McCoy, Elika J. Etemad, and Matt Garrish. EPUB content documents 3.0. IDPF recommended specification, IDPF, October 2011. `http://www.idpf.org/epub/30/spec/epub30-contentdocs-20111011.html`.

[5] Renate Henschel, John Bateman, and Judy Delin. Automatic genre-driven layout generation. In *Proceedings of the 6th "Konferenz zur Verarbeitung natürlicher Sprache" (KONVENS) Conference*, Saarbrücken, September 2002.

[6] Masayasu Ishikawa and Shane McCarron. XHTML 1.1 — module-based XHTML — second edition. W3C recommendation, W3C, November 2010. `http://www.w3.org/TR/2010/REC-xhtml11-20101123/`.

[7] Ian Jacobs, David Raggett, and Arnaud Le Hors. HTML 4.01 specification. W3C recommendation, W3C, December 1999. `http://www.w3.org/TR/1999/REC-html401-19991224/`.

[8] Michael Kay. XSL transformations (XSLT) version 2.0. W3C recommendation, W3C, January 2007. `http://www.w3.org/TR/2007/REC-xslt20-20070123/`.

[9] Sanders Kleinfeld. The case for authoring and producing books in (X)HTML5. In *Proceedings of Balisage: The Markup Conference 2013*, volume 10 of *Balisage Series on Markup Technologies*, Montréal, August 2013.

[10] Donald Ervin Knuth. *The TeXbook*, volume A of *Computers & Typesetting*. Addison-Wesley, March 1986.

[11] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, 2nd edition, November 1994.

[12] Douglas Lovell. TeXML: Typesetting XML with TeX. *TUGboat*, 20(3):176–183, September 1999. `http://tug.org/TUGboat/tb20-3/tb64love.pdf`.

[13] Shane McCarron. XHTML-print — second edition. W3C recommendation, W3C, November 2010. `http://www.w3.org/TR/2010/REC-xhtml-print-20101123/`.

[14] Oleg Parashchenko. TeXML: Resurrecting TeX in the XML world. *TUGboat*, 28(1):5–10, March 2007. `http://tug.org/TUGboat/tb28-1/tb88parashchenko.pdf`.

[15] S. Sankar, S. Mahalakshmi, and L. Ganesh. An XML model of CSS3 as an XƎLaTeX-TeXML-HTML5 stylesheet language. *TUGboat*, 32(3):281–284, December 2011. `http://tug.org/TUGboat/tb32-3/tb102sankar.pdf`.

◇ Frederik R. N. Schlupkothen
University of Wuppertal
Rainer-Gruenter-Str. 21
D-42119 Wuppertal
Germany
`schlupko (at) uni-wuppertal dot de`

# Scientific documents written by novice researchers: A personal experience in Latin America

Ludger O. Suarez–Burgoa

## Abstract

This article presents 20 years of the author's personal experience — described as a particular Latin American experience — about the elaboration of scientific documents created by novice researchers, from the use of the typewriter to prepare a school scientific report up to the conception of a TeX-family class file (i.e. the `unbDscThesisEng` class of the *Universidade de Brasília* Doctoral Thesis, English Version) to prepare his theses. He also gives opinions about a possible Universal Editable Format for scientific documents made by novice researchers, to allow such documents to persist over time without losing information due to changing encoding formats of proprietary software.

## 1 Introduction

Processing scientific documentation is an essential part of publishing results. It is a concern in scientific institutions and academia, because it requires text, tables, equations, figures, and references in a relational structured and compact manner; and all these in conjunction reflect the quality of the message desired to be presented.

Nowadays, researchers individually or in small groups (called in this article *novice researchers*, NRs) have also become actors in this important process, because they can disseminate their results in various effective media now available: indexed journals, both printed and electronic, conference articles, Internet links, and blogs, for example.

With the general availability of free and open source software (FOS) and with increasingly abundant information on the Internet with details and recommendations for proper use of these FOS tools, NRs can now present fully-developed, high-quality, and well-formatted scientific documents. They can be part of the development of scientific documentation by creating support for particular institutions (e.g. TeX-family class files for university theses).

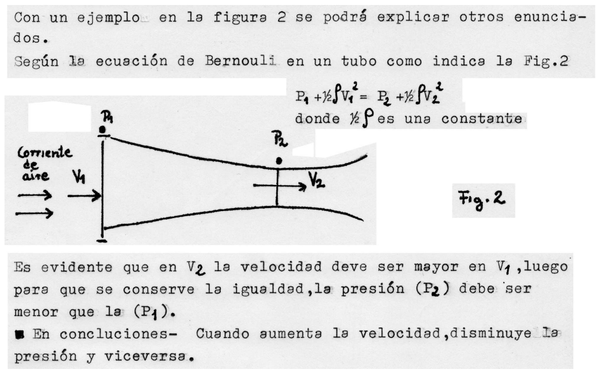
In the following sections the particular experience of an NR of Latin America will be discussed, who has prepared scientific documents and technical reports for around 20 years in Bolivia, Colombia, Brazil, Argentina, Perú, and Chile; and now feels comfortable using: TeX for text, tables, and equation creation, editing and management; SVG to create and edit graphics and plots; and BibTeX to store and manage references of scientific documents.

## 2 Some particular past experiences

Most of the Latin American NRs who nowadays are writing doctoral theses have suffered in the transition from the typewriter to computer software, when dealing with scientific and technical documents.

For example, for a technical note at a school science fair, it was common to see in the 1980s the text and some lines of the document prepared with a mechanical typewriter (e.g. an Olympia AG coming from Wilhelmshaven, Germany, with a two-color ink tape: black and red), and graphics, sketches and formulas executed by hand. In special cases, one might have the privilege of using an electronic typewriter (e.g. a Brother CE-30, made in Taiwan), which offered the possibility of correcting some mistakes before typing on the paper. In this stage, one used photocopies for mass reproduction; therefore, graphics should be conceived only in black and white (B&W) and should be drawn on good paper with ink.

The artistic part of this was to use different widths and types for lines, different textures for fills, and different font sizes. Figure 1 shows a part of the author's first scientific document, made in 1992 [6]. Observe that the black color of the letters is not uniform, being dependent on the force with which one's finger triggered the letter key and the ink tape quality or usage. Also observe the equation with superscripts, subscripts, and Greek letters done by hand; and the graphic was also done by hand.



Con un ejemplo en la figura 2 se podrá explicar otros enunciados.
Según la ecuación de Bernouli en un tubo como indica la Fig.2

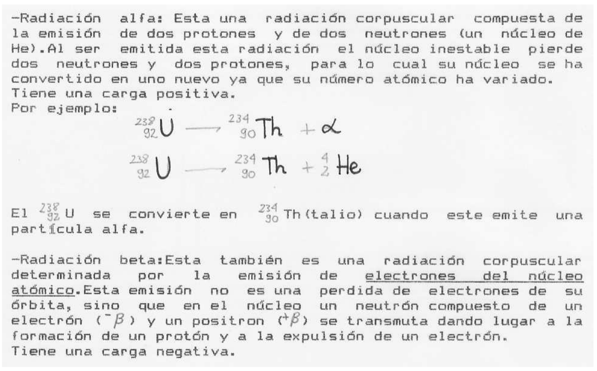
$P_1 + \frac{1}{2}\rho V_1^2 = P_2 + \frac{1}{2}\rho V_2^2$
donde $\frac{1}{2}\rho$ es una constante

$P_1$
$P_2$
Corriente de aire
$V_1$
$V_2$
Fig. 2

Es evidente que en $V_2$ la velocidad deve ser mayor en $V_1$ ,luego para que se conserve la igualdad,la presión ($P_2$) debe ser menor que la ($P_1$).
■ En concluciones- Cuando aumenta la velocidad,disminuye la presión y viceversa.

**Figure 1**: Part of a document presented at a science fair, done with typewriter and by hand [6]

A few novice researchers had first contact with a computer word processor in the late 1980s, e.g. WordStar 4.0 under the CP/M operating system. But, in the early 1990s — as the present author was finishing secondary school age and in the initial stages of a university bachelor's course — some NRs had the opportunity to use *friendly* word processors on personal computers (e.g. Word Perfect 5.1 under the Disk Operating System (DOS) version 4.0).



-Radiación alfa: Esta una radiación corpuscular compuesta de la emisión de dos protones y de dos neutrones (un núcleo de He).Al ser emitida esta radiación el núcleo inestable pierde dos neutrones y dos protones, para lo cual su núcleo se ha convertido en uno nuevo ya que su número atómico ha variado.
Tiene una carga positiva.
Por ejemplo:

$$^{238}_{92}U \longrightarrow {}^{234}_{90}Th + \alpha$$

$$^{238}_{92}U \longrightarrow {}^{234}_{90}Th + {}^{4}_{2}He$$

El $^{238}_{92}$U se convierte en $^{234}_{90}$Th(talio) cuando este emite una partícula alfa.

-Radiación beta:Esta también es una radiación corpuscular determinada por la emisión de electrones del núcleo atómico.Esta emisión no es una perdida de electrones de su órbita, sino que en el núcleo un neutrón compuesto de un electrón ($^-\beta$) y un positron ($^+\beta$) se transmuta dando lugar a la formación de un protón y a la expulsión de un electrón.
Tiene una carga negativa.

**Figure 2**: Part of the document presented to another science fair, done with Word Perfect 5.1 under DOS 4.0 and by hand [7]

The problems of this system (both hardware and software) were that in the region only English-language keyboards were available, and the text processor was initially available only for English. Therefore, in order to use the system in Spanish language, the ñ and vowels with accent (e.g. á, ó, í) had to be introduced by a combination of the Alt key plus the appropriate three numbers taken from the American Standard Code for Information Interchange (ASCII). So, it was common to see a laminated cheat sheet above the function keys on a keyboard.

Formulas and sketches were normally a combination usage of the word processor application tools — which were not good for complex sketches and formulas — and handwriting; and graphics could be prepared with other software (e.g. Harvard Graphics, which was not the proper tool, but it did manipulate vector graphics) or by hand. At this point, one began to use dot matrix printers, but they were very slow. For example, in order to print a 32-page document on a Wang Labs PM 016/160 dot matrix printer, one consumed five hours listening to the particular sound of those printers and taking care that no paper problems arose. Later, this printer type became faster in the region, e.g. the EPSON LQ-300.

Figure 2 shows an example from a document created in 1992 in the environment described above. Observe that even though the document was made with a word processor, the quality remains close to the document prepared two years ago (Figure 1) because it was hardware-dependent (i.e. on the printer type, which was a dot matrix printer that also uses an ink tape).

Also, observe in Figure 2 that some special characters in the Spanish language — for example the `í` letter — are printed with a different quality (e.g. the last but one word *partícula* of the paragraph after

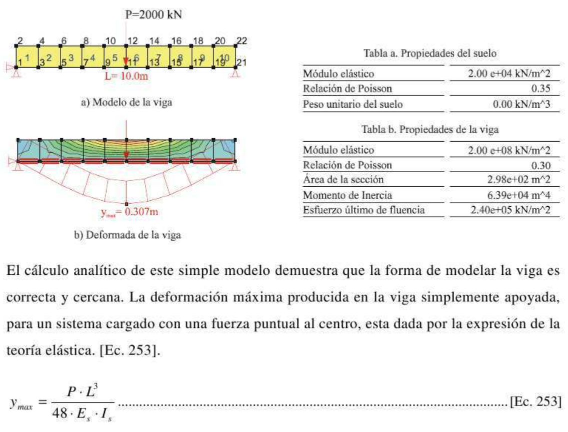

P=2000 kN
L= 10.0m
a) Modelo de la viga
$y_{max}$= 0.307m
b) Deformada de la viga

Tabla a. Propiedades del suelo

| | |
|---|---|
| Módulo elástico | 2.00 e+04 kN/m^2 |
| Relación de Poisson | 0.35 |
| Peso unitario del suelo | 0.00 kN/m^3 |

Tabla b. Propiedades de la viga

| | |
|---|---|
| Módulo elástico | 2.00 e+08 kN/m^2 |
| Relación de Poisson | 0.30 |
| Área de la sección | 2.98e+02 m^2 |
| Momento de Inercia | 6.39e+04 m^4 |
| Esfuerzo último de fluencia | 2.40e+05 kN/m^2 |

El cálculo analítico de este simple modelo demuestra que la forma de modelar la viga es correcta y cercana. La deformación máxima producida en la viga simplemente apoyada, para un sistema cargado con una fuerza puntual al centro, esta dada por la expresión de la teoría elástica. [Ec. 253].

$$y_{max} = \frac{P \cdot L^3}{48 \cdot E_s \cdot I_s} \quad \text{[Ec. 253]}$$

**Figure 3**: Part of the BSc thesis, this done with MS Word under Windows 3.1 and Graphics with CorelDraw 9 exported to a medium or low raster JPG format [8]

the equations). Also, the equations

$$^{238}_{92}\text{U} \rightarrow^{234}_{90} \text{Th} + \alpha \qquad (1a)$$

$$^{238}_{92}\text{U} \rightarrow^{234}_{90} \text{Th} +^{4}_{2} \text{He} \qquad (1b)$$

and in-text variable symbols (e.g. `...un positrón` ($^+\beta$) `se transmuta...`) still needed human intervention.

At the final stages of the university bachelor course, at the end of the 1990s, things went better: the Windows 3.1 (or higher) Operating System (OS) in Spanish came with the not-well-known — at that time — MS Word text processor, and now keyboards were available for Spanish. Also, this text processor included programs to prepare sketches and formulas, which were more flexible, allowed more complex cases, and, best of all: they supported full color. Also, because color bubble jet printers were accessible to an NR, one started to conceive of full-color figures in scientific and technical documents.

Figure 3 shows the full-color graphics (grayscaled for *TUGboat* hardcopy, though) made in a vector program, which decreased in quality when exporting to a raster format; but this was necessary because the text processor did not import properly and exactly the given graphic, even when trying to use its own EPS vector format. The equation editor of the word processor allowed these simple equations to be executed well.

For the first decade of this new century, use of these tools has been accepted by some scientific researchers and most industrial professionals, with the difference being that there was more diversity of fonts, document templates, and printing resources. This happened more due to hardware improvements (rather than improvements in the commercial software), which now allowed storing huge amounts of data in memory during editing; therefore, one could insert in a What You See Is What You Get (WYSIWYG) document file — for example — high resolution raster figures and be unlikely to suffer a program crash and consequently suffer a file corruption.

Nowadays, the quality of hardcopy documents has been improved, because laser color printers are more accessible for NRs than in the past; therefore, more color texts have been produced and they can also be reproduced economically in small quantities.

For example, most Master's degree dissertations in civil engineering in Latin American universities have been prepared with the combination of proprietary WYSIWYG software (i.e. word processor, spreadsheet, and vector graphics editor), known as *office* programs. Figure 4 shows part of an MSc thesis prepared with these programs: reference citations, reference lists, and lists of variables, abbreviations, and acronyms were handled manually; on the other hand, figure and table referencing, and equation numbering, were created automatically.

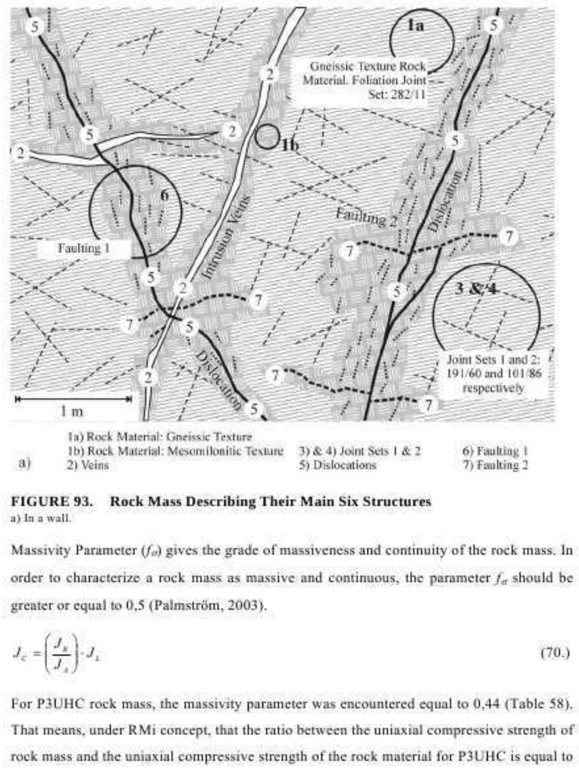

1a
Gneissic Texture Rock Material. Foliation Joint Set: 282/11
1b
6
Faulting 1
Intrusion Veins
Faulting 2
Dislocation
3 & 4
Joint Sets 1 and 2: 191/60 and 101/86 respectively
1 m
a)
1a) Rock Material: Gneissic Texture
1b) Rock Material: Mesomilonitic Texture
2) Veins
3) & 4) Joint Sets 1 & 2
5) Dislocations
6) Faulting 1
7) Faulting 2

**FIGURE 93. Rock Mass Describing Their Main Six Structures**
a) In a wall.

Massivity Parameter ($f_\sigma$) gives the grade of massiveness and continuity of the rock mass. In order to characterize a rock mass as massive and continuous, the parameter $f_\sigma$ should be greater or equal to 0,5 (Palmström, 2003).

$$J_C = \left(\frac{J_R}{J_A}\right) \cdot J_L \qquad (70.)$$

For P3UHC rock mass, the massivity parameter was encountered equal to 0,44 (Table 58). That means, under RMi concept, that the ratio between the uniaxial compressive strength of rock mass and the uniaxial compressive strength of the rock material for P3UHC is equal to

**Figure 4**: Part of an MSc thesis, done with MS Office under Windows XP and graphics with CorelDraw X4 exported in high-resolution raster (JPG) format [9]

But the biggest drawback of using WYSIWYG programs was that users did not learn about document structuring. They used a text processor as a simple electronic typewriter with improved graphical capabilities. This situation is becoming less common in recent years; now, users are considering the importance of hierarchically structuring documents, with the goals of facilitating exporting to XML and of disseminating documents via the Internet.

## 2.1 Digital Dark Age also touches NR

Past documentation stored in an encrypted coding and unable nowadays to be reopened, because there is no access to the hardware and software with which it was created, has suffered a permanent loss — even when the stored data is well preserved. If this situation occurs in a period between the time when the first electronic document was stored without preventing this situ, to the time when the last electronic document was stored before this situation is solved, then that period so determined can be considered a Digital Dark Age (DDA), because nobody in the future will be able to know what humans had documented in that period.

Because humans have not in fact resolved this situation, we are now inside a DDA. On a small scale and for particular usage, NR are also suffering the consequences of DDA. For example, some plots of the scientific document shown in Figure 2 [7] were made using proprietary software (Harvard Graphics, as mentioned above), which stored the files with the extension CHT. To date, no emulator can visualize these graphics, even though the file is stored perfectly safely on a hard disk.

NRs being in a DDA also causes the loss of many of their old documents, which are frequently more vulnerable than most others. This is because most documents developed by NRs are not of global interest; therefore, those documents are not stored in data centers but instead on a home system. In 20 years of managing personal scientific documentation, the present author — an NR — has suffered two important losses of documentation: first in the year 2000, when the entire computer was stolen; the other in 2007, when the author's first old PC computer and the programs' floppy disks — preserved in order to minimize personal DDA — were erroneously sold by the author's father.

Nowadays the situation is becoming less problematic for NR, since large external companies offer well-implemented digital data storage alternatives (e.g. Dropbox).

## 2.2 The experience with the TEX family

TEX is a computer typography program, free and open source in today's terminology, created in 1978 by Donald E. Knuth of Stanford University, which has revolutionized digital typesetting for scientific publishing and transformed the process of putting mathematical ideas on paper [2, 3].

Since then, many improvements and proposals have followed from it, creating the so-called TEX family (also denoted (LA)TEX). In general, this family has shown in its time that it was designed with two main goals in mind: to allow anybody to produce high quality documents; and to provide a system that would give exactly the same results on all computers, now and in the future [4].

One advantage of (LA)TEX, among many others, is that one can use escape sequences for characters beyond basic ASCII; therefore, one can cover all possible special characters with only 128 characters defined in the OS. Even though this limit was superseded many years ago with improved OSs, the use of only ASCII characters can still be very useful when one wants to put document information in structured databases: for a word-searching process, it is better to use the least possible number of different characters, as a broader range can easily introduce more errors.

The personal experience of the author with TEX began in 1996 when a friend seriously illuminated the benefits of the new "free" OS Linux — erroneously conceived by the author as *free of charge* — because the text processor software it offered was LATEX-based. In those years, another friend that came from Switzerland to visit his native country (i.e. Bolivia) gave the author three or five $3\frac{1}{2}$ inch floppy disks with a program that dealt well with scientific documents (i.e. Scientific Workplace under Windows 3.0 OS). The present author tried to make use of this, but because tutorials were lacking, finally he abandoned the program.

Unfortunately, the author ignored this clever advice, and 14 years passed before he recognized that the pair GNU/Linux and TEX had an important ideology behind them (see for example [5]) and improved advantages for NRs.

## 3 Universal Editable Format

Unfortunately, novice researchers coming from Latin American countries generally still believe that encoded binary files coming from popular office packages are the correct candidate as a universal format for document editing. In the lexicon of individuals, one finds phrases such as "send me a DOC file, please" when one wants to receive a document.

Also, in many university libraries of Latin America, the format they have adopted as universal is, erroneously, MS Word DOC format in its 2003 version. Strangely, they do not at least adopt the MS Word DOCX format, which permits exporting DOCX to XML. A disadvantage of adopting DOCX is that one must run the proper program to do the transformation to XML, and therefore one also is still dependent on commercial software.

Other universities adopted Rich Text Format (RTF), but this does not support graphics, formulas, tables, etc; therefore, this solution is even worse than the first, and worse also than PDF.

In general terms, a Universal Editable Format (UEF) need not necessarily be known by everybody. Instead, it should meet the following basic requisites:

- persist across time without losing information (i.e. be 100% identical on all machines and avoid DDA).
- be freely accessible anytime and anywhere (i.e. not be proprietary);
- be independent from hardware;
- not be encrypted or represented using binary codes; therefore, be readable by any text editor, even the simplest one;
- have commands understandable by any individual after reading available free[1] documentation;
- the binary programs that parse the commands should run under any OS.

Particularly for an NR, the TEX family can be a serious candidate because:

- it has persisted for many years without important weaknesses;
- it is the oldest system that has been a reliable, free, de facto standard for decades;
- it has a great number of users; and
- it can be used by any individual, including those without enough money to buy commercial editing program licenses, but with access to a computer and the Internet (e.g. pre-university, bachelor, and postgraduate students, i.e. NRs).

### 3.1 Document interchange formats

Electronic documentation and dissemination was not commonly used by NRs in Latin America until the middle of the 1990s. Before the Portable Document File (PDF) format was devised, other formats were available in the world — for example the DeVice Independent format (DVI) developed as part of the TEX family, and the PostScript format coming from Adobe — these were not known by the novice researchers in this region.

It was only after the PDF format and the Acrobat reader came into prominence that people encountered a viable use for and storage of electronic

[1] Free in the sense of freedom

documentation. The interchange of documents nowadays in this region is handled well via this format. And, since 2008 this format has become even more popular, after it was *liberated* by Adobe, made an ISO standard and in general available for any individual to make, use, sell and distribute PDF-compliant implementations.[2] Since then, FOS software has been developed with considerable added value, for example, PDF editing and electronic signatures.

The PDF format can be a good option to preserve a document as it was conceived by its author(s) and for electronic libraries (i.e. static preservation); but perhaps is not the proper one to be adjustable to the flexibility of web pages, and for the more volatile and interactive areas of the web which require dynamic document preservation and/or continuing editing, as for example wiki pages (e.g. Wikipedia) [2]. Also, browsers cannot display PostScript or PDF files without the aid of extra software (i.e. add-ons, many of which are proprietary), which in turn causes many readers to choose to download such files and view them offline or print them.

In the near future — not to say *in the present* — portable devices (e.g. tablets, cell phones, smart phones) will be more popular, and programs *in the cloud* will be used more; therefore, it is possibly necessary to adopt another format for document interchange for this new technological tendency.

With various scripts or programs, documents created by TeX can be transformed to XML which can be a first choice candidate for an NR to have a good interchange document format meeting such Internet-related requirements.

## 3.2 The case of graphics

In the 1990s, one typically used so-called vector graphic editor programs (e.g. CorelDraw and Adobe Illustrator for Windows, which were and are commercial programs) to execute figures for documents. Unfortunately, in the early years of the Internet, information about the existence of FOS programs was lacking. Therefore, piracy of the most famous graphic programs was the common "solution" for NRs in Latin America during those years.

Other NRs erroneously used other sorts of proprietary programs to make graphics for their documents, for example presentation programs (e.g. PowerPoint) and two-dimensional computer aided design (CAD) programs (e.g. AutoCAD). This was typically because they had no choice other than to use the software they had available, legal or otherwise.

[2] But this donation by Adobe did not follow the free (as in freedom) or open software concepts, because the source code was not liberated.

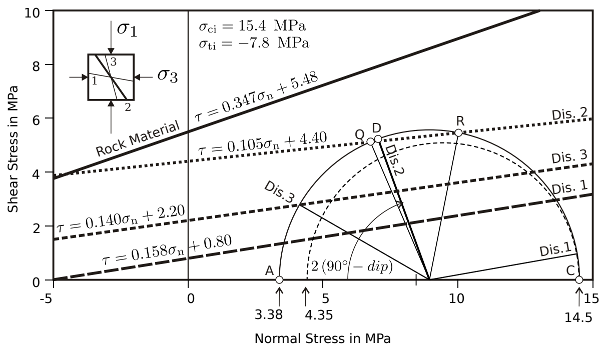


**Figure 5**: A graphic from the PhD thesis, created with Inkscape 0.47 under i486-PC-Linux-GNU (Debian Squeeze 4.4.5-8) and exported to PDF format [10]

One principal limitation found by NRs working this way was that they became dependent on proprietary file formats, which were — and still are — difficult to convert to a hypothetical universal editable format for 2D and 3D vector graphics. This might be the reason why some CAD proprietary file formats became so *important* for interchange of graphics (e.g. the AutoCAD DWG format); at the same time, they are a poor choice for interchange, because the structure of the DWG format regularly changes with new releases, and has never been made publicly available.

This situation continues today, but with fewer devotees. Nowadays, NRs more often use less complicated and costly proprietary software, or have decided on well-developed FOS alternatives.

With the advent of the new century, a new vector graphics specification was developed for public use: Scalable Vector Graphics (SVG), based on XML, which nowadays may be considered a universal vector editable format, being widely supported on the Internet and in applications. Unfortunately, SVG still supports only 2D graphics, but efforts are being made to cover 3D as well in the future.

SVG figures can be created and edited with any text editor, but it is often more convenient to create and edit them with a natural human interface for graphics. In the area of FOS software, one interesting program that deals with this format is the excellent Inkscape, which is available for all major OSs. Figure 5 shows a scientific figure developed entirely with this program using the SVG format; observe that TeX equations are included.

## 3.3 (LA)TeX classes as document managers

Returning to discussion of the TeX family, for many years the *class* concept has been used for (LA)TeX-based documentation, by creating a *class file* (CLS extension by default). Although most commonly known for LaTeX, the concept can be applied anywhere in the TeX family.

This file — if properly designed — permits an NR to create a document structure and formatting rules once (i.e. a document template), and then re-use this many times. This makes it possible to create uniform documents with respect to format and structuring, broadly used for book collections, journal articles, institutional reports, university dissertations and theses, and by extension any serialized document.

A CLS file is used under the TeX family concepts, formats and engines. Thus, its use can result in high quality document creation and a suitable document manager, since in a class file one can, for example:

- restrict the document formatting and the document structure;
- define the text and mathematics font styles;
- define table formatting;
- limit the number of chapters and appendixes;
- use a specific format for bibliographic references;
- define proper format for the frontmatter, main text, and backmatter; and
- define the page size, margins, headings and footers, and numbering.

The elaboration of a CLS file for (LA)TeX is not terribly difficult, but nor is it an easy task; by personal experience, it can be done by people who have used (LA)TeX for at least one year, with the help of the plethora of information on the Internet (a situation that was not possible 15 years ago). On the other hand, the use of an existing CLS file within (LA)TeX is a very easy procedure, which any NR beginner can do.

Therefore, as this is an excellent tool, the creation of any CLS file for any commonly used document at public institutions can be useful for others. For this reason, over the last few years (perhaps five years), the availability of CLS files for universities' documentation (i.e. dissertations and theses) has been increasing.

An observation about this last statement: such dissemination is not usually driven by the libraries or other university offices; instead, it is usually comes from the student community, with or without the aid of a professor (i.e. an NR). Regarding this, it is the mathematical departments of the universities that have typically promoted the use of (LA)TeX and who have created and disseminated CLS files.

Following this last-mentioned tendency, in the following section it is briefly presented how a CLS script file can be structured and the minimum environments it might provide. This CLS file was developed by the author for his doctoral thesis, and his general conclusion — among others mentioned in the next section — that emerged when working with the TeX family was that finally, after 20 years, he has found the proper tool for scientific documentation.

## 4 The unbDscThesisEng TeX family class

The `unbDscThesisEng` TeX family class is an unofficial template (i.e. not approved by the university) for the English language version thesis of the Geotechnical Postgraduate Program of the *Universidade de Brasília*, Brazil.[3] It is composed of a main CLS script file which requires other secondary files and a particular directory structure.

This class was based on the `book` class with modifications to accomplish local formatting requirements. It could also be properly used for the Portuguese language by doing some modifications not yet included in this version. The CLS file is free software that is released under the terms of GNU General Public License Version 3, as published by the Free Software Foundation.

The class permits the student to use the following eleven environments:

- `princover` for making the main cover of the thesis by attaching a `background.pdf` file;
- `maketitle`, a redefinition of the original in the `book` class, that permits making the title page according to the university rules and format;
- `approbationpage` for the page with the jury members' names and signatures;
- `catalogingpage` for the page of the thesis cataloging and copyright information;
- `dedicatory` for the dedicatory text;
- `acknowledgements` and `agradecimentos` for the acknowledgments in English and Portuguese;
- `abstract` and `resumo` for the abstract in English and Portuguese;
- `tableofcontents` for the TOC;
- `listoffigures` and `listoftables` for the lists of figures and tables;
- `listofabbrevsymbs` for the list of abbreviations and symbols (separated by Latin and Greek characters);
- `listofreferences` for the references after the main matter;
- `invitationpage` for a page with the date and time of the thesis presentation;
- `reportpage` for a page with general report information of the document.

In order to use any of the environments, the `unbDscThesisEng.cls` file should reside in the document's main directory, and 48 variables are available

[3] This class is at Version v1.0 as of 2012/15/08

for the user to fill. These variables are filled by the user in a separate TEX file (i.e. the `initials.tex` file), in order to avoid editing the CLS file.

Also, a main TEX file was designed (`aaaThesis.tex`), in which the user can: define the page size and its margins; define the text size among 10 pt to 12 pt; define if the document should be printed as two-side or single-sided; introduce other (LA)TEX packages; redefine some commands; and insert the document text. All the figures should be placed in a directory called `FIGURES`; and front-, main-, and backmatters should be inside the directories `FRONT_MTR`, `MAIN_MTR`, and `BACK_MTR`, respectively. The bibliographic references should be written in BIBTEX format and be named `bibliography.bib`. The bibliographic style used for this class is `chicnarm.bst`, recommended to reside in the same document main directory.

Because this class uses the `nomenclature` package to manage the list of variables, a `nomencl.cfg` file should also be in the document main directory.

The following listing shows the directory structure for the `unbDscThesisEng` class and the minimal required files. The user interface used in this example listing is the KDE Integrated LATEX Environment (KILE) under KDE Platform Version 4.4.5 for GNU/Linux; the `unbDscThesisEng.kilepr` file also defines the complete document structure.

```
drwx  BACK_MTR
drwx  FIGURES
drwx  FRONT_MTR
drwx  MAIN_MTR
-rw-  aaaThesis.nlo
-rw-  aaaThesis.nls
-rw-  aaaThesis.pdf
-rw-  aaaThesis.tex
-rw-  background.pdf
-rw-  bibliography.bib
-rw-  chicnarm.bst
-rw-  initials.tex
-rw-  invitationBackground.png
-rw-  nomencl.cfg
-rw-  nomencl.dtx
-rw-  nomencl.ins
-rw-  nomencl.ist
-rw-  unbDscThesisEng.cls
-rw-  unbDscThesisEng.kilepr
```

Experiences using this class for the doctoral thesis brought about improvements in: text formatting; correct use of SI units according to the proper rules; good mathematical symbolization of variables; robust index generation — especially with the variables index; robust reference structuring and full hypertextual citations. All of these in general improved the document in quality and also reduced the time needed for its elaboration.

Experiences using the SVG format for graphics in the thesis brought about: homogeneous text formatting in all figures; retaining in the graphics the same mathematical variables defined in the documents (by using the LATEX equation rendering extension of Inkscape); easy editing independent from the text of the document; and high quality output, being vector graphics and not bitmaps.

## 5 Final comments

Comments given here are based on the author's personal experience in his region and language, having had the opportunity to deal with the scientific and technical document creation for some 20 years, and knowing the popular text processors and typesetting programs.

In the historical narration of the author, it appears that milestones of that theme (i.e. use of computer typesetting, use of DOS, Windows OS, bubble and laser jet printers, and other issues mentioned in Section 2) in Latin America came later than in other regions; but this is not accurate. The true reason is that the abovementioned milestones came relatively late to NRs, who are ordinary people: university students and teachers. It is likely that these milestones came sooner to the region in more specialized areas, for example, high-level research institutions.

The TEX family tries to reduce Digital Dark Age (DDA) hazards for the NR. Preventing the DDA for FOS formats could be less costly and time-consuming than doing the same for proprietary formats. The latter requires signing agreements with the format owners, who try to market their work by praising it as *social labor* when preventing DDA, while in fact they are part of the DDA problem; proprietary formats are by nature against DDA prevention.

What seems to be true is that the correct way forward in scientific document preparation for NRs is through the FOS concepts and by typesetting programs such as the TEX family. When mentioning this duality, FOS concepts & TEX family, it is inevitable to hear about the robust ideology they have behind them; and probably from this duality can emerge the proper UEF sought by the NR.

Perhaps the tendency of individuals in using SVG format for graphics, and TEX for text, tables, equations and references could mark a future tendency in using XML format for scientific documents (e.g. DocBook). But TEX is evidently easier to learn than XML, and at any rate, there exist proper ways to translate TEX to XML.

Also, in the near future, it is possible that the ePub format — a structured compressed XML based format broadly used for mobile devices — can be another possible interchange document format for the NR, rather than PDF. This can work because the Internet is a channel for distributing publications and preprints in many disciplines, as well as becoming a venue for less formal jottings and conversations using mobile technology.

At this moment, TeX has become the *de facto* standard text processing system in many academic high-level scientific and research institutions, while — in parallel — it is increasingly the choice of NRs in developing countries (e.g. Latin America). The example of usage of the `unbDscThesisEng` class is one of many found in literature which improved a scientific document conception, elaboration, and competitive dissemination, which has a good opportunity to prevail for many years without being caught by DDA. But TeX family classes should be better promoted by universities, who should also define more specific and rigorous document elaboration policies.

Finally, digital storage is easy but digital preservation is not. Preservation means keeping the stored information cataloged, accessible, and usable on current systems, which requires constant effort and expense. One cannot reverse the digitization of everything; what one has to do is convert the design of software from brittle to resilient, from heedlessly headlong to responsible, and from time-corrupted to time-embracing [1].

## References

[1] S. Brand. Escaping the digital dark age. *Library Journal*, 124(2):46–49, February 1999.

[2] H. Brian. Writing math on the web. *American Scientist*, 97(2):98–102, March–April 2009.

[3] S. Ditlea. Rewriting the Bible in 0's and 1's. *Technology Review*, 102(5):66–70, September–October 1999.

[4] A. Gaudeul. Do open source developers respond to competition?: The (LA)TeX case study. *Social Science Research Network*, (908946), March 2006.

[5] L.E. Rosen. *Open Source Licensing: Software Freedom and Intellectual Property Law.* Prentice Hall, Upper Saddle River, NJ, 1st edition, 2004.

[6] L.O. Suarez-Burgoa. Estudio aerodinámico, fuerza de sustentación y resistencia en un avión. Technical report, Colegio Alemán Mariscal Braun, La Paz, Bolivia, October 1990.

[7] L.O. Suarez-Burgoa. Estudio nuclear de átomos radiactivos y efectos de la radiactividad. Technical report, Colegio Alemán Mariscal Braun, La Paz, Bolivia, October 1992.

[8] L.O. Suarez-Burgoa. Consideraciones para estabilizar taludes por medio de un sistema de tablestacado y enrejado vegetado. BSc. thesis, Universidad Mayor de San Andrés, La Paz, Bolivia, June 2001.

[9] L.O. Suarez-Burgoa. Rock mass mechanical behavior assessment at the Porce III underground hydropower central, Colombia, South America. MSc. thesis, Facultad de Minas, Universidad Nacional de Colombia, Medellín, Colombia, February 2008.

[10] L.O. Suarez-Burgoa. *A qualitative physical modeling approach of rock mass strength.* Ph.D. thesis, Departamento de Engenharia Civil e Ambiental, Universidade de Brasília, Brasília DF, August 2012. Publication G.TD-079/12.

◇ Ludger O. Suarez–Burgoa
Universidad Nacional de Colombia
Fac. Min., Esc. Ing. Civ.
Cl. 65 #78 – 28, Bl. M1 Of. 320
Medellín, Colombia
`losuarezb (at) unal dot edu dot co`
`http://geomecanica.org/`

# Macro memories, 1964–2013

David Walden

## Contents

## Introduction

In the summer of 2013, I was looking at a 1973 listing of the ARPANET IMP (Interface Message Processor) program[1] which makes extensive use of macros. This caused me to muse about the various macro processors I have used over the past 50 years which, in turn, led to this note.

This note is not a thorough study, extensive tutorial, or comprehensive bibliography about macro processors. Such descriptions have already been provided by, for instance, Peter Brown, Martin Campbell-Kelly, John Metzner, and Peter Wegner in their longer presentations of the topic.[2,3,4,5] Instead, the macro technology thread I follow herein is guided by the order in which I used or studied the various macro processors. I hope this is usefully representative of the scope of macro processor technology.

I have three reasons for writing this note. (1) I haven't seen much new written about macro processors in recent years (other than what is on the web); thus, it is perhaps time for a new paper on this sometimes under-appreciated topic. (2) Computer professionals and computing historians who have come to their fields only in the last decade or two may not know much about macros, and this is a chance to share my fondness for and perspective on macros. The citations in the endnotes also may be a useful starting point for further study of macros, and maybe these notes will rekindle memories for other long-time computing people like me about some of their own experiences. (3) For (LA)TeX users who may not be computer programmers or familiar with other macro processor systems and who accomplish impressive things using TeX macros, this note sketches the long technical history of which TeX macros are a part.

I assume most readers know what macros are, but just in case: Typically one gives a name to a string of text, e.g., "`\define\Macroname{Textstring}`"; then each time "`\Macroname`" is found ("called") in the input text stream, it is replaced by "`Textstring`".

The macro definition can involve the additional substitution of text specified when the macro definition is called. For example,

```
\define\Name#1{His name is #1}
```

defines a macro named "`Name`" where "`#1`" indicates a parameter to be substituted for when the macro is called. The macro might be called with "`John`" as the substitution text, as in the following

```
\Name{John}
```

which would result in

```
His name is John
```

In addition to the "`#1`" indicating where a substitution is to take place in the macro definition when the macro is called, in this example the "`#1`" immediately after the macro name indicates there is one such substitutable parameter. If there were more than one such parameter, a list such as "`#1#2#3`" would appear after the macro name in the definition, specifying three such parameters.

Donald Knuth in the index to Volume 1 of *The Art of Computer Programming* gives this succinct definition relating to macros: "Macro instruction: Specification of a pattern of instructions and/or pseudo-operators that may be used repeatedly within a program."

## 1 McIlroy's 1960 ACM paper

I'm pretty sure that while I was still in college at San Francisco State (1962–64) and using an IBM 1620 computer, I had no concept of macros. The IBM 7094 at MIT Lincoln Laboratory, my first employer after college (starting in June 1964), may have had a macro assembly capability, but I don't think I ever used it.

Probably my first contact with the concept of macros was when an older colleague at Lincoln Lab gave me his back issues of the *Communications of the ACM* (and I joined the ACM myself to get future issues of the CACM). In one of these back issues, I read the article by Doug McIlroy on macros for compiler languages.[6]

McIlroy has the following example of defining a macro (although I am using an equal sign where he used an identity symbol):

```
ADD, A, B, C = FETCH, A
               ADD, B
               STORE, C
```

where `ADD`[7] is the macro name, and `A`, `B`, and `C` are the names of macro arguments to be filled in at the time of the macro call, and the three lines of code are what is substituted. Thus, McIlroy shows this macro being called with the sequence

```
ADD, X, Y, Z
```

resulting in the following:

```
FETCH, X
ADD, Y
STORE, Z
```

This style of macro definition uses symbolic names for the substitutable parameters, which can be useful in remembering what one is doing with long macro definitions. However, it is also a bit more complicated to implement such symbolic macro parameter names compared with using special codes such as "`#1`".

McIlroy's 1960 paper goes on to show examples of macros in an ALGOL-like language and to explain the benefits of various features of macro processors. For instance,

```
macro exchange(x,y;z) :=
    begin
        begin integer x,y,z;
             z:=y;
             y:=x;
             x:=z;
        end exchange x and y
    end exchange
```

defines a macro which, if called with

```
    exchange(r1,ss3)
```

results in

```
    begin integer r1,ss3,.gen001
          .gen001:=r1;
          ss3:=r1;
          r1:=.gen001;
    end exchange r1 and ss3
```

Note that a special temporary register, `.gen001`, was created to replace `z` which was defined following the semicolon in the parameter list.

McIlroy's paper also has a summary list of "salient features" [the comments below in square brackets are my notes on McIlroy's list]:

1. definitions may contain macro calls
2. parenthetical notation for compounding calls [e.g., so arguments to macro calls can include multiple items separated by commas]
3. conditional assembly
4. created symbols [e.g., so labels or local variable names in the body of a macro definition are unique for each call of the macro]
5. definitions may contain definition schemata
6. repetition over a list [see the example in the discussion of Midas in section 4]

Apparently Bell Labs was a particular hotbed of macro activity in those early days. In a memorial note for Douglas Eastwood,[8] Doug McIlroy recounts:

> On joining the Bell Labs math department, I was given an office next to Doug Eastwood's. Soon after, George Mealy ... suggested to a small group of us that a macro-instruction facility be added to our assembler ... This idea caught the fancy of us two Dougs, and set the course of our research for some time to come. We split the job in half: Eastwood took care of defining macros; McIlroy handled the expansion of macro calls.
>
> The macro system we built enabled truly astonishing applications. Macros took hold in the Labs' most important project, electronic switching systems, in an elaborated form that served as their primary programming language for a couple of decades.
>
> Once macros had been incorporated, the assembler was processing code written wholesale by machine (i.e., by the assembler itself) rather than retail by people. This stressed the assembler in ways that had never been seen before. The size of its vocabulary jumped from about 100 different instructions to that plus an unlimited number of newly defined ones. The real size of programs jumped because one human-written line of code was often shorthand for many, many machine-written lines. And the number of symbolic names in a program jumped, because macros could invent new names like crazy.

By the way, Rosen's book[7] also had a paper (pp. 535–559) by George Mealy that touched on macros: "A Generalized Assembly System (Excerpts)".

## 2 Some prior history of macros

McIlroy's paper also hints at some of the history of macro processors including half a dozen references[9] to prior macro processors; they were becoming fairly widespread by the early 1960s. Lots of people were thinking about macros and macro processing by 1960. In section 6 of his paper, McIlroy says,

> ... Conditional macros were devised independently by several persons beside the author in the past year. In particular, A. Perlis pointed out that algorithms for algebraic translation could be expressed in terms of conditional macros. Some uses of nested definitions were discovered by the author; their first implementation was by J. Benett also of Bell Telephone Laboratories. Repetition over a list is

> due to V. Vyssotsky. Perlis also noted that macro compiling may be done by routines to a large degree independent of ground language. One existing macro compiler, MICA (Haigh), though working in only one ground language is physically separated from its ground-language compiler. An analyzer of variable-style source languages exists in the SHADOW routine of M. Barnett, but lacks an associated mechanism for incorporating extensions. Created symbols and parenthetical notation are obvious loans from the well-known art of algebraic translation.

Donald Knuth and Luis Trabb Pardo also touch on the history of macros in their paper "The Early Development of Programming Languages".[10] Early in the paper,[11] they note that Turing's 1936 paper on a universal computing machine used a notation for programs which amounted to being "macroexpansions or open subroutines". Later in the paper,[12] they say that Grace Hopper in 1951 came up with the "idea that pseudocodes need not be interpreted; pseudocodes could also be expanded out into direct machine language instructions." Later on the page they note, "M.V. Wilkes came up with a very similar idea and called it the method of 'synthetic orders'; we now call this macroexpansion."[13]

## 3 Strachey's General Purpose Macrogenerator

At the time I joined the ACM to receive the CACM, I also subscribed to *The Computer Journal*. In this I studied Christopher Strachey's GPM (General Purpose Macrogenerator).[14] The paper presents Strachey's macro processor and its possible uses. Then the paper explains how it is implemented. Finally, it has the code for the CPL language (sort of ALGOL-like) which can be transliterated to implement GPM in any other computing language.

GPM was a change in the way macro definitions and calls were formatted from the series of macro processors originally developed at Bell Labs in what I will call the McIlroy style. These early assemblers and the macro processors tended to be shown in columns with keywords (`DEFINE`, a defined macro name, `IRP`, etc.) being recognized by the processor as a special symbol and the other parts of the definition or call being detected by their separation with spaces or commas, or perhaps bracketing parentheses. In fact, many of the early macro processors were embedded parts of an assembler or language compiler. GPM indicated its macro definitions and calls and their arguments with unusual characters, and it was independent of any particular language — a possible preprocessor for any other language or as a stand-alone string processor.

Here is a simple definition in GPM:

```
§DEF,REFORMATNAME,<LAST=~2, FIRST=~1>;
```

It could be called like this:

```
§REFORMATNAME,David,Walden;
```

which would produce the output:

```
LAST=Walden, FIRST=David
```

Note that the GPM approach to macro definitions does not specify how many substitutable parameters there are. Note also that bracketing macro definitions and macro calls with special symbols ("`§`", "`;`") makes it a bit simpler for definitions and calls to occur anywhere in the input stream.

Strachey's paper is a wonderful and now classic article (it's a shame it resides behind an overly expensive paywall for the journal). By introducing the macro processor and its uses, describing its implementation, and then providing the code for its implementation, Strachey's paper is a superb model for presenting a programming language. This was perhaps possible because Strachey, purportedly a genius programmer, had managed a very general and beautiful implementation. The CPL code was only two double-column journal pages long; and, according to the history of the `m4` macro processor,[15] fit into 250 words of machine memory.

The just-mentioned `m4` history also touches on the influence of GPM on later macro processors. Also, GPM was used by later authors as an illustrative example of a macro processor.[2,16]

Some readers by this time may be asking, "But how is a macro processor implemented?" One can sketch this intuitively. Text in the input stream to the macro processor that is not a macro definition or macro call is just passed on to the output stream. Macro definitions in the input stream are saved in a software data structure with their names and associated definitions. When a macro call is spotted in the input stream, the definition is pulled out of storage to replace the macro call in the output stream with the call parameters being substituted at the proper places in the definition. If all this is done using a first-in-last-out stack in the proper way, definitions within definitions, recursive calls, and so forth are possible. For a detailed description of a macro processor implementation, access Strachey's GPM paper in its journal archive or find a used or library copy of Peter Wegner's book[4] (pp. 134–144).

## 4 Midas macro processor

The next macro processor I came across (and the first I actually used) was the macro processor that was part of the Midas assembler for the PDP-1. PDP-1 Midas had its origins in MIT's TX-0 computer, all the way back to MACRO on the TX-0.

MACRO was an assembler with a macro processor capability written by Jack Dennis for the TX-0. I don't know of a manual for TX-0 macros; however, MACRO was later released by DEC with its PDP-1 computer, and Jack Dennis states[17] that he wrote the manual for that.[18]

When I asked Jack Dennis about predecessor technology to MACRO, he mentioned McIlroy's paper (which was published after MACRO was available on the TX-0 in 1959, so perhaps Jack saw a draft or preprint). Of his MACRO, Jack said,[19] "Doug's macro processor was of the string substitution sort ... Mine was different: it permitted a user to give a name to a sequence of assembly instructions, with integer parameters that would be added to instructions to create modified addresses. (Thus the essential mechanism was one's complement binary arithmetic instead of string concatenation.)"

Next on the TX-0 came Midas, which was derived by Robert Saunders from TX-0 MACRO. Then, TX-0 Midas was moved to the PDP-1.[20] The TX-0 Midas memo[21] is dated November 1962 which suggests that the PDP-1 Midas was up and running sometime in 1963, as the Midas manual for the PDP-1 says it was ported from the TX-0 where it had been running for about a year.

In any case, the PDP-1 editing, assembling, and debugging set of programs was probably the best set of interactive program development and debugging tools that were available for a mini-computer in the mid-1960s. Therefore, four of us using a Univac 1219 computer at Lincoln Lab decided to reimplement these PDP-1 tools for our 1219.[22]

Midas for the PDP-1 and our version for the Univac 1219 had macro processor definition and call formats that were similar to those in the tutorial part of McIlroy's paper, e.g., "`MACRO NAME ARG1, ARG2 (string)`" to define a macro and "`NAME X,Y`" to call it with `X` and `Y` to be substituted for `ARG1` and `ARG2` in the macro definition. For example,

```
MACRO MOVE X,Y
      (ENTAL X
      STRAL Y)
```

The above when called with

```
MOVE K,L
```

resulted in

```
ENTAL K
STRAL L
```

Midas also had what I now think of as map commands, i.e., apply some function over a list of arguments — the Midas commands `IRP` (indefinite repeat over a list of arguments) and `IRPC` (indefinite repeat over a string of characters). In our version of Midas, the `IRP` command might have been used as follows:

```
IRP A, (W1, W2, W3)
(ADD   A
)
```

expanding to

```
ADD  W1
ADD  W2
ADD  W3
```

and in another example

```
IRP  X,Y,(A,Q,B,R,C,T)
(CLA X
 STO Y
)
```

expanding to

```
CLA  A
STO  Q
CLA  B
STO  R
CLA  C
STO  T
```

The same thing could have been accomplished using the command to repeat for each character in a string of characters:

```
IRPC  X,Y,(AQBRCT)
(CLA X
 STO Y
)
```

I'm pretty sure that the following also worked in our version of Midas:[23]

```
MACRO ADDTHEM X,Y,Z
      (ENTAL X
       IRP W,(Z)
      (ADD W
      )
       STRAL Y
      )
```

when called with `ADDTHEM A,B,(C,D,E)` resulted in

```
ENTAL A
ADD C
ADD D
ADD E
STRAL B
```

All in all, this macro effort was a significant piece of computing and programming education for me.[24]

## 5 More study and use of macro processors (and language extension capabilities)

In September 1967, I moved to Bolt Beranek and Newman Inc. in Cambridge, MA, where I had access to the company's PDP-1d time sharing system. I immediately began extensive use of the macro facility built into the editing program TECO.[25]

TECO[22] was an early, very powerful, text editor with a macro capability using the same type of keystrokes one used for editing. One could type a list of keystrokes to do some complex editing function but delay evaluation and instead save the sequence of keystrokes (i.e., defining a macro) and then later give a few keystrokes to execute the saved string of keystrokes (i.e., calling or executing the macro). TECO macros typically looked very cryptic. People also played games with what complicated things they could do with TECO macros, e.g., calculating digits of pi, implementing Lisp, etc.[26]

Also early on at BBN, as a weekend hack, I transcribed the Algol-like listing of Strachey's GPM system from his 1965 paper into PDP-1 Midas assembly language and made it run. This was easy to do given Strachey's complete description of the system.

I also investigated the TRAC language.[27,28] TRAC was presented by Calvin Mooers as a text processing language; but to my mind, TRAC was not so different from a macro processor in the way it defines and manipulates strings.[29]

The basic TRAC operation is a call to a built-in function introduced by a pound sign, e.g.,

```
#(function-name,arg1,arg2,...)
```

Two of the built-in functions are define string (`ds`) and call (`cl`), as follows:

```
#(ds,greeting,Hello World)
```

and

```
#(cl,greeting)
```

resulting in replacement of the call by the string "`Hello World`".

Another built-in TRAC function, `ss`, specifies the strings for which a substitution is to be made at call time. Thus,

```
#(ds,greeting,Hello, name)
#(ss,greeting,name)
```

creates a macro with one call-time argument, such that

```
#(cl,greeting,Hello Dave)
```

results in the text string "Hello, Dave".

There are other built-in functions for string comparison, and so on.

By this time, I was pretty fascinated by programming languages and macros, in particular the idea of extensions to programming languages.

Macros have often been used as a form of language extension. For instance, complex add might be defined, using McIlroy's notation, as

```
COMPLEXADD, A, B, C, D, E, F = FETCH, A
                               ADD, C
                               STORE, E
                               FETCH B
                               ADD D
                               STORE F
```

with the obvious substitution when called with

```
COMPLEXADD, U, V, W, X, Y, Z
```

From macros as a way of extending languages it was a short step to the idea of extensible high-level languages. From 1966–1968 I took computer science courses at MIT as a part-time graduate student and eventually did Master's thesis work (never completed) on a capability for extending a high level language. Unfortunately, I have lost the complete draft of the thesis report (and I never finished the accompanying program). However, I am sure that my thesis literature research and thinking influenced the next project I had at BBN.

In 1967 to 1968, with the assignment to think about a programming language for what became BBN's PROPHET system (a tool to help medicinal chemists and research pharmacologists), I looked deeply into extensible languages. I studied and reported on McIlroy's ideas[6] and the ideas and implementations of several other researchers.[30] Eventually, as a proof of concept, I translated James Bell's Proteus from the Fortran implementation in his thesis into PDP-10 assembly language. I turned this effort over to Fred Webb, who eventually replaced what I had done with a fresh implementation of an extensible language named PARSEC.[31] PARSEC was used in various versions (and later as the basis for RPL for the RS/1 system) by a multitude of people for many years.

This note on macros is not the place to go more deeply into extensible languages. The references in note 30 are a decent introduction to the state of the art circa 1968. For the state of the art a decade later, John Metzner's graded bibliography on macro systems and extensible languages is relevant.[5]

## 6 Midas, macros, and the ARPANET IMP program

At BBN I was part of the small team that in 1969 developed the ARPANET packet switch.[1,32] We developed the packet switch software for a modified Honeywell 516 computer using the PDP-1d Midas assembler, using lots of macros, etc., to adapt Midas to know about the 516's instruction set and paged memory environment. Bernie Cosell primarily constructed this hairy set of conversation macros. Our three person software team (Cosell, Will Crowther, and me) also used Midas macros to facilitate development of the packet switching algorithms to run on the 516. All this may be seen in a listing of the IMP program available on the web.[33] We also used Midas macros to generate a concordance for the IMP program[34] as well as to reduce the probability of writing time-sharing bugs.[35] A contemporary version of this Midas macro assembler written in Perl by James Markevitch is also available for study.[36]

## 7 Ratfor and Infomail

The next project on which I saw something like a macro processor was an effort in the early 1980s to develop a commercial email system that would run on a variety of vendor platforms, e.g., DEC VAX, IBM CICS, IBM 360, and Unix on the BBN C/70. For portability we decided to implement our email system (called InfoMail) in Fortran, for which compilers already existed for the target platforms. However, to avoid having to actually write Fortran code, we developed the system using Ratfor (Rational Fortran).[37] Ratfor was not actually a macro processor but rather a programming language that acted as a preprocessor to emit a Fortran program that could be compiled by a standard Fortran compiler. Nonetheless, that seems to me to be quite a lot like what a macro processor does. Also, Chapter 8 of the *Software Tools* book (see previous note) describes the implementation in Ratfor of a macro processor (based on the macro processor for the programming language C).

Also during my BBN years between 1967 and 1995, I briefly used the C language, which includes a macro processor,[38] which optionally can be used independently of the rest of C. But from 1982 on I did no real computer programming and thus didn't track what was happening in the world of macro processors.

## 8 TeX, macros for typesetting

People who use (LA)TeX macros may already know much of what is in this section. However, some of it may be new to some readers.

After retirement from BBN in 1995, I had started using (LA)TeX in place of a word processor such as Microsoft Word, particularly for documents that were more than a one-page, one-time letter. This brought me in contact with the very sophisticated and complex macro processor embedded in the TeX typesetting system. I have now been using this system and its macro processor for typesetting for nearly 20 years, the longest in my life I have used any single macro processor.

Here is an example of a TeX macro.

```
\def\Greeting#1{Hi #1! I hope you're well.}
```

If this is called with `\Greeting{Dave}`, it results in

```
Hi Dave! I hope you're well.
```

The text "`#1`" tells the macro definition processor (a) that there is one argument, and (b) where the call-time argument is to be substituted in the body of the macro. If the macro definition allowed three arguments, for instance, the sequence "`#1#2#3`" would appear after the macro name and before the open curly bracket of the definition.

The TeX macro processor is enormously powerful and flexible, in its unique way, and a comprehensively documented piece of software.[39,40,41,42] Massive programs have been (and continue to be) written in its macro language. For example, LaTeX is implemented entirely with TeX macros, as are other variants or supersets of TeX (called "formats" in TeX jargon) as well as thousands of LaTeX "packages" which extend or modify the capabilities of LaTeX. One modest-size example of the use of TeX macros to extend LaTeX can be found on pages 6–10 of "The `bibtext` Style Option".[43]

Personally, I tend to do my LaTeX extensions using packages that other people have already written, although sometimes I make minor modifications to existing packages. More commonly I use TeX macros (or a LaTeX variation) to replicate small snippets of LaTeX code which are used repeatedly — for consistency, and also to easily allow later changes of such snippets as I figure out what I actually want them to do. In 2004 I published an example of one such use of a TeX macro;[44] I encourage readers to take a look at it as it describes (far from completely) various ways TeX macros are defined and can be called.

Historically, there has been an interesting set of pressures around TeX's macro capability. Originally Donald Knuth included only enough macro capability to implement his typesetting interface. However, he was persuaded by early users to expand that macro capability. That expanded capability allowed users to construct pretty much any logic they wanted on top of TeX (although often such add-on logic was awkward to code using macro-type string manipulations). On the one hand, TeX and its macro-implemented derivatives have always been very popular and there have been non-stop macro-based additions for over 30 years. On the other hand, users despair at how annoying coding using macros is, moan about "why Knuth couldn't have included a real programming language within TeX", and otherwise cast aspersions on TeX's macro capability.

Over the years various attempts have been made to link TeX to a "real" programming language, typically invoking the programming language from TeX or the reverse; however, none of these efforts have come into widespread use. Over the past few years, however, a small group of TeX hackers have accomplished an apparently successful merger of the Lua programming language and TeX, called LuaTeX.[45] LuaTeX maintains TeX's macro processor (there is really no way, and no reason, to get rid of it while keeping TeX). Thus, the full power of the TeX macro processor is available for the many situations in which it is the best tool, and the Lua language is available for things which can be done much more easily in a procedural language.

The history of the TeX macro processor partially explains the above. Knuth has made the point that he was designing a typesetting system that he didn't want to make too fancy, i.e., by including a high level language. He has also noted that when he was designing TeX he created some primitive typesetting operations and then created a set of macros for the more complete typesetting environment he wanted. He expanded the original macro capability when fellow Stanford professor Terry Winograd wanted to do fancier things with macros. Knuth's idea was that TeX and its macro capability provided a facility with which different people could develop their own typesetting user interfaces, and this has happened to some extent, e.g., LaTeX, ConTeXt, etc.

It is perhaps worth discussing a few of the things that make the TeX macro capability different from, for example, the capability of GPM.

GPM has a simple and unchanging definition and calling syntax, as was described in Section 3. Macro definitions can include other macro definitions, and macros can have recursive calls (and without going back to study the GPM paper carefully, I assume that the scope of macro definitions happens in the natural and obvious way). The definition associated with a macro name can be "looked at" without evaluating the definition; the definition associated with a macro name can be assigned to a different macro name; and there is a capability for converting numbers between binary and decimal formats and for doing binary arithmetic. No limit is specified for the number of arguments a macro definition and call can have. Finally, GPM is a stand-alone program; it processes its input, and it is up to the user what happens next with its output.

The TeX macro capability can do all those things; but it cannot be used independent of TeX, and in its straightforward form its macro definitions and calls are limited to nine arguments. TeX also has a way of defining a macro call to have a much more free-form format, and with some programming (or use of an appropriate TeX macro package) macro call arguments can be specified with attribute-name/value pairs. There are explicit commands in TeX creating local or global definitions, as well as various other definition variations, such as delayed definition and delayed execution of macro calls. TeX has a rich rather than minimal set of conditional and arithmetic capabilities (some related only to position in typesetting a page). There are also ways to pass information between macros and, more generally, to hold things to be used later during long, complicated sequences of evaluation and computation. These capabilities allow big programs to be written in the macro language, and thus TeX also has a capability to trace the flow of macro definition and execution.

TeX has another unusual capability that is sometimes used with macros, although it is a capability more closely related to lexical analysis than to macro definitions and calls; this is the TeX "category code" feature. TeX turns its input sequence of characters into a list of tokens. A token is either a single character with a category code (catcode) or a control sequence. For instance (using an example from Knuth's *The TeXbook*), the input "`{\hskip␣36␣pt}`" is tokenized into

```
{ hskip 3 6 ␣ p t }
```

where the opening brace is given the catcode of 1 (begin group), `hskip` gets no catcode because it is a control sequence, `3` and `6` get catcodes of 12 (other), the space after `36` gets catcode 10 (space), `p` and `t` get catcodes of 11 (letter), and the closing brace gets catcode 2 (end group). (There are 16 such category codes in all.)

The next step in the TeX engine decides what to do with these different tokens, e.g., put the numbers together into a numeric value and the letters together into a unit of measurement, execute the primitive `hskip` command, and so on. The backslash has a catcode of zero indicating an escape character, thus telling TeX that the following letters (five in this case) form a control sequence; the space after the command name delimits the end of the name and doesn't otherwise count as a space.

TeX also has the capability of changing which character(s) have which catcode(s). For instance, the dollar sign (by default having catcode 3, indicating a shift to math mode) could be given the escape character catcode, and the backslash could be given the math shift catcode. This capability is routinely used in TeX program libraries to define macro names internal to a program in the library which users of the library cannot use when (normally) defining their own macro names. Basically, the entire input language of TeX can be changed.

The typeface design system, METAFONT, that Knuth developed in parallel with TeX has a more powerful macro capability (my memory is that he says somewhere that this was because he assumed more sophisticated people would be using METAFONT than TeX).

When Knuth rewrote TeX and METAFONT using literate programming techniques and targeting the Pascal programming language,[46] he included a macro capability in his literate programming WEB system for two reasons: simple numeric and string macro definitions were used to simplify coding given the limitations of Pascal; another kind of macro supported literate programming by allowing source material to be presented in an order suitable for human readers, which was then reordered by the system into the order needed for compiler processing.[47] (When literate programming systems targeting C were later developed, WEB's numeric and string macros were no longer strictly needed since C has its own macro processor, as noted above. The macros supporting literate programming continued to be an essential part of the system.)

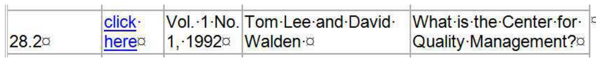
I got to wondering when and where Donald Knuth got his own introduction to macro processors. He said:[48]

> I worked with punched cards until the 70s. My first "macro processor" was a plugboard for the keypunch, setting up things like tab stops! The assembler that I wrote as an undergrad, SuperSoap for the IBM 650, had a primitive way for users to define their own "pseudo-operations"; but it was extremely limited. For example, the parameters basically had to be in fixed-format positions.
>
> I learned about more extensible user-definability with the first so-called "compiler-compilers", notably D. Val Schorre's "META II" (1964)[49] and E. T. Irons's syntax-directed compiler written at Yale about the same time.[50] Later I knew about the sort-of macros in other compilers, e.g., PL/I.
>
> But the first really decent work on what we now call macro expansion was done I think by Peter Brown ... it was his book *Macro Processors* (Wiley, 1974)[2] that was my main source for macros in TeX, as far as I can remember now.

## 9 M4

In about 2001, I was looking for a macro processor to help me format an HTML list of the articles in the *Journal of the Center for Quality of Management* that were relevant to the chapters of a book I co-authored.[51] I found the m4 macro processor.[52]

The following first entry in an HTML table:[53]



| | | | | |
|---|---|---|---|---|
| 28.2¤ | click·here¤ | Vol.·1·No. 1,·1992¤ | Tom·Lee·and·David·Walden·¤ | What·is·the·Center·for·Quality·Management?¤ |

was created with the following m4 macro definition that outputs HTML markup:

```
define(‘_te’,‘
<TR VALIGN="top">
<TD ALIGN="center"><FONT FACE="Arial">$1</FONT></TD>
<TD ALIGN="left"><FONT FACE="Arial">$2</FONT></TD>
<TD ALIGN="left"><FONT FACE="Arial">$3</FONT></TD>
<TD ALIGN="left"><FONT FACE="Arial">$4</FONT></TD>
<TD ALIGN="left"><FONT FACE="Arial">$5</FONT></TD>
</TR>
’)
```

which was called as follows (except all on one line):

```
    _te(28.2,_url(00100),_i(1,1,1992),
        Tom Lee and David Walden,What...)
```

where "`_url`" and "`_i`" are other m4 macro calls. I gave the m4 macros names beginning with underscores so the names would be recognizable in the m4 source file which didn't otherwise use underscores.

Use of m4 brought me full circle, in a sense, as the m4 macro processor is somewhat derived from GPM.[54] It was also used for and with Ratfor.

One incarnation of m4 is a freely available program under the GNU GPL, and thus that implementation is also available for study and other use.[55]

My writing and publishing work using a LaTeX engine involved me in publishing work flows, particularly the need to have one source file be processable into multiple publication formats. For example, in the case of the TeX Users Group Interview Corner,[56] we use m4 macros to define a new markup language which can, with different sets of definitions, be targeted to output markup language code for either LaTeX for paper printing or HTML for web posting. In our interview sources files, we might write the macro call "`_i(Book Name)`". With our HTML set of m4 definitions, that would convert to "`<i>Book Name</i>`". With our LaTeX set of m4 definitions, that would convert to "`\textit{BookName}`". We have hundreds of m4 definitions in each of the LaTeX and HTML m4 definition files that implement this ad hoc markup language that targets the two methods of output.

A language independent macro process such as GPM or m4 can be used at any point in a workflow:

- as a preprocessor for a programming language (as in many of the examples in this note)
- as a post processor (Martin Richards has noted that a GPM-like macro processor was used to

convert the O-code output of his BCPL compiler into machine language)[57]

- as an intermediate step in the work flow (for instance, a Perl program processes an input driver file to generate `m4` macro calls, the macro calls generate LaTeX markup, and the LaTeX engine generates PDF files from the LaTeX markup).

## 10 Reflections

**General reflections.** A computing historian who is a potential reader of this note might ask, "Why should I care about macro processors?" There are a couple of answers to this. First, macro processors played an important role in the history of programming languages. They were used with early assemblers, before higher level languages became widespread. In fact, they allowed assembly language users to create higher level language constructs by grouping together useful sequences of assembly language instructions under a single name, for example, `complex-add`. As higher level languages came into use (e.g., Fortran, ALGOL), the usefulness of macro processors was carried over into higher level languages (e.g., to create yet higher level linguistic constructs). Soon stand-alone macro processors were created that could act as a preprocessor for any programming language rather than being embedded within a particular programming language. Second, both embedded and stand-alone macro processors continue in widespread use today. We also find macro processors embedded in the user interface of many computer tools other than programming languages, for instance, as part of text editors, operating system shells, makefiles, and other applications and software packages (one example is Actions in Photoshop — repeatable sequences of Photoshop commands).

The reader might next ask, "If macro processors have been important in the history of programming, why isn't more written about them?" I think the answer to that question may have to do with the relative simplicity of many macro processors, which substitute one character string for another. That is not an area that requires much research, and generally useful implementation methods have been well known since the 1960s or earlier; also, it is not too hard to do an ad hoc implementation of a macro processor as part of some piece of software one is writing. Being mostly implemented as string manipulators, macro processors don't have to have all of the debatable research topics that a full programming language has (side effects, call by name, scope, etc. — although the sophisticated macro processors such as `m4` and TeX have parallels for these sorts of programming-language-philosophy topics). Also, as macro processors tend to operate at compile time, sometimes in ways that increase runtime efficiency, there probably hasn't been as much interest in optimizing macro processor performance as there has been with programming languages.

From the point of view of the professional programmer (versus historian), macro processors are just another development tool, and they are available as preprocessors or embedded in whatever programming language or other development tools the programmer is using. Details about how a macro processor works may annoy but probably won't deter such working programmers. They just use what is available as best they can, and don't write the theoretical articles. Practical use of macros also tends to be awkward, and describing their use can be messy. Thus, it is difficult in a short paper to describe serious uses of macros.

On the other hand, people who might be reluctant to use a "real" programming language may use macros (e.g., editor macros) in quite sophisticated ways. In the multi-chapter interview of Knuth in his *Companion to the Selected Papers of Donald Knuth* (page 161),[58] Knuth says of his wife, "I don't think she will ever enjoy programming. She is good at creating macros for a text editor, sometimes impressing me with subtle tricks that I didn't think of, but macro writing is quite different from creation of what we call 'recursive procedures' or even 'while loops.' "

Macro processors are also not as tractable a topic for historical or theoretical writing as high level languages perhaps are. While many macro processors may be for relatively straightforwardly extending a programming language or aiding in a development task, some macro processors (such as `m4` and the macro capability in TeX) have a big set of capabilities for dealing with sophisticated programming situations — they are set up to write big complicated programs, just as regular higher level languages are, but often in more clumsy-to-use ways. There is a 160-page manual to describe all the capabilities of the stand-alone `m4` macro processor. In addition to Knuth's own writings,[39,41] Eijkhout's book[40] has more than 50 pages on the extensive macro capabilities embedded in TeX, and Stephan von Bechtolsheim wrote (perhaps excessively) a 650-page book[59] on using TeX macros. (There is probably an interesting paper to be written comparing the issues inherent in these powerful macro processors with similar issues in regular procedural programming languages.)

The immediately preceding discussion brings to mind the question, "Where on the spectrum of programming languages do we put macro processors?"

Early on they were used as ways to extend assemblers and compilers. Quickly they were used to implement programming languages (e.g., WISP and SNOBOL). They were used to simulate different "computers" (e.g., BCPL's compiler of O-code, the assembly language of the Honeywell 516 IMP computer). I personally have used them to define and process little special purpose markup languages (as described in the m4 and TeX sections above).

Many macro processors are Turing-complete and in this sense can compute anything a typical higher level language can compute. However, they are distinct from many higher level language compilers (discounting macro capabilities they might have, or other compile-time evaluation capabilities, as languages which execute interpretively often have, e.g., Lisp, Perl) in that much computation in macro processors inherently goes on at "compile" time. This can make their syntax and semantics harder to specify.

In Knuth's paper "The Genesis of Attribute Grammars", reprinted in his *Selected Papers on Computer Languages*,[60] Knuth explains (pages 434–435) that he didn't use an attribute grammar to specify the semantics of TeX because he hadn't been able to think of *any* good way to define TeX precisely except through the implementation using Pascal. He also couldn't think of how "to define any other language for general-purpose typesetting that would have an easily defined semantics, without making it a lot less useful than TeX. The same remarks also apply to METAFONT. 'Macros' are the main difficulty. As far as I know, macros are indispensable but inherently difficult to define. And when we also add a capability to change the interpretation of input characters, dynamically, we get into a realm for which clean formalisms seem impossible."

**Personal reflections.** Macro overview books, such as those by Brown and Campbell-Kelly, try to categorize the uses of macro processors, for example, to extend a programming language, to allow late binding of variables to values, to communicate with the operating system, to implement desirable-but-non-existent machine instructions, to insert debugging code into a program, and to simply abbreviate long strings of text. Below is a short list of the ways I feel I have made use of the macro processors described in this note.

- The macro processor described in McIlroy's paper, Strachey's GPM, and Mooers' TRAC were for the study of programming languages, understanding of how macro processors work, and for coding practice.
- Midas was about implementing a macro processor and using one for production work including retargeting Midas to a new computer and putting hints in a real-time program to reduce the probability of time-sharing bugs.
- TECO and other editor macros were to reduce editing keystrokes, especially when regular expression searches and replacements are needed.
- Ratfor and C were for programming efficiency.
- TeX macros are for production typesetting, creating extensions to LaTeX (e.g., the style for a particular book), and as another sophisticated macro processor to study.
- m4 use has been to create new markup languages with which to target document source files to different output media.

In addition to assembly languages for different computers, Fortran for various mathematical projects, and the languages mentioned above, I briefly programmed in Lisp and BCPL years ago. Lisp has a macro capability and I may have used it, but, if so, I don't remember anything about it.[61] While writing these notes, I have learned that a version of Strachey's GPM existed for BCPL, known as BGPM.

In recent years I have used Perl for several large and many small projects. Apparently Perl has a capability for redefining parts of the Perl language (the equivalent of macros and more), but it confused me when I tried to read about it and am not likely to try it. I suppose I can just use m4, which I already know, as a preprocessor for Perl; but that doesn't stop me from wishing that a conventional macro capability was built into Perl.

## Acknowledgments

Chuck Niessen helped me remember the capabilities of the Univac 1219 macro assembler we wrote. Ralph Alter and Will Crowther responded to queries relating to macro capabilities on Lincoln Laboratory computers. Stan Mazor confirmed my memory about whether or not we used macros in the early 1960s on the IBM 1620 at San Francisco State College. Tom Van Vleck reminded me of what languages with macro processors were available at MIT in the late CTSS and early Multics era. Jim Wood reminded me of some things about PARSEC. Bill Aspray and Martin Campbell-Kelly provided guidance on how this paper might be developed into a more traditional academic computing history paper. Bernie Cosell pointed out half a dozen subtle issues or possibilities in the paper. Ralph Muha gave me his copy of Gimpel's SNOBOL4 algorithms book. Most helpfully, Karl Berry read drafts of this note, noted typos, made suggestions for improvement, and provided additional insight about and references for a number

of macro processors, particularly TEX. Karl, along with Barbara Beeton, also edited the final draft of the paper to ready it for publication.

## Notes

[1] `http://walden-family.com/impcode/imp-code.pdf`

[2] [*Macro overview*] P. J. Brown, *Macro Processors and Techniques For Portable Software*, John Wiley & Sons, 1974.

[3] [*Macro overview*] Martin Campbell-Kelly, *An Introduction to Macros* (Computer monographs, 21), MacDonald & Co., London, 1974.

[4] [*Macro overview*] Peter Wegner, *Programming Languages, Information Structures, and Machine Organization*, McGraw-Hill Book Company, 1968, section 2.6 and section 3.1–3.4, pp. 130–180.

[5] [*Macro overview*] John R. Metzner, A graded bibliography on macro systems and extensible languages, *ACM SIGPLAN Notices*, Volume 14 Issue 1, January 1979, pp. 57–64.

[6] [*McIlroy's model*] M.D. McIlroy, Macroinstruction Extensions for Compiler Languages, *Communications of the ACM*, vol. 3 no. 4, 1960, pp. 214–220.

[7] In 1967 McIlroy's paper was reprinted in the now classic book *Programming Systems and Languages*, edited by Saul Rosen, McGraw-Hill, New York, 1967. For this paper I was looking at the reprint of Rosen's book. In the reprint McIlroy doesn't say anything about the macro name in this example also being used within the definition but not apparently as a recursive call of the macro, i.e., the 3-argument call can be distinguished from the 1-argument instruction.

[8] [*Macros at Bell Labs*] May 4, 2009, `http://deememorial.blogspot.com/2009/05/doug-mcilroy-recalls-bell-labs.html`

[9] [*Other macro systems*] M. Barnett, Macro-directive Approach to High Speed Computing, Solid State Physics Research Group, MIT, Cambridge, MA, 1959; D.E. Eastwood and M.D. McIlroy, Macro Compiler Modifications of SAP, Bell Telephone Laboratories Computation Center, 1959; I.D. Greenwald, Handling of Macro Instructions, *Communications of the ACM*, vol. 2 no. 11, 1959, pp. 21–22; M. Haigh, Users Specification of MICA, SHARE User's Organization for IBM 709 Electronic Data Processing Machine, SHARE Secretary Distribution SSD-61, C-1462, 1959, pp. 16-63; A.J. Perlis, Official Notice on ALGOL Language, *Communications of the ACM*, vol. 1 no. 12, 1958, pp. 8–33; A.J. Perlis, Quarterly Report of the Computation Center, Carnegie Institute of Technology, October. 1969; Remington-Rand Univac Division, Univac Generalized Programming, Philadelphia, 1957.

[10] [*Other macro systems*] This article was originally published in Volume 7 of the *Encyclopedia and Computer Science and Technology*, published by Michel Dekker in 1977. The article is reprinted as chapter 1 of Donald E. Knuth, *Selected Papers on Computer Languages*, Center for the Study of Language and Information, Stanford University, 2003.

[11] Page 6 in the volume of Knuth's selected papers.

[12] Page 42 of the selected papers volume.

[13] Martin Campbell-Kelly told me [email of 2013-11-23], "Maurice Wilkes developed a macro-based system called WISP for list processing (`http://ai.eecs.umich.edu/people/conway/CSE/M.V.Wilkes/M.V.Wilkes-Tech.Memo.63.5.pdf`) ... Peter Brown[2] was Wilkes' PhD student."

[14] [*GPM*] C. Strachey, A General Purpose Macrogenerator, *The Computer Journal*, vol. 8 no. 3, 1965, pp. 225–241.

[15] See Section 1.2, Historical References, at `http://www.gnu.org/software/m4/manual/`

[16] [*GPM*] Another implementation of GPM can be found in Section 8.8 of James F. Gimpel's book *Algorithms in SNOBOL4*, John Wiley & Sons, 1976. (SNOBOL4 itself was implemented using a collection of macros which create a virtual machine for the purpose of language portability: Ralph E. Griswold, *The Macro Implementation of SNOBOL4*, W.H. Freeman and Company, San Francisco, 1972. SNOBOL4 is sometimes known as "Macro SNOBOL".)

[17] Email of November 16, 2013.

[18] [*Midas*] MACRO Assembly Program for Programmed Data Processor-1 (PDP-1), Digital Equipment Corporation, Maynard, MA, 1963, `http://bitsavers.informatik.uni-stuttgart.de/pdf/dec/pdp1/PDP-1_Macro.pdf`

[19] Email of October 28, 2013.

[20] [*Midas*] `http://ia601609.us.archive.org/9/items/bitsavers_mitrlepdp1_1535627/PDP-1_MIDAS.pdf`

[21] [*Midas*] `http://ia601601.us.archive.org/11/items/bitsavers_mittx0memo_2363951/M-5001-39_MIDAS_Nov62.pdf`

[22] Ralph Alter wrote a version of TECO, Dan Murphy's paper Tape Editing and Correcting Program: Dan Murphy, The Beginnings of TECO, *IEEE Annals of the History of Computing*, 31(4), October–December 2009, pp. 225–241. Will Crowther wrote a version of Alan Kotok's DDT debugging program. Chuck Niessen and I wrote a version of Robert Saunder's macro assembler: Chuck wrote the basic assembler and I wrote the macro-processor.

[23] I have the line printer assembly listing and my handwritten flow chart for this macro processor. It is tempting to try to make this macro processor run again on a 1219 emulator.

[24] This was the first program in which I used stacks and for which I thought about recursion in a profound way. I think this was also the first time I wrote code for managing a symbol table.

[25] I don't remember using this capability in TECO at Lincoln Lab.

[26] I won't bother with further mention of macros in other editors (or macros in various job control languages, shells, or makefiles) in the rest of this note.

[27] [*TRAC*] Calvin Mooers and Peter Deutsch, TRAC: A Text Handling Language, *Proceedings of the 20th ACM National Conference*, 1965, pp. 229-246.

[28] As part of my investigation, I visited TRAC creators Calvin Mooers at his Cambridge office (of the Rockford Research Institute) and Peter Deutsch at the Cambridge home of his parents.

[29] Brown's book[2] in fact classifies TRAC as a macro processor.

[30] [*Extensible languages*] Niklaus Wirth, On Certain Basic Concepts of Programming Languages, Technical Report No. CS 65, Computer Science Department, Stanford University, May 1, 1967; B.A. Galler and A.J. Perlis, A Proposal for Definitions in ALGOL, *Communications of the ACM*, vol. 10 no. 4, April 1967, pp. 204–219; T.E. Cheatham, Jr., A. Fischer, and P. Jorrand, On the Basis for ELF — An extensible language facility, AFIPS Fall Joint Computer Conference, 1968, pp. 937–948; J.V. Garwick, J.R. Bell, and L.D. Krider, The GPL Language, Programming Technology Report TER-05, Control Data Corporation, Palo Alto, CA; Thomas A. Standish, A Data Definition Facility for Programming Languages, PhD thesis, Carnegie Institute of Technology, 1967; James Richard Bell, The Design of a Minimal Expandable Programming Language, PhD thesis, Stanford University, 1968.

[31] [*Extensible languages*] F. Webb, The PROPHET System: An Overview of the PARSEC Implementation, BBN Report 2319, September 1, 1972; PARSEC User's Manual, Bolt Beranek & Newman, Cambridge, MA, December 1972.

[32] F.E. Heart et al., The Interface Message Processor for the ARPA Computer Network, AFIPS Conference Proceedings 36, June 1970, pp. 551–567.

[33] `http://walden-family.com/impcode/c-listing-ps.txt`

[34] `http://walden-family.com/impcode/d-concordance.pdf`

[35] `http://walden-family.com/impcode/detect-interrupt-bugs.pdf`

[36] [*Midas*] `http://walden-family.com/impcode/midas516.txt`

[37] [*Ratfor*] Kernighan, B. and Plauger, P., *Software Tools*, Addison-Wesley, 1976.

[38] [*C preprocessor*] `http://gcc.gnu.org/onlinedocs/cpp/`

[39] [*TEX*] Donald Knuth, *The TEXbook*, Addison-Wesley, 1986, particularly chapter 20.

[40] [*TEX*] Victor Eijkhout, *TEX by Topic*, Addison-Wesley, 1991, particularly chapters 11–14, `http://mirror.ctan.org/info/texbytopic`

[41] [*TEX*] Donald Knuth, *Computers & Typesetting, Volume B, TEX: The Program*, Addison-Wesley, 1986.

[42] [*TEX*] Donald E. Knuth, *Digital Typography*, Center for the Study of Language and Information, Stanford University, 1999.

[43] [*TEX*] `http://mirror.ctan.org/macros/latex209/contrib/biblist/biblist.pdf`

[44] [*TEX*] `http://tug.org/TUGboat/tb25-2/tb81walden.pdf`

[45] [*TEX*] `http://www.luatex.org/`

[46] [*TEX*] Donald E. Knuth, Literate Programming, `http://literateprogramming.com/knuthweb.pdf`

[47] Email of December 5, 2013, from Karl Berry.

[48] Private communication, January 11, 2014; the footnotes in the quotation are from the author of the present paper, not from Knuth.

[49] D. V. Schorre, META-II: A Syntax-oriented Compiler Writing Language, D. V. Schorre, *Proceedings of the ACM, 19th ACM National Conference*, ACM, New York, 1964, `http://ibm-1401.info/Meta-II-schorre.pdf`

[50] Edgar T. Irons, A syntax-directed compiler for ALGOL 60, *Communications of the ACM*, vol. 4, 1961, pp. 51–55; Edgar T. Irons, The structure and use of a syntax-directed compiler, *Annual Review of Automatic Programming 3*, 1962, pp. 207–227.

[51] `http://walden-family.com/4prim/`

[52] [*M4*] `http://www.gnu.org/software/m4/manual/`

[53] The full table is at `http://walden-family.com/4prim/archive/issues-list.htm`

[54] [*M4*] `http://en.wikipedia.org/wiki/M4_(computer_language)`

[55] While `m4` was originally developed by Brian Kernighan and Dennis Ritchie in 1977 and released as part of AT&T Unix, the GNU version mentioned here was a complete rewrite by René Seindal with continuing updates by many others. The GNU `m4` manual's history section[15] has a good bit of additional history.

[56] `http://tug.org/interviews/`

[57] Martin Richards, Christopher Strachey and the Cambridge CPL Compiler, *Higher-Order and Symbolic Computation* (a special Christopher Strachey memorial issue), 13, 2000, pp. 85–88.

[58] Donald E. Knuth, *Companion to the Selected Papers of Donald Knuth*, Center for the Study of Language and Information, Stanford University, 2012.

[59] [*TEX*] Stephan von Bechtolsheim, *TEX in Practice, Volume III: Tokens, Macros*, Springer-Verlag, 1993.

[60] Donald E. Knuth, *Selected Papers on Computer Languages*, Center for the Study of Language and Information, Stanford University, 2003.

[61] Tim Hart at MIT added a macro capability to Lisp in 1963. Macros in Warren Teitelman's BBN Lisp, which I used, were perhaps more well developed. In some sense the original Lisp interpreter functioned a bit like a macro processor.

◇ David Walden
`http://walden-family.com`

## *Eutypon* 30–31, October 2013

*Eutypon* is the journal of the Greek TEX Friends (`http://www.eutypon.gr`).

Yiannis Mamaïs, Books made with love and pride; pp. 1–9

The author is one of the few former craftsmen printers of Greece who did not quit the book world after the abrupt introduction of photocomposition in the early 1980s. On the contrary, he continued to work as an editor, trying to teach typographic æsthetics to younger generations. He has more than 1,600 books to his credit, most of which came from the publisher Gutenberg. This article is a confessional outpouring by him, with comments on the past and current situation in Greek typography. (*Article in Greek with English abstract.*)

Philip Taylor, Cataloguing the Greek manuscripts of the Lambeth Palace Library: An exercise in transforming Excel into PDF via XML using (Plain) XeTEX; pp. 11–28

Work is in progress to prepare an analytical catalogue of the Greek manuscripts held in the Lambeth Palace Library. The catalogue will be published online in downloadable PDF format and (at a later stage) in print, using a single set of source documents marked up in the extensible markup language XML. This paper discusses the various stages through which the documents pass, starting as Excel spreadsheets and ending up both in Adobe Portable Document Format (PDF) and as Text Encoding Initiative (TEI)-compliant XML. (*Article in English.*)

Apostolos Syropoulos, A short introduction to MathML; pp. 29–49

MathML is now the *de facto* standard for the presentation of mathematical formulas on the Internet. Indeed, with the introduction of HTML5, which integrates MathML while replacing HTML4 and XHTML, it becomes obvious that some knowledge of MathML is almost essential for anyone interested in presenting mathematical content online. (*Article in Greek with English abstract.*)

Ioannis Dimakos, A brief introduction to ShareLATEX; pp. 51–58

A brief introduction to `sharelatex.com`, a site offering online editing and compilation of LATEX files. ShareLATEX allows users to access and process their LATEX files via their browser without the necessity of having a local LATEX installation. It also allows the online sharing and cooperation between users of the same file(s). (*Article in Greek with English abstract.*)

Dimitrios Filippou, TEXniques: Hanging characters; pp. 59–62

As explained in this short note, character protrusion — particularly protrusion of accents in front of Greek capital letters — can be done *via* `\llap`, `microtype` or with the use of appropriately designed fonts. (*Article in Greek.*)

Dimitrios Filippou, Book presentations; pp. 63–64

A short appraisal of two books: (i) M.R.C. van Dongen, *LATEX and Friends*, Springer, Berlin Heidelberg 2012; and (ii) Victor Eijkhout, *The Computer Science of TEX and LATEX*, Lulu (Victor Eijkhout) 2012. (*Article in Greek.*)

[Received from Dimitrios Filippou and Apostolos Syropoulos.]

## *Die TEXnische Komödie* 1/2014

*Die TEXnische Komödie* is the journal of DANTE e.V., the German-language TEX user group (`http://www.dante.de`). [Non-technical items are omitted.]

Dominik Wassenhofen, Explizite Positionierung in LATEX [Explicit positioning in LATEX]; pp. 17–19

Absolute text positioning with the `textpos` and `grid-system` packages.

Uwe Ziegenhagen, Schneller »TEXen« mit Tastenkürzeln [Faster “TEXing” using shortcut expanders]; pp. 20–25

“Shortcut Expanders”, sometimes also called “Text Expanders”, are software tools that allow the user to define keyboard shortcuts for often-used pieces of text or source code.

Thomas Hilarius Meyer, Honigetiketten für die Schulimkerei [Creating honey jar labels for a school’s beekeeping]; pp. 26–30

This article shows how LATEX can be used to create bilingual labels.

Heiko Oberdiek, Durchstreichen einer Tabellenzelle [Striking through table cells]; pp. 31–32

This article shows an example that uses the `zref` and `savepos` packages to strike through table cells and to keep the position of the cell.

Herbert Voss, Rechenoperationen mit LuaTEX [Calculations with LuaTEX]; p. 33

Usually LATEX is not the tool of choice when it comes to doing more complex calculations. With LuaTEX this has fundamentally changed.

Herbert Voss, LuaTEX und `pgfplots` [LuaTEX and `pgfplots`]; pp. 34–35

LATEX’s computing functions cannot dramatically improve with the upcoming `l3fp` package from LATEX3. With LuaTEX the whole computing effort can be “outsourced”, with LATEX only being used to display the internal or external data.

[Received from Herbert Voß.]

# The Treasure Chest

This is a list of selected new packages posted to CTAN (`http://ctan.org`) from December 2013 through March 2014, with descriptions based on the announcements and edited for extreme brevity.

Entries are listed alphabetically within CTAN directories. A few entries which the editors subjectively believe to be of especially wide interest or otherwise notable are starred; of course, this is not intended to slight the other contributions.

We hope this column and its companions will help to make CTAN a more accessible resource to the TeX community. Comments are welcome, as always.

◇ Karl Berry
`http://tug.org/ctan.html`

## fonts

`lobster2` in `fonts`
  Family of script fonts.

* `zlmtt` in `fonts`
  Latin Modern Typewriter with all features.

## graphics

`aobs-tikz` in `graphics/pgf/contrib`
  Overlay picture elements in Beamer.

`pgf-pie` in `graphics/pgf/contrib`
  Draw pie charts with Ti*k*Z.

`pgf-umlcd` in `graphics/pgf/contrib`
  Draw UML class diagrams with Ti*k*Z.

`pst-intersection` in `graphics/pstricks/contrib`
  Compute intersections between PostScript paths or Bézier curves using the Bézier clipping algorithm.

`pst-ovl` in `graphics/pstricks/contrib`
  Overlay macros for PSTricks.

`repere` in `graphics/metapost/contrib/macros`
  MetaPost macros for drawing grids, vectors, functions, statistical diagrams, and more.

`rulercompass` in `graphics/pgf/contrib`
  Draw straight-edge and compass diagrams in Ti*k*Z.

`tikz-opm` in `graphics/pgf/contrib`
  Draw Object–Process Methodology (OPM) diagrams.

## indexing

`zhmakeindex` in `macros/latex/contrib`
  Enhanced `makeindex` for Unicode and Chinese sorting.

## info

`Practical_LaTeX` in `info/examples`
  Example files for the book *Practical LaTeX*.

`fonts/lobster2`

## macros/latex/contrib

`cnbwp` in `macros/latex/contrib`
  Format working papers of the Czech National Bank.

`cv4tw` in `macros/latex/contrib`
  CV class supporting assets, social networks, etc.

`dccpaper` in `macros/latex/contrib`
  Typeset papers for the *International Journal of Digital Curation*.

`graphicxbox` in `macros/latex/contrib`
  Use a graphic as a box background.

`koma-script-obsolete` in `macros/latex/contrib`
  Obsolete and deprecated packages that are no longer part of KOMA-Script.

`perfectcut` in `macros/latex/contrib`
  Brackets with size determined by nesting.

* `pkgloader` in `macros/latex/contrib`
  Address package conflicts in a general way. (See article in this issue.)

`refenums` in `macros/latex/contrib`
  Referenceable enumerated items.

`rubik` in `macros/latex/contrib`
  Typeset Rubik cube notation via Ti*k*Z.

`scanpages` in `macros/latex/contrib`
  Import and embellish scanned documents.

`sr-vorl` in `macros/latex/contrib`
  Template for books at Springer Gabler, Vieweg, and Springer Research.

`tabstackengine` in `macros/latex/contrib`
  Allowing tabbed stacking with `stackengine`.

`vgrid` in `macros/latex/contrib`
  Overlays a vertical grid on the page.

## macros/latex/contrib/beamer-contrib

`themes/detlev-cm` in `m/l/c/beamer-contrib`
  Originally for the University of Leeds.

## macros/latex/contrib/biblatex-contrib

`biblatex-manuscript-philology` in `m/l/c/b-c`
  Manage classical manuscripts with BibLaTeX.

## support

`convbkmk` in `support`
  Correct (u)pLaTeX PDF bookmarks.

`copac-clean` in `support`
  Snobol program to edit Copac/BibTeX records.

`splint` in `support`
  Write LALR(1) parsers in TeX using `bison` and `flex`. (See article in this issue.)

`texfot` in `support`
  Attempt to reduce online output from (LA)TeX to interesting messages.

## systems

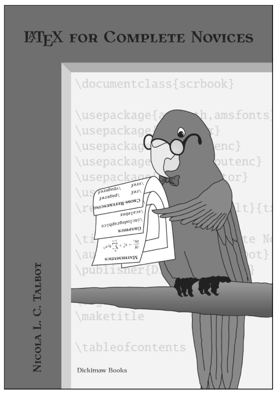
`yandy` in `systems/win32`
  Y&Y TeX, released under the GNU GPL.

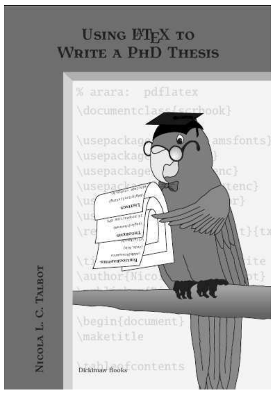
## Book reviews: *LaTeX for complete novices* and *Using LaTeX to Write a PhD Thesis*, by Nicola Talbot

Boris Veytsman

Nicola L. C. Talbot, *LaTeX for complete novices*. Dickimaw Books, 2012, x+279 pp. Paperback, GB£12.99. ISBN 978-1-909440-00-5.

Nicola L. C. Talbot, *Using LaTeX to write a PhD thesis*. Dickimaw Books, 2013, x+148 pp. Paperback, GB£9.99. ISBN 978-1-909440-02-9.





These two books by Nicola Talbot are actually volumes 1 and 2 in her *Dickimaw LaTeX Series*; however, the other volumes are being revised at the time of writing, so this review deals only with these two.

*Dickimaw Books* is an independent publishing house created and owned by Nicola Talbot herself; besides this LaTeX series it has published crime fiction and children's books (including a fairy tale based on Vladimir Propp's structural theory). Both LaTeX books are available as paperbacks (with a discount for TUG and UK-TUG members), or can be downloaded for free from the Dickimaw Books web site (`http://www.dickimaw-books.com`), licensed under the GNU FDL. The electronic versions provide helpful internal and external hyperlinks.

As its title suggests, the first volume is intended for beginners. There are many books for new LaTeX users today. The number of free (as in free speech) ones is smaller: I can recall only Oetiker's great "The not so short introduction to LaTeX $2_\varepsilon$", now included in the major TeX distributions as the *lshort* package. What distinguishes Talbot's book is its firm grasp of modern TeX customs including the modes of communication between TeX users. The book sends the reader to the relevant sections of the UK FAQ for detailed information on the topics discussed, it explains how to ask questions on Stack Exchange and LaTeX community forums. Another sign of modernity is that the book effectively drops the venerable DVI output format; the author merely mentions it in a short paragraph basically saying, "people do not do this anymore" and then discusses only the PDF output. I think this is the right decision, at least for "complete novices". On the other hand, the book includes some topics traditionally considered out of scope for beginners, such as the use of fonts other than Computer Modern. Modern TeX installations have a large number of high quality fonts with very good transparent TeX support, so this topic may now be considered appropriate for novices. On the other hand, the author wisely does not discuss the further possibilities in this area provided by XeTeX or LuaTeX.

The tradition of active use of TeX has resulted in the fact that almost any book for beginners teaches how to define and redefine commands and environments; any TeX user is seen as an (at least budding) TeX programmer. Talbot's book is not an exception: it has two good sections on TeX programming with the proper warning about the danger of redefining macros without understanding them.

The author does not try to cover the multitude of TeX-aware editors and integrated environments. Instead she introduces TeXworks as the preferred work environment and *latexmk* as the compilation wrapper. Again, this seems to be a very good decision: too many choices are good for an experienced user, but can be intimidating for a novice.

Another decision is slightly unusual. While most books for beginners start with the standard classes (*article*, *book*, *report*, *letter* — it seems nobody uses *slides*, however), this book is based on the KOMA-script bundle. I agree that the base classes are now rather long in the tooth, so it makes sense to teach novices some newer and arguably typographically better stuff. Thus the author's experiment seems to be a very promising one. On the other hand, unlike base LaTeX, KOMA is not frozen, which may require updates of the book after changes in the package.

The book is well written and has many carefully chosen exercises. It is a very good candidate for a first LaTeX book.

Of course, this does not mean the book has no flaws. I am not comfortable with the fact that the user's first TeX document appears only on page 32; the previous chapters contain definitions of terms and installation instructions. Unfortunately students today have become used to instant gratification, so some of them may just stop reading much earlier than this if not provided with a hands-on experience. I would suggest starting with the "Hello, world" document and put the definitions and installation instructions later — or in an appendix.

Also, independent publishing means the absence of a professional editor, who might question or correct an author's (rare) mistakes. For example, TEX glue is not called such because in the days of manual typesetting the blocks were glued in place; rather, it is a metaphor invented by DEK. (I doubt it is practical to glue reusable type, and anyway, TEX glue is *between* the boxes rather than *under* them.) The author's advice to substitute *displaymath* for `\[...\]` is strange: *displaymath* is defined by the kernel basically as `\[...\]`. On the other hand, the advice not to use the ugly *eqnarray* is right on the point. Mistakes are evidently rare and don't touch on essential points, so the book can be recommended for studying LATEX.

The second volume is intended for more advanced users. It covers advanced formatting (listings, verbatim environments), splitting the document into separate files, theorem-like environments, creation of bibliography, indices and glossaries, etc. Besides *latexmk*, described in the first volume, this book introduces *arara* as an alternative compilation wrapper. In addition to LATEX and typographic information, it contains some general advice for graduate students, which I find very good and well thought out. For example, the author notes that a skeleton thesis with the chapters in place is a great help against writer's block, giving one a sense of achievement early in the process. The book provides sound advice on many other topics, from the way to format listings of computer code to the proper style of reacting to your advisor's comments. Here is the discussion of the well-known requirement to double space the thesis:

> Many universities still require double-spaced, single-sided documents with wide margins. Double-spacing is by and large looked down on in the world of typesetting, but this requirement for a PhD thesis has nothing to do with æsthetics or readability. In England the purpose of the PhD viva is to defend your work (I gather this is not the case in some other countries, where the viva is more informal, and the decision to pass or fail you has already been made before your viva.). Before your viva, paper copies of your thesis are sent to your examiners. The double spacing and wide margins provide the examiners room to write the comments and criticisms they wish to raise during the viva, as well as any typographical corrections. Whilst they could write these comments on a separate piece of paper, cross-referencing the page in the thesis, it is more efficient for the comments to actually be on the relevant page of the thesis. That way, as they go through the manuscript during your viva, they can easily see the comments, questions or criticisms they wish to raise alongside the corresponding text. If you present them with a single-spaced document with narrow margins, you are effectively telling them that you don't want them to criticise your work!

To tell the truth, I miss the exercises that were an important part of the first volume. On the other hand, one may argue that a stressed graduate student may not find the time to do the exercises.

A rather unfortunate omission in this book is a guide of the thesis classes available on CTAN. Graduate school administrations are (in)famous for strict (and often insensible) requirements for the way a thesis should be typeset, so KOMA classes might not do the job. The author mentions that some universities provide LATEX packages and templates for their students. Such packages, of course, should be used if available. However, other universities do not publish "official" LATEX styles. Nevertheless in many cases one can find a CTAN package that gives a passable alternative. Thus an annotated list of such packages would help a struggling student (cf. "A university thesis class" by Peter Flynn, *TUGboat* 33:2.)

Both books are nicely typeset with Antykwa Toruńska as the main serif font and Libris ADF as the sans serif one. This is again a pleasant diversion from the too-uniform look and feel of many LATEX books. I am not too happy with the way the footnotes are numbered, with the section number used as a prefix (e.g.$^{4.5}$). Unlike equations, footnotes are not usually referenced from other parts of the book, so this prefix looks a bit mannered. However, this is just about my only issue with the typesetting. The books have functional margins with the notes and signs; the text is pleasant to read. There are detailed bibliographies and well constructed indices.

These books are a good addition to any TEX user's library; I wonder whether the author would be willing to put them on CTAN and to include them in the TEX distributions.

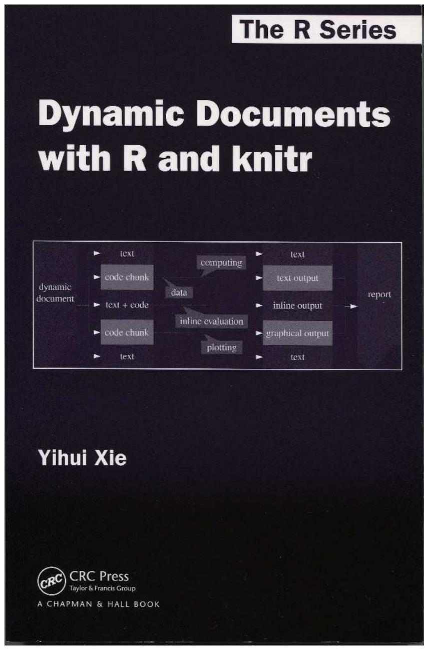
It will be interesting to see the next volumes in the series published. If they are written as well as these two, the world of TEX textbooks and reference books will have more welcome additions.

◇ Boris Veytsman
Systems Biology School and
Computational Materials
Science Center, MS 6A2,
George Mason University,
Fairfax, VA 22030
`borisv (at) lk dot net`
`http://borisv.lk.net`

## Book review: *Dynamic Documents with R and knitr*, by Yihui Xie

Boris Veytsman

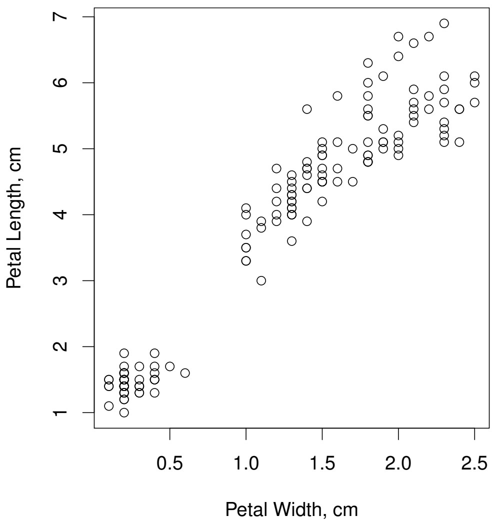
Yihui Xie, *Dynamic Documents with R and knitr*. Chapman & Hall/CRC Press, 2013, 190+xxvi pp. Paperback, US$59.95. ISBN 978-1482203530.



There are several reasons why this book might be of interest to a TeX user. First, LaTeX has a prominent place in the book. Second, the book describes a very interesting offshoot of literate programming, a topic traditionally popular in the TeX community. Third, since a number of TeX users work with data analysis and statistics, R could be a useful tool for them.

Since some *TUGboat* readers are likely not familiar with R, I would like to start this review with a short description of the software. R [1] is a free implementation of the S language (sometimes R is called GNU S). The latter is a language for statistical computations created at Bell Labs during its "Golden Age" of computing, when C, *awk*, Unix, et al. were developed at this famous institution. S is very convenient for a data exploration. For example, consider the dataset *iris* (included in the base R distribution) containing measurements of 150 flowers of *Iris setosa*, *Iris versicolor* and *Iris virginica* [2]. We may inquire whether petal length and petal width of irises are related. To this end we can plot the data:

```
plot(Petal.Length ~ Petal.Width,
    data = iris, xlab = "Petal Width, cm",
    ylab = "Petal Length, cm")
```

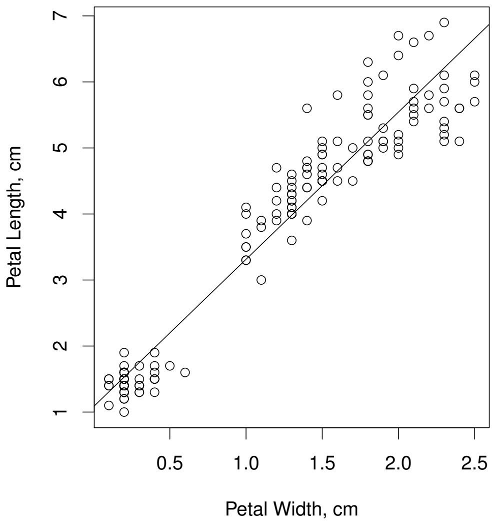


This plot shows an almost linear dependence between the parameters. We can try a linear fit for these data:

```
model <- lm(Petal.Length ~ Petal.Width,
    data = iris)
model$coefficients

## (Intercept) Petal.Width
##       1.084       2.230

summary(model)$r.squared

## [1] 0.9271
```

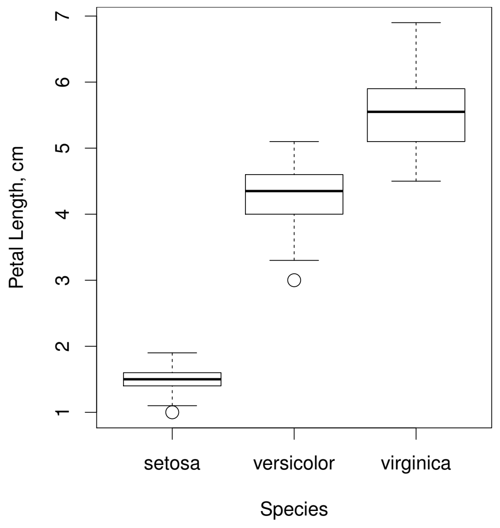
The large value of $R^2 = 0.9271$ indicates the good quality of the fit. Of course we can replot the data together with the prediction of the linear model:

```
plot(Petal.Length ~ Petal.Width,
    data = iris, xlab = "Petal Width, cm",
    ylab = "Petal Length, cm")
abline(model)
```



We can also study how petal length depends on the species of iris:

```
boxplot(Petal.Length ~ Species,
    data = iris, xlab = "Species",
    ylab = "Petal Length, cm")
```



This plot shows a significant difference between the petal lengths of the different species of iris. We could further investigate this difference using appropriate statistical tests, but this is out of scope for this very short introduction. Instead we refer the reader to the many books on S and R (for example, [3, 4]).

While interactive computations and data exploration are indispensable in such research, one often needs a permanent record that can be stored and shared with other people. A saved transcript of a computer session (or output of a batch job) provides such a record, but in a rather imperfect way. From the transcript one can see what we asked the computer and what the computer replied, but the most important parts of the exploration, namely, why did we ask these questions, which hypotheses were tested, and what our conclusions were, remain outside this record. We can add such information in comments to the R code, but it is awkward to express a complex discussion, often heavy with mathematics, using only an equivalent of an old-fashioned typewriter. This is why many commercial interactive computation systems offer so-called "notebook" interfaces, where calculations are interlaced with the more or less well typeset text and figures. Unfortunately these interfaces, being proprietary, cannot be easily integrated with scientific publishing software, and it takes a considerable effort to translate these "notebooks" into papers and reports. The power of free software is the possibility to combine different building blocks into something new, often not foreseen by their original authors. Thus an idea to combine a free statistical engine and a free typesetting engine is a natural one.

This idea is close to that of literate programming [5]. We have a master document which describes our investigation and contains blocks of text, possibly with equations and figures, and blocks of computational code, that can also generate text, equations and figures. As in the conventional literate programming paradigm, this document can be *weave*d or *tangle*d. Weaving creates a typeset document, while tangling extracts the program (almost) free of comments. However, there is an important difference between the conventional literate programming and the literate programming of data exploration. In the conventional case we are usually interested in the program itself, which is supposed to run many times with different inputs giving different results. For the data exploration the input is usually as important as the program. In most cases we want to run the program just once and show the results. Therefore weaving becomes a more complex process, involving running the tangled program and inserting the results in the appropriate places of the typeset document. On the other hand, tangling by itself is used more rarely (but is still useful in some cases, for example, to typeset this review, as described below).

The first (and a very successful) attempt to apply the ideas of literate programming to S was the package *Sweave* [6]. Uwe Ziegenhagen introduced the package to the TeX community in his brilliant talk at TUG 2010 [7]. In *Sweave* we write a file `source.rnw`, which looks like a TeX file, but includes special fragments between the markers `<<...>>` and `@` (this notation is borrowed from the *Noweb* literate programming system [8]). For example, the box plot above can be produced by the following fragment of an `.rnw` file:

```
% Boxplot in Sweave syntax
<<iris-boxplot, fig=TRUE>>=
boxplot (Petal.Length ~ Species,
     data=iris,
     xlab="Species",
     ylab="Petal Length, cm")
@
```

We also can "inline" R code using the command `\Sexpr`, for example,

```
  The large value of
  $R^2=\Sexpr{summary(model)$r.squared}$
  indicates the good quality of the fit.
```

Note the different usage of `$` inside `\Sexpr` (a part of R code) and outside it (TeX math mode delimiter).

When we feed the file `source.rnw` to *Sweave*, it runs the R code in the chunks, and replaces these

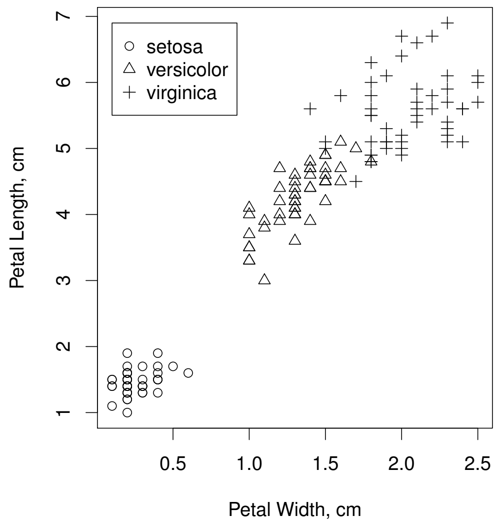


**Figure 1**: Petal widths and petal lengths for irises of different species

chunks and `\Sexpr` macros by the properly formatted results and/or the code itself. This produces a "weaved" file `source.tex`. We can control the process by the options inside `<<...>>`. For example, setting there `echo=FALSE`, we suppress the "echoing" of the source code. We can wrap a fragment into a `figure` environment and get a result like the one in Figure 1. Alternatively we can use R functions that output LaTeX code and, setting the proper options, get a table like Table 1.

These figures and tables, as well as inline expressions, can be configured to hide the code, for example for a formal publication. One can write a paper as an `.rnw` file and submit a PDF or TeX file to a journal without copying and pasting the results of the calculations, eliminating the risk of introducing new errors and typos. The versatility of this approach allows one to create quite varied publications, from a conference poster to a book. On the other hand, if the user does not suppress the code, a nicely formatted and reproducible lab report is produced.

Since *Sweave* was written, many packages have been devised to extend its capabilities and to add new features to it. At some point the user community felt the need for a refactoring of *Sweave* with better integration of the new features and a more modular design. The package *knitr* [9], which lists Yihui Xie as one of its principal authors, provides such refactoring. The name "knitr" is a play on "Sweave". The *knitr* functions performing weaving and tangling are called "knit" and "purl" correspondingly.

**Table 1**: Linear model for iris petal length and width

| Parameter | Est. | $\sigma$ | $t$ | $p$-value |
|---|---|---|---|---|
| Intercept | 1.08 | 0.0730 | 14.8 | $4.04 \times 10^{-31}$ |
| Slope | 2.23 | 0.0514 | 43.4 | $4.68 \times 10^{-86}$ |

An important feature of *knitr* is the closeness of its syntax to that of *Sweave*. Up to version 1.0 *knitr* supported full compatibility with *Sweave*, but even today most *Sweave* code runs without problems in *knitr* (the function `Sweave2knitr` can help in the remaining cases). For example, the code producing the iris boxplot above becomes the following in *knitr*:

```
% Boxplot in knitr syntax
<<iris-boxplot>>=
boxplot (Petal.Length ~ Species,
     data=iris,
     xlab="Species",
     ylab="Petal Length, cm")
@
```

The only difference between this code and *Sweave* code is the absence of `fig=TRUE` option, which is not needed for the new package. However, the *Sweave* code above still works in *knitr*. This makes the switch to the new package rather easy.

It should be noted that some of the features of *knitr* discussed below are also available in *Sweave* when using add-on packages; *knitr* offers them "out of the box" and better integrates them.

For example, *knitr* has two dozen or so different graphics formats (or "devices") to save the graphics. One very interesting device is *tikz*, which can be used to add TeX annotations to the plots.

Another useful feature of *knitr* is the option of caching the computation results. R calculations can often take a significant time. The full recompilation of all chunks after a mere wording change in the TeX part may be too slow for a user. This problem is addressed by the options `cache` and `dependson` in *knitr*. They instruct R to recalculate only the modified chunks and the chunks that depend on them (a clever hashing of the code is used to determine whether a chunk was modified between the runs).

The typesetting of the input code in *knitr* is closer to the requirements of literate programming than the simple *Sweave* output: *knitr* can recognize R language elements like keywords, comments, variables, strings etc., and highlight them according to user's specifications.

While these features are nice, the real selling points of *knitr* are its flexibility and modularization. The package has many options for fine-tuning the output. The modular design of the package makes adding new options just a matter of writing a new "hook" (an R function called when processing a chunk). Since the chunk headers in *knitr*, unlike *Sweave*, are evaluated as R expressions, one can write quite sophisticated "hooks".

The modularity of *knitr* allowed the authors to introduce new typesetting engines. Besides LaTeX, the package can work with other markup languages, e. g. HTML, Markdown and reStructuredText.

The flexibility and ease of customization of *knitr* are especially useful for book publishing. Yihui Xie lists several books created with *knitr*. Barbara Beeton [10] reports a positive experience with AMS publishing such books.

This review turned out to be a test of the flexibility of *knitr*. At this point an astute reader may have already guessed that it was written as an `.rnw` file. However, that is not the full story. *TUGboat* reviews are published in the journal, and the HTML versions are posted on the web at `http://www.tug.org/books`. As a matter of policy, the HTML version is automatically generated from the same source as the hard copy. This created a certain challenge for this review. First, the hard copy uses plots in PDF format, while the Web version uses them in PNG format. Second, due to the differences in column widths R code should be formatted differently for the print and the Web versions. Third, code highlighting of R chunks was done using font weights for the print version and using colors for the Web version. The following work flow was used: (1) The review was written as an `.rnw` file. (2) A `.tex` file was knit and an `.R` file was purled from the `.rnw` source. (3) The `.tex` file was copy-edited by the *TUGboat* editors. (4) A `.pdf` output for the journal was produced from this `.tex` file. (5) A special `.Rhtml` file was produced from the same `.tex` file with *tex4ht*. This file included commands that read the `.R` program and inserted the code chunks in the proper positions. (6) This `.Rhtml` file was knit again to produce HTML output and the images for the Web. All this but the actual writing and copy-editing was done by scripts without human intervention.

The package *knitr* is being actively developed, and many new features are being added. I would like to mention a feature that I miss in the package. The default graphics device, PDF, uses fonts different from the fonts of the main document, and does not allow TeX on the plots. While the *tikz* device is free of this limitation, it is very slow and strains TeX memory capacity when producing large plots. I think the trick used by the *Gnuplot pslatex* terminal [11] might be very useful. This terminal creates two files: a TeX file with the textual material put in the proper places using the `picture` environment, and a graphics file with the graphical material, included through the `\includegraphics` command. This terminal is much faster than *tikz* and more flexible than *pdf*. Moreover, since the TeX file is evaluated in the context of the main document, one can include `\ref` and other LaTeX commands in the textual part.

To use *knitr* on the most basic level it is enough to know two simple rules and one option: (1) put R code between `<<...>>` and `@`; (2) use `\Sexpr{`*code*`}` for inline R code; (3) use `echo=FALSE` to suppress echoing the input if necessary. However, since *knitr* has many options and is highly customizable, one might want to learn more about it to use it efficiently. This justifies the existence of books like the one by Yihui Xie. While there are many free sources of information about *knitr*, including its manual and the author's site (`http://yihui.name/knitr`), there are important reasons why a user would consider investing about $60 in the book.

The book provides a systematic description of the package, including its concepts, design principles, and philosophy. It also has many examples, well thought out advice, and useful tips and tricks.

Here is just one of the techniques I learned from the book. There are two ways of presenting calculation results: using the option `echo=TRUE` (default) we typeset R code, while using the option `echo=FALSE` we suppress it. The inclusion of the code has both advantages and disadvantages: we tell the reader more with the code included, but we risk overwhelming and confusing the audience with too much detail. One way to solve this problem is to put only the results in the main text, and show the code itself in an appendix. Of course we do not want to copy and paste the same code twice; a computer can take care of this much better. The book describes how to make this automatic by putting in the appendix these lines:

```
% List the code of all chunks
<<ref.label=all_labels(), eval=F, echo=T>>=
@
```

There are many other useful tips on the pages of this book. Some of them can be found at `http://yihui.name/knitr/` and `http://tex.stackexchange.com/`. Having all these tips collected in a book saves time, however, and reading the book helps to learn *knitr* and the general interaction of TeX and R.

The book is well written. It has introductory material useful for novices as well as advice for more seasoned users, all explained in conversational English without unnecessary technical jargon. The book describes several integrated work environments (RStudio, LyX, Emacs/ESS) and the interaction of *knitr* with popular LaTeX packages such as *beamer* and *listings*. It discusses in depth package options for formatting input and output, graphics, caching, code reuse, cross-referencing and other purposes. Besides "traditional" applications such as publishing books and reports, the author describes the use of *knitr* for writing blog entries, student homework, etc. The book covers inclusion of fragments of code written in languages other than R (Python, Perl, C/C++, etc.), dynamic plots with the *animate* LaTeX package, macro preprocessing and many other topics.

The book itself is written, of course, in *knitr* and LaTeX (with LyX as the integrated environment). It is well typeset and nicely printed. Regrettably, I find its index rather inadequate: it has only two pages and omits many key concepts. A book like this should at a minimum have an alphabetic list of all package options. However, other than this, I like the way the book is written and published.

While I have been using *Sweave* and then *knitr* for several years, I still learned many new useful things from the book. Thus I think it is worth investing money and reading time.

I think the book deserves a place on the bookshelves of both new *and* experienced R and TeX users.

## References

[1] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2013. ISBN 3-900051-07-0.

[2] Edgar Anderson. The irises of the Gaspe Peninsula. *Bulletin of the American Iris Society*, 59:2–5, 1935.

[3] Peter Dalgaard. *Introductory Statistics with R.* Statistics and Computing. Springer, New York, second edition, 2008.

[4] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S.* Statistics and Computing. Springer, New York, fourth edition, 2010.

[5] Donald Ervin Knuth. *Literate Programming.* Number 27 in CSLI lecture notes. Center for the Study of Language and Information, Stanford, CA, 1992.

[6] Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In Wolfgang Härdle and Bernd Rönz, editors, *Compstat 2002 — Proceedings in Computational Statistics*, pages 575–580. Physica Verlag, Heidelberg, 2002. ISBN 3-7908-1517-9.

[7] Uwe Ziegenhagen. Dynamic reporting with R/Sweave and LaTeX. *TUGboat*, 31(2):189–192, 2010. `http://tug.org/TUGboat/tb31-2/tb98ziegenhagen.pdf`.

[8] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994. `http://www.cs.tufts.edu/~nr/noweb/`.

[9] Alastair Andrew, Alex Zvoleff, Brian Diggs, Cassio Pereira, Hadley Wickham, Heewon Jeon, Jeff Arnold, Jeremy Stephens, Jim Hester, Joe Cheng, Jonathan Keane, J.J. Allaire, Johan Toloe, Kohske Takahashi, Michel Kuhlmann, Nacho Caballero, Nick Salkowski, Noam Ross, Ramnath Vaidyanathan, Richard Cotton, Romain François, Sietse Brouwer, Simon de Bernard, Taiyun Wei, Thibaut Lamadon, Tom Torsney-Weir, Trevor Davis, Weicheng Zhu, Wush Wu, and Yihui Xie. *knitr: A general-purpose package for dynamic report generation in R*, 2013. R package version 1.5.

[10] Barbara Beeton. Private communication, 2014.

[11] Philipp K. Janert. *Gnuplot in Action. Understanding Data with Graphs.* Manning Publications Co., 2009.

◇ Boris Veytsman
Systems Biology School and Computational Materials Science Center, MS 6A2
George Mason University
Fairfax, VA 22030
`borisv (at) lk dot net`
`http://borisv.lk.net`

## TUG financial statements for 2013

Karl Berry, TUG treasurer

The financial statements for 2013 have been reviewed by the TUG board but have not been audited. As a US tax-exempt organization, TUG's annual information returns are publicly available on our web site: `http://tug.org/tax-exempt`.

### Revenue (income) highlights

Membership dues revenue was down about 4% in 2013 compared to 2012; product sales were also down, while contributions were substantially up. Services income is a new category, reflecting TUG's accepting payments on behalf of font workshops and other TeX-related projects. Interest and advertising income were slightly down. Overall, 2013 income was down 3%.

### Cost of Goods Sold and Expenses highlights, and the bottom line

Payroll, *TUGboat*, DVD production, postage, and other office overhead continue to be the major expense items. All were nearly as budgeted; overall, 2013 expenses were up about 6% from 2012.

The "prior year adjustment" compensates for estimates made in closing the books for the prior year; in 2013 the total adjustment was positive: $194.

The bottom line for 2013 was positive: ≈ $2,000.

### Balance sheet highlights

TUG's end-of-year asset total is down around $3,000 (2%) in 2013 compared to 2012.

The Committed Funds are administered by TUG specifically for designated projects: LaTeX, CTAN, the TeX development fund, and so forth. Incoming donations have been allocated accordingly and are disbursed as the projects progress (disbursements in 2013 substantially account for the overall lower asset balance). TUG charges no overhead for administering these funds.

The Prepaid Member Income category is member dues that were paid in earlier years for the current year (and beyond). Most of this liability (the 2013 portion) was converted into regular Membership Dues in January of 2013.

The payroll liabilities are for 2013 state and federal taxes due January 15, 2013.

### Summary

The membership fees remained unchanged in 2013; the last increase was in 2010. TUG remains financially solid as we enter another year.

### TUG12/31/2013 (vs 2012) Revenue, Expense

| | Jan - Dec 13 | Jan - Dec 12 |
|---|---|---|
| Ordinary Income/Expense | | |
| Income | | |
| Membership Dues | 94,800 | 98,725 |
| Product Sales | 7,498 | 11,351 |
| Contributions Income | 9,126 | 6,821 |
| Annual Conference | | 1,222 |
| Interest Income | 625 | 832 |
| Advertising Income | 410 | 490 |
| Services Income | 3,493 | |
| Total Income | 115,952 | 119,441 |
| Cost of Goods Sold | | |
| TUGboat Prod/Mailing | 24,850 | 21,674 |
| Software Production/Mailing | 3,038 | 2,685 |
| Postage/Delivery - Members | 2,923 | 2,566 |
| Lucida Sales Accrual B&H | 2,875 | 4,835 |
| Member Renewal | 417 | 444 |
| Total COGS | 34,103 | 32,204 |
| Gross Profit | 81,849 | 87,237 |
| Expense | | |
| Contributions made by TUG | 3,324 | 2,000 |
| Office Overhead | 12,121 | 12,804 |
| Payroll Exp | 64,486 | 65,375 |
| Interest Expense | 94 | |
| Total Expense | 80,025 | 80,179 |
| Net Ordinary Income | 1,824 | 7,058 |
| Other Income | | |
| Prior year adjust | 194 | 222 |
| Total Other Income | 194 | 222 |
| Net Income | 2,018 | 7,280 |

### TUG 12/31/2013 (vs 2012) Balance Sheet

| | Dec 31, 13 | Dec 31, 12 |
|---|---|---|
| ASSETS | | |
| Current Assets | | |
| Total Checking/Savings | 186,696 | 187,506 |
| Accounts Receivable | 180 | 2,496 |
| Total Current Assets | 186,876 | 190,002 |
| TOTAL ASSETS | 186,876 | 190,002 |
| LIABILITIES & EQUITY | | |
| Liabilities | | |
| Committed Funds | 27,712 | 31,384 |
| TUG conference | | -250 |
| Administrative Services | 4,879 | |
| Deferred contributions | 90 | |
| Prepaid member income | 4,550 | 11,315 |
| Payroll Liabilities | 1,100 | 1,024 |
| Total Current Liabilities | 38,330 | 43,473 |
| TOTAL LIABILITIES | 38,330 | 43,473 |
| Equity | | |
| Unrestricted | 146,529 | 139,249 |
| Net Income | 2,017 | 7,280 |
| Total Equity | 148,546 | 146,529 |
| TOTAL LIABILITIES & EQUITY | 186,876 | 190,002 |

## TUG Institutional Members

American Mathematical Society, *Providence*, *Rhode Island*

Aware Software, Inc., *Midland Park*, *New Jersey*

Center for Computing Sciences, *Bowie*, *Maryland*

CSTUG, *Praha*, *Czech Republic*

IBM Corporation, T J Watson Research Center, *Yorktown*, *New York*

Institute for Defense Analyses, Center for Communications Research, *Princeton*, *New Jersey*

Marquette University, Department of Mathematics, Statistics and Computer Science, *Milwaukee*, *Wisconsin*

Masaryk University, Faculty of Informatics, *Brno*, *Czech Republic*

MOSEK ApS, *Copenhagen*, *Denmark*

New York University, Academic Computing Facility, *New York*, *New York*

Springer-Verlag Heidelberg, *Heidelberg*, *Germany*

StackExchange, *New York City*, *New York*

Stanford University, Computer Science Department, *Stanford*, *California*

Stockholm University, Department of Mathematics, *Stockholm*, *Sweden*

University College, Cork, Computer Centre, *Cork*, *Ireland*

Université Laval, *Ste-Foy*, *Québec*, *Canada*

University of Ontario, Institute of Technology, *Oshawa*, *Ontario*, *Canada*

University of Oslo, Institute of Informatics, *Blindern*, *Oslo*, *Norway*

University of Wisconsin, Biostatistics & Medical Informatics, *Madison*, *Wisconsin*

VTEX UAB, *Vilnius*, *Lithuania*

## TEX Consultants

The information here comes from the consultants themselves. We do not include information we know to be false, but we cannot check out any of the information; we are transmitting it to you as it was given to us and do not promise it is correct. Also, this is not an official endorsement of the people listed here. We provide this list to enable you to contact service providers and decide for yourself whether to hire one.

TUG also provides an online list of consultants at `http://tug.org/consultants.html`. If you'd like to be listed, please see that web page.

**Aicart Martinez, Mercè**
Tarragona 102 $4^o$ $2^a$
08015 Barcelona, Spain
+34 932267827
Email: `m.aicart (at) ono.com`
Web: `http://www.edilatex.com`

We provide, at reasonable low cost, LATEX or TEX page layout and typesetting services to authors or publishers world-wide. We have been in business since the beginning of 1990. For more information visit our web site.

**Dangerous Curve**
PO Box 532281
Los Angeles, CA 90053
+1 213-617-8483
Email: `typesetting (at) dangerouscurve.org`
Web: `http://dangerouscurve.org/tex.html`

We are your macro specialists for TEX or LATEX fine typography specs beyond those of the average LATEX macro package. If you use XeTEX, we are your microtypography specialists. We take special care to typeset mathematics well.

Not that picky? We also handle most of your typical TEX and LATEX typesetting needs.

We have been typesetting in the commercial and academic worlds since 1979.

Our team includes Masters-level computer scientists, journeyman typographers, graphic designers, letterform/font designers, artists, and a co-author of a TEX book.

**Latchman, David**
4113 Planz Road Apt. C
Bakersfield, CA 93309-5935
+1 518-951-8786
Email: `david.latchman (at) texnical-designs.com`
Web: `http://www.texnical-designs.com`

LATEX consultant specializing in: the typesetting of books, manuscripts, articles, Word document conversions as well as creating the customized packages to meet your needs.

Call or email to discuss your project or visit my website for further details.

**Peter, Steve**
295 N Bridge St.
Somerville, NJ 08876
+1 732 306-6309
Email: `speter (at) mac.com`
Specializing in foreign language, multilingual, linguistic, and technical typesetting using most flavors of TeX, I have typeset books for Pragmatic Programmers, Oxford University Press, Routledge, and Kluwer, among others, and have helped numerous authors turn rough manuscripts, some with dozens of languages, into beautiful camera-ready copy. In addition, I've helped publishers write, maintain, and streamline TeX-based publishing systems. I have an MA in Linguistics from Harvard University and live in the New York metro area.

**Shanmugam, R.**
No. 38/1 (New No. 65), Veerapandian Nagar, Ist St.
Choolaimedu, Chennai-600094, Tamilnadu, India
+91 9841061058
Email: `rshanmugam92 (at) gmail.com`
As a Consultant, I provide consultation, training, and full service support to individuals, authors, typesetters, publishers, organizations, institutions, etc. I support leading BPO/KPO/ITES/Publishing companies in implementing latest technologies with high level automation in the field of Typesetting/Prepress, ePublishing, XML2PAGE, WEBTechnology, DataConversion, Digitization, Cross-media publishing, etc., with highly competitive prices.
I provide consultation in building business models & technology to develop your customer base and community, streamlining processes in getting ROI on our workflow, New business opportunities through improved workflow, Developing eMarketing/E-Business Strategy, etc. I have been in the field BPO/KPO/ITES, Typesetting, and ePublishing for nearly 20 years, and handled various projects. I am a software consultant with Master's Degree. I have sound knowledge in TeX, LaTeX2$\varepsilon$, XMLTeX, Quark, InDesign, XML, MathML, eBooks, ePub, Mobi, iBooks, DTD, XSLT, XSL-FO, Schema, ebooks, OeB, etc.

**Sievers, Martin**
Klaus-Kordel-Str. 8, 54296 Trier, Germany
+49 651 4936567-0
Email: `info (at) schoenerpublizieren.com`
Web: `http://www.schoenerpublizieren.com`
As a mathematician with ten years of typesetting experience I offer TeX and LaTeX services and consulting for the whole academic sector (individuals, universities, publishers) and everybody looking for a high-quality output of his documents.

From setting up entire book projects to last-minute help, from creating individual templates, packages and citation styles (BibTeX, biblatex) to typesetting your math, tables or graphics — just contact me with information on your project.

**Sofka, Michael**
8 Providence St.
Albany, NY 12203
+1 518 331-3457
Email: `michael.sofka (at) gmail.com`
Skilled, personalized TeX and LaTeX consulting and programming services.

I offer over 25 years of experience in programming, macro writing, and typesetting books, articles, newsletters, and theses in TeX and LaTeX: Automated document conversion; Programming in Perl, C, C++ and other languages; Writing and customizing macro packages in TeX or LaTeX; Generating custom output in PDF, HTML and XML; Data format conversion; Databases.

If you have a specialized TeX or LaTeX need, or if you are looking for the solution to your typographic problems, contact me. I will be happy to discuss your project.

**Veytsman, Boris**
46871 Antioch Pl.
Sterling, VA 20164
+1 703 915-2406
Email: `borisv (at) lk.net`
Web: `http://www.borisv.lk.net`
TeX and LaTeX consulting, training and seminars. Integration with databases, automated document preparation, custom LaTeX packages, conversions and much more. I have about eighteen years of experience in TeX and three decades of experience in teaching & training. I have authored several packages on CTAN, published papers in TeX related journals, and conducted several workshops on TeX and related subjects.

**Young, Lee A.**
127 Kingfisher Lane
Mills River, NC 28759
+1 828 435-0525
Email: `leeayoung (at) morrisbb.net`
Web: `http://www.thesiseditor.net`
Copyediting your `.tex` manuscript for readability and mathematical style by a Harvard Ph.D. Your `.tex` file won't compile? Send it to me for repair. Experience: edited hundreds of ESL journal articles, economics and physics textbooks, scholarly monographs, LaTeX manuscripts for the Physical Review; career as professional, published physicist.

# Calendar

## 2014

| | |
|---|---|
| Apr 11 – 14 | DANTE Frühjahrstagung (25th anniversary of DANTE e.V.) and 50th meeting, Universität Heidelberg, Germany. `www.dante.de/events/dante2014.html` |
| Apr 30 – May 4 | BachoTeX 2014: 22nd BachoTeX Conference, "What can typography gain from electronic media?", Bachotek, Poland. `www.gust.org.pl/bachotex/2014` |
| Jun 9 – Aug 1 | Rare Book School, University of Virginia, Charlottesville, Virginia. Many one-week courses on type, bookmaking, printing, and related topics. `www.rarebookschool.org/schedule` |
| Jun 23 – 26 | Book history workshop, École de l'institut d'histoire du livre, Lyon, France. `ihl.enssib.fr` |
| Jul 2 – 4 | International Society for the History and Theory of Intellectual Property (ISHTIP), 6th Annual Workshop, "The Instability of Intellectual Property". Uppsala, Sweden. `www.ishtip.org/?p=596` |
| Jul 8 – 12 | Digital Humanities 2014, Alliance of Digital Humanities Organizations, Lausanne, Switzerland. `dh2014.org`, `adho.org/conference` |

**TUG 2014**
**Mark Spencer Hotel, Portland, Oregon.**

| | |
|---|---|
| Jul 27 | Evening reception |
| Jul 28 – 30 | The 35th annual meeting of the TeX Users Group. Presentations covering the TeX world. `tug.org/tug2014` |
| Jul 28 | LaTeX workshop (concurrent) |

| | |
|---|---|
| Jul 30 – Aug 3 | TypeCon 2014: "Capitolized", Washington, DC. `typecon.com` |
| Aug 4 – 8 | Balisage: The Markup Conference, Washington, DC. `www.balisage.net` |
| Aug 10 – 14 | SIGGRAPH 2014, Vancouver, British Columbia. `s2014.siggraph.org` |
| Aug 11 | *TUGboat* **35**:2, submission deadline (TUG 2014 proceedings) |
| Sep 8 – 9 | "Forms and formats: Experimenting with print, 1695-1815", Centre for the Study of the Book, Bodleian Library, University of Oxford, UK. `www.bodleian.ox.ac.uk/csb/community` |
| Sep 8 – 13 | 8th International ConTeXt Meeting, Bassenge, Belgium. `meeting.contextgarden.net/2014` |
| Sep 14 – 19 | XML Summer School, St Edmund Hall, Oxford University, Oxford, UK. `xmlsummerschool.com` |
| Sep 16 – 19 | ACM Symposium on Document Engineering, Fort Collins, Colorado. `www.doceng2014.org` |
| Sep 17 – 21 | Association Typographique Internationale (ATypI) annual conference, Theme: "Point Counter Point", Barcelona, Spain. `www.atypi.org` |
| Sep 17 – 21 | SHARP 2014, "Religions of the Book", Society for the History of Authorship, Reading & Publishing, Antwerp, Belgium, `www.sharpweb.org` |
| Oct 3 | *TUGboat* **35**:3, submission deadline. |
| Oct 3 – 5 | Oak Knoll Fest XVIII, and Fine Press Book Association annual meeting, New Castle, Delaware. `www.oakknoll.com/fest` |
| Nov 8 – 9 | The Twelfth International Conference on Books, Publishing, and Libraries, "Disruptive Technologies and the Evolution of Book Publishing and Library Development", Simmons College, Boston, Massachusetts. `booksandpublishing.com/the-conference` |

*Status as of 25 March 2014*

For additional information on TUG-sponsored events listed here, contact the TUG office (+1 503 223-9994, fax: +1 815 301-3568. e-mail: `office@tug.org`). For events sponsored by other organizations, please use the contact address provided.

A combined calendar for all user groups is online at `texcalendar.dante.de`.

Other calendars of typographic interest are linked from `tug.org/calendar.html`.

**The 35th Annual Meeting of the TEX Users Group**

**July 28–30, 2014**

**Mark Spencer Hotel**
**Portland, Oregon, USA**

**http://tug.org/tug2014 ▪ tug2014@tug.org**

| | |
|---|---|
| April 15 | — bursary application deadline |
| May 16 | — presentation proposal deadline |
| May 23 | — early bird registration deadline |
| June 6 | — preprint submission deadline |
| June 30 | — hotel reservation discount deadline |
| | — (but book earlier!) |
| July 28–30 | — conference |
| July 28 | — concurrent LATEX workshop |
| August 11 | — deadline for final papers for proceedings |

*Sponsored by the TEX Users Group and DANTE e.V.*

**TUG**BOAT Volume 35 (2014), No. 1