TUGBOAT

The Communications of the TEX Users Group

Volume 15, Number 3, September 1994
1994 Annual Meeting Proceedings

TeX Users Group

Memberships and Subscriptions

TUGboat (ISSN 0896-3207) is published quarterly by the TeX Users Group, Balboa Building, Room 307, 735 State Street, Santa Barbara, CA 93101, U.S.A.

1994 dues for individual members are as follows:

- Ordinary members: $60
- Students: $30

Membership in the TeX Users Group is for the calendar year, and includes all issues of *TUGboat* and *TeX and TUG NEWS* for the year in which membership begins or is renewed. Individual membership is open only to named individuals, and carries with it such rights and responsibilities as voting in the annual election. A membership form is provided on page ???.

TUGboat subscriptions are available to organizations and others wishing to receive *TUGboat* in a name other than that of an individual. Subscription rates: North America $60 a year; all other countries, ordinary delivery $60, air mail delivery $80.

Second-class postage paid at Santa Barbara, CA, and additional mailing offices. Postmaster: Send address changes to *TUGboat*, TeX Users Group, P. O. Box 869, Santa Barbara, CA 93102-0869, U.S.A.

Institutional Membership

Institutional Membership is a means of showing continuing interest in and support for both TeX and the TeX Users Group. For further information, contact the TUG office.

TUGboat © Copyright 1994, TeX Users Group

Permission is granted to make and distribute verbatim copies of this publication or of individual items from this publication provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this publication or of individual items from this publication 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.

Permission is granted to copy and distribute translations of this publication or of individual items from this publication into another language, under the above conditions for modified versions, except that this permission notice may be included in translations approved by the TeX Users Group instead of in the original English.

Some individual authors may wish to retain traditional copyright rights to their own articles. Such articles can be identified by the presence of a copyright notice thereon.

Printed in U.S.A.

Board of Directors

Donald Knuth, *Grand Wizard of TeX-arcana*[†]
Christina Thiele, *President*[*]
Michel Goossens[*], *Vice President*
George Greenwade[*], *Treasurer*
Peter Flynn[*], *Secretary*
Barbara Beeton
Johannes Braams, *Special Director for NTG*
Mimi Burbank
Jackie Damrau
Luzia Dietsche
Michael Doob
Michael Ferguson
Bernard Gaulle, *Special Director for GUTenberg*
Yannis Haralambous
Dag Langmyhr, *Special Director for the Nordic countries*
Nico Poppelier
Jon Radel
Sebastian Rahtz
Tom Rokicki
Chris Rowley, *Special Director for UKTeXUG*
Raymond Goucher, *Founding Executive Director*[†]
Hermann Zapf, *Wizard of Fonts*[†]

[*] *member of executive committee*
[†] *honorary*

Addresses

All correspondence, payments, etc.
TeX Users Group
P. O. Box 869
Santa Barbara, CA 93102-0869 USA

Parcel post, delivery services:
TeX Users Group
Balboa Building
Room 307
735 State Street
Santa Barbara, CA 93101
USA

Telephone

805-963-1338

Fax

805-963-8358

Electronic Mail

(Internet)
General correspondence:
`TUG@tug.org`
Submissions to *TUGboat*:
`TUGboat@Math.AMS.org`

TeX is a trademark of the American Mathematical Society.

1994 Annual Meeting Proceedings
TeX Users Group
Fifteenth Annual Meeting
University of California, Santa Barbara, July 31–August 4, 1994

TUGBOAT

COMMUNICATIONS OF THE TeX USERS GROUP

TUGBOAT EDITOR BARBARA BEETON
PROCEEDINGS EDITORS MICHEL GOOSSENS
SEBASTIAN RAHTZ

VOLUME 15, NUMBER 3 • SEPTEMBER 1994
SANTA BARBARA • CALIFORNIA • U.S.A.

Editorial and Production Notes

These *Proceedings* were prepared with TeX on various Unix workstations at CERN in Geneva. PostScript files for a Linotronic typesetter at 1270 dpi resolution were generated with Tom Rokicki's `dvips` program. From these files Philip Taylor produced the bromides on the Linotronic of the Computing Centre of the University of London. The color pages were completely done in the United States.

The present *Proceedings* are typeset in the Lucida Bright typeface designed by Bigelow & Holmes. For LaTeX the `lucbr` package (coming with LaTeX2ε in the PSNFSS system) for defining the fonts was used and a scaling factor of .94 has been applied to make the pages come out at an information density close to that of Computer Modern at 10pt. The complete set of fonts used is LucidaBright for text, LucidaSans for sans serif, LucidaTypewriter for teletype, and LucidaNewMath for the maths.

The authors sent their source files electronically via electronic mail or deposited them with *ftp* on a CERN machine. Most referees were also able to use *ftp* to obtain a PostScript copy of the paper they had to review, and I got their comments, if practical, via email, which made communication relatively straightforward and fast. I would like to thank the authors for their collaboration in keeping (mostly) to the original production schedule. I also want to express my gratitude to the various referees, who kindly agreed to review the paper assigned to them. I am convinced that their comments and suggestions for improvements or clarifications have made the papers clearer and more informative.

Eight of the contributed papers were in plain TeX while the others used LaTeX. All files associated to a given paper reside in a separate subdirectory in our `tug94/papers` directory, and each of the papers is typeset as a separate job. A `makefile` residing in our `tug94/papers/tug94` directory takes care that each paper is picked up from its directory and is processed with the right parameters. Information about the page numbers for the given job is written to the `aux` file using the `\AtEndDocument` command for LaTeX and by redefining the `\endarticle` command for plain TeX. A `sed` script then collects this information and writes it into a master file. This master file is read in a subsequent run by using the `\AtBeginDocument` command for LaTeX and by redefining the `\article` command for plain TeX.

All LaTeX files were run in native LaTeX2ε mode (if they were not already coded in LaTeX2ε—about half of the LaTeX papers were—it was in most cases sufficient to replace `\begin{documentstyle}` by `\begin{documentclass}`). At CERN we run TeX version 3.1415, based on Karl Berry's `web2c-6.1` directory structure. This system could be used for most papers without problems, but Haralambous' Ω

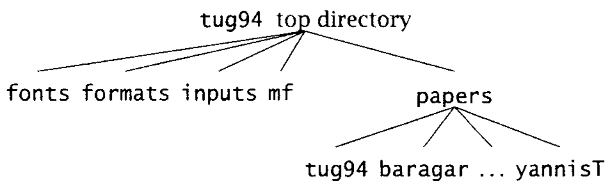


Figure 1: The directory structure for preparing the TUG94 *Proceedings*

(`yannisO`), and Phil Taylor's NTS paper, needed the TeX--XeT extensions, which have not yet been ported to that latest version of `web2c`. Therefore we had to build two special formats (one for LaTeX2ε, and one for plain) with the TeX--XeT mods and the older TeX3.141/`web2c-d`. The fonts used in Haralambous's Tiqwah paper needed 60 instead of the standard 50 `fontdimens`, so we also had to recompile METAFONT.

When the dvi-files were translated into PostScript with `dvips`, METAFONT would generate the font bitmap pk files on the fly, as they were needed, with the desired `mode_def`. In total 334 supplementary METAFONT source files were received for running the various papers in the *Proceedings*.

Although most pictures were available as Encapsulated PostScript files, for two articles (the one by Sofka, and the *BM2FONT* paper by Sowa) they could not be printed. Therefore we pasted originals obtained from the respective authors into the relevant places in the text.

Acknowledgements

These *Proceedings* would never have been ready in time were it not for the help of Sebastian Rahtz during the final stages of the production cycle. Building upon his experience gained last year when editing the TUG93 *Proceedings*, he developed a vastly improved production system for the generation of this year's *Proceedings*. Together we translated the old *TUGboat* styles into LaTeX2ε classes, and used these for all LaTeX runs. With the help of Oren Patashnik and Joachim Schrod we also developed a first version of a Chicago-like *TUGboat* BibTeX bibliography style and introduced the corresponding necessary changes into the class files.

I also want to thank Barbara Beeton, Mimi Burbank, Pierre MacKay, and Christina Thiele who, together, have reread the preprint versions of all papers. They have pointed out several remaining typos and provided me and the authors with many useful comments and suggestions for improvement. Last but not least I want to acknowledge the competence and dedication of Phil Taylor (RHBNC, University of London) during the final production stage of going to film.

Michel Goossens

Innovation

The 15th Annual TEX Users Group Meeting, Santa Barbara, USA

Abstract

TUG'94 was organized by:

Chairperson:	Patricia Monohon
Bursary:	Bernard Gaulle
Culture and Events:	Janet Sullivan
Courses:	John Berlin
Proceedings:	Michel Goossens
Programme:	Malcolm Clark & Sebastian Rahtz
TUGly Telegraph	John Berlin & Malcolm Clark

Acknowledgements and Thanks

The organizers would like to publicly acknowledge the contributions made by several individuals, by TEX Local User Groups, or by companies to the Bursary and Social Funds, or who offered free copies of books or software to the participants. In particular we would like to thank DANTE e.V., GUTenberg, UK-TUG, and TUG, as well as Addison-Wesley, O'Reilly & Associates, and Prime Time Freeware.

We would also like to mention the vendors: Addison-Wesley, Blue Sky Research, Kinch Computer Co., Micro Programs, Inc., Quixote Digital Typography, Springer Verlag, and Y&Y, who by their continuing support contribute to the success of the Annual TUG Meetings.

Special thanks go to Katherine Butterfield, Suki Bhurgi, and Wendy McKay for helping with staffing the on-campus TUG office.

Conference Programme

Sunday July 31st

Keynote

Lucida and TEX: lessons of logic and history: *Charles Bigelow*

Publishing, languages, literature and fonts

Real life book production—lessons learned from *The LATEX Companion*: *Frank Mittelbach and Michel Goossens*

Typesetting the holy Bible in Hebrew, with TEX: *Yannis Haralambous*

Adaptive character generation and spatial expressiveness: *Michael Cohen*

Humanist: *Yannis Haralambous*

Automatic conversion of METAFONT fonts to Type1 PostScript: *Basil Malyshev, presented by Alan Hoenig*

Monday August 1st

Keynote

Looking back at, and forward from, LATEX: *Leslie Lamport*

Colour and LATEX

The (Pre)History of Color in Rokicki's `dvips`: *James Hafner*

Advanced 'special' support in a dvi driver: *Tom Rokicki*

Colour separation and PostScript: *Angus Duggan*

Simple colour design and LATEX: *Sebastian Rahtz and Michel Goossens*

Printing colour pictures: *Friedhelm Sowa*

Color book production using TEX: *Michael Sofka*

Inside PSTricks: *Timothy van Zandt and Denis Girou, presented by Sebastian Rahtz*

A LATEX style file generator: *Jon Stenerson*

Document classes and packages in LATEX2ε: *Johannes Braams*

PostScript font support in LATEX2ε: *Alan Jeffrey*

Tuesday August 2nd

TEX Tools

BIBTEX 1.0: *Oren Patashnik*

A typesetter's toolkit: *Pierre MacKay*

Symbolic Computation for Electronic Publishing: *Michael P. Barnett and Kevin R. Perry*

Concurrent Use of Interactive TEX Previewer with an Emacs-type Editor: *Minato Kawaguti and Norio Kitajima*

An Indic TEX preprocessor — Sinhalese TEX: *Yannis Haralambous*

Pascal pretty-printing: an example of "preprocessing within TEX": *Jean-luc Doumont*

Wednesday August 3rd

Futures

Towards Interactivity for TEX: *Joachim Schrod*

The Floating World: *Chris Rowley and Frank Mittelbach*

Sophisticated page layout with TEX: *Don Hosek*

Progress in the Omega project: *John Plaice*

Object-Oriented Programming, Descriptive Markup, and TEX: *Arthur Ogawa*

An Object-Oriented Programming System in TEX: *William Erik Baxter*

A World Wide Web interface to CTAN: *Norm Walsh*

First applications of Ω: Adobe Poetica, Arabic, Greek, Khmer: *Yannis Haralambous and John Plaice*

ε-TEX & NTS: progress so far, and an invitation to discussion: *Philip Taylor, Jiří Zlatuška, Peter Breitenlohner and Friedhelm Sowa*

Thursday August 4th

Publishing and design

TEX innovations by the Louis-Jean Printing House: *Maurice Laugier and Yannis Haralambous*

Design by template in a production macro package: *Michael Downes*

Less is More: Restricting TEX's Scope Enables Complex Page Layouts: *Alan Hoenig*

Documents, Compuscripts, Programs and Macros: *Jonathan Fine, presented by Malcolm Clark*

Integrated system for encyclopaedia typesetting based on TEX: *Marko Grobelnik, Dunja Mladenić, Darko Zupanič and Borut Žnidar*

An Example of a Special Purpose Input Language to LATEX: *Henry Baragar and Gail E. Harris*

Colour Pages These are in a separate section at the back of these proceedings. They are referenced in the articles with the tag "Color Example".

Author page index

Henry Baragar: 388
Michael P. Barnett: 285
William Baxter: 331
Charles Bigelow: 169
Johannes Braams: 255
Michael Cohen: 192
Jean-luc Doumont: 302
Michael Downes: 360
Angus Duggan: 213
Jonathan Fine: 381
Denis Girou: 239
Michel Goossens: 170, 218
Marko Grobelnik: 386
James Lee Hafner: 201
Yannis Haralambous: 174, 199, 301, 344, 359
Gail E. Harris: 388
Alan Hoenig: 369
Don Hosek: 319
Alan Jeffrey: 263
Minato Kawaguti: 293
Norio Kitajima: 293
Maurice Laugier: 359
Basil K. Malyshev: 200
Pierre A. MacKay: 274
Frank Mittelbach: 170, 318
Dunja Mladenić: 386
Arthur Ogawa: 325
Oren Patashnik: 269
Kevin R. Perry: 285
John Plaice: 320, 344
Sebastian Rahtz: 218
Tomas Rokicki: 205
Chris Rowley: 318
Joachim Schrod: 309
Michael D. Sofka: 228
Friedhelm Sowa: 223
Jon Stenerson: 247
Philip Taylor: 353
Timothy Van Zandt: 239
Norman Walsh: 339
Borut Žnidar: 386
Darko Zupanič: 386

Lucida and TEX: lessons of logic and history

Charles Bigelow
Bigelow & Holmes, P.O. Box 1299, Menlo Park, CA 94026
bigelow@cs.stanford.edu

Abstract

The development of Lucida fonts for TEX included many lessons, some being simply the idiosyncracies of Don Knuth's self-taught opinions about typography, and others being important aspects of mathematical and scientific composition that are unknown to typographers. Another aspect of this talk is how typeface designs are conceived, created, developed, evolved, etc., which involves reference to Times and Computer Modern.

A paper similar in content was published elsewhere.[1]

[1] C. Bigelow and K. Holmes. *The Design of a Unicode Font.* Proceedings of RIDT'94. Electronic Publishing, Origination, Dissemination and Design, **6**(3), pages 289–306, 1993.

Real life book production—lessons learned from *The LaTeX Companion*

Michel Goossens
CERN, CN Division, CH1211 Geneva 23, Switzerland
michel.goossens@cern.ch

Frank Mittelbach
Zedernweg 62, D55128 Mainz, Germany
Mittelbach@mzdmza.zdv.Uni-Mainz.de

Abstract

Some aspects of the production of *The LaTeX Companion* are described.

Deciding to write a book

Text processing support staff at CERN, as, without doubt, in many other research institutes, universities or companies, had followed Leslie Lamport's advice in the LaTeX Reference manual (Lamport 1985), and developed a *Local Guide*, which describes how LaTeX can be used on CERN's various computer platforms, explains which interesting style files are available, and provides a set of examples and pointers to further information. Alexander Samarin and one of the authors (Michel) had long planned to expand the material in that guide, to make it more generally available.

When Frank visited CERN in April 1992 to give a presentation on the LaTeX3 project, we talked to him about our idea. We outlined vaguely what we wanted to write, and Frank found the idea "interesting". After he got back home he proposed to talk to Peter Gordon of Addison-Wesley, to see whether they would be interested. They were, and at that point, all three of us decided that it would be a good idea to collaborate.

Defining contents and time scale

At the end of June 1992 Michel had a first meeting with Frank in Mainz, where they wrote a detailed table of contents, down to the section level, which contained in most cases an extended outline, with an estimated number of pages.

Work by each of the authors, as assigned in the plan discussed in Mainz, continued over the summer, so that by the time of the Prague EuroTeX Conference in September 1992 we already had a nice 300 page preprint, which we discussed in great detail during various meetings in the Golden City. We also met with Peter Gordon, our editor at Addison-Wesley, and finalized aspects of the contract we had been discussing previously. The final date for delivery of the compuscript was tentatively set for April 1st 1993, in order to having a chance of getting the book printed for the TUG Conference in Aston (Birmingham, UK) in the summer.

Further work on the book during the autumn and the winter was essentially carried out by Frank and Michel, since Alexander went back to Russia at the end of October 1992, and when he finally returned to Geneva in March 1993, he took up a job with ISO, and had very little time left to spend on the book.

Getting feedback

The text, as it was at the end of 1992, was sent to several of our colleagues and friends in the LaTeX world, and they kindly spent part of their Christmas holidays reading the first complete draft of the book. At the same time Addison-Wesley had some chapters read by a few of their reviewers.

It is extremely important and helpful to have feedback at an early stage, not only to find possible mistakes, but also to receive comments and suggestions from other people, who can often shed an interesting new light on points which are taken for granted, or point out grey areas in style and explanation.

Design specification

In the meantime Frank was hard at work trying to translate the page specifications (for headings, figures, captions, running titles, etc.) as given by the Addison-Wesley designer into values of TeX glue, rules, boxes, and penalties. It was not always evident how to translate the fixed-space approach of the classical design specs into TeX's page-layout paradigm; so on various occasions "clarifications" had to be obtained.

Coding conventions

It was soon realized that it is extremely important to have a common way to generically mark up commands, environments, counters, packages, or any

other global distinctive document element. This not only ensures a homogeneous presentation throughout the book, but also allows one to change presentation form without modifying the input text; one merely has to change the definition of the generic command (a few examples are shown in table 1). As a supplementary benefit one can decide to globally (and automatically) enter certain of the marked-up entities in the index.

Setting up communication channels

In order to make communication between the production people in the Boston area and ourselves easier, it was decided to bring in the expertise of a production bureau, *Superscript*, with its competent manager Marsha Finley, and her colleague Anne Knight. Another decision was to have the complete text reviewed by a professional copy-editor.

Around Easter 1993 we thus started to copy PostScript files with ftp (quite a new procedure to all the production people involved, who were, at best, used to transporting 3.5 inch diskettes between their Macs or IBM PCs and the printing bureau). These files were then retrieved on the Sun in the Reading office of Addison-Wesley, printed locally, and picked up by Marsha, who took the pages to the copy-editor. The latter returned the edited copy to Marsha, several copies were made, and Frank and Michel both got a copy via Federal Express. The changes were then introduced into the text, by either Frank or Michel, and, whenever we had a problem, we would solve it via e-mail with Marsha.

And then came LaTeX2ε

While this iterative process was getting well under way, an unexpected event happened. Frank and Leslie Lamport, who was visiting Mainz in the spring of 1993, decided to consolidate LaTeX into a new version, LaTeX2ε, which would bring together the various dialects and formats floating around on the various networks and archives, and include the New Font Selection Scheme (NFSS) by default. It would also include a few limited extensions and propose a better style writer interface.

This very good news for the LaTeX community, however, meant for us that we were now describing and using an evolving software system. After the copy-editing stage, in several tens of places nontrivial changes had to be introduced in the text, new paragraphs written, and complete new sections added in some parts. Moreover, the *Companion* was typeset with the alpha release of the continuously-changing version of LaTeX2ε, thus giving us some surprises from time to time (of which one or two even ended up in the first printing of the book).

Getting ready to print

By the end of the summer we had included all the comments of the copy-editor into the compuscript, and done most of the updates for LaTeX2ε. We then went on to the proofreading stage, where, again with Marsha Finley acting as liaison, a proofreader reread all pages after "final typesetting", pointing out remaining typos or errors in cross-references.

Tuning LaTeX and hand work

While we were preparing the final run, we had to tune the LaTeX parameters extensively, in particular to allow for the huge number of floats we had to deal with, but also for finding suitable glue settings and penalties. There was also some hand tuning needed.

Table 2 shows the amount of hand-formatting we found necessary to produce the final copy of the book.

To flag all visual formatting clearly (so that it could easily be identified and removed in case the text needed changing), we never used the standard LaTeX commands directly. Instead we defined our own set of commands, often simply by saying, e.g., `\newcommand{\finalpagebreak}{\pagebreak}`.

The table divides the commands used into three groups. The first deals with changes to the length of the page: `\finallongpage` and `\finalshortpage` run a page long or short, respectively, by one `\baselineskip`. The command `\finalforcedpage` enlarges a given page and is therefore always followed by an explicit page break in the source. The second group contains the commands for "correcting" LaTeX's decision about when to start a new page, and the final group contains a single command for adding or subtracting tiny bits of vertical space to improve the visual appearance.

The average number of corrections made with commands from the first group is slightly over 20%, or one out of five double spreads, since we applied such a change always to pairs of pages. If you look at the chapters with a large percentage of corrections, you will find that they contain either very large inline examples or large tables that should stay within their respective sections.

Hard page breaks were inserted, on average, every tenth page, often in conjunction with a command from the first group. In most cases this was used to decrease the number of lines printed on the page.

Most uses of `\finalfixedskip` can be classified as "correcting shortcomings in the implementation of the design." With an average of about 16% this may seem high. But in fact such micro adjustments usually come in pairs, so this corresponds to approximately one correction every 12 pages.

Preparing the index

As already mentioned above, most of the important document elements were entered into the index in an automatic way by using generic mark-up to tag them. But that is not enough to have a good index, and indeed, we went over each page and asked ourselves which keywords should be entered into the index so as to direct the reader to the information on that page. In fact quite a few readers' comments that we received after the first printing had to do with suggestions for adding additional keywords into the index.

Production problems

Since we were working in different locations (Geneva and Mainz) on different workstations (Digital and Hewlett-Packard) and with mostly non-identical versions of LaTeX (Frank was "improving" LaTeX 2_ε continuously, whereas Michel was using a "frozen" version that got updated every now and then), small differences could appear in line and page breaks, leading on many occasions to a state of mini-panic, which had to be relieved by an exchange of one or more urgent e-mail messages, often sent well after midnight, when the other members of our families had already long gone to bed.

The first printing

After a final LaTeX run of our complete 560-page book, late on December 1st, Michel was able to copy the the resulting PostScript file, 9.5 Mbytes in size, in 26 self-contained pieces, by ftp from the CERN computer to Reading. This was necessary since the PostScript files had to be transferred on 1.44 Mbyte PC diskettes between the Sun at Addison-Wesley and the Varityper 4300P 1200 PostScript printer of the service bureau, where the camera-ready pages were produced.

Taking a break, or so we thought

So, we could spend a nice 1993 Christmas holiday, hoping that everything would go all right, and, fair enough, we received the first printed copy of our book just after the New Year. Soon our first readers started to send us comments and suggestions, and to point out problems of various kinds (printing, typos, unclear explanations).

In March Addison-Wesley informed us that we had to prepare an updated version of the book for a second printing at the beginning of May. We thus started to introduce the suggested corrections and improvements into the text, finally ending up with over 160 pages that we wanted to reprint (many of them containing only tiny changes, but also, owing to knock-on effects, sometimes several pages in a row had to be reproduced). We also took advantage of readers' comments to redo the complete index.

Conclusion

We hope we have been able to convey in this short article some of the excitement, fun and frustration people experience when trying to write a book.

We are well aware of the fact that those of you who have been involved in the production of books or large documents have come across several of these problems before. We nevertheless hope that by telling our "story" some of the lessons we learnt will be useful to some of you.

References

Goossens, Michel, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Reading Mass.: Addison-Wesley, 1994.

Lamport, Leslie. *LaTeX—A Document Preparation System—User's Guide and Reference Manual*. Reading Mass.: Addison-Wesley, 1986.

Acknowledgements

We would like to thank Geeti Granger and Gareth Suggett for their helpful comments and suggestions.

LaTeX command (control sequence) '\stop' should be input as \Lcs{stop} to produce the text and the reference, as \xLcs{stop} to produce only the reference and as \nxLcs{stop} to only typeset the command sequence in the text.

```
\newcommand{\Lcs}[1]{\mbox{\normalfont\ttfamily\bs#1}\xLcs{#1}}
\newcommand{\xLcs}[1]{\index{#1@{\ttfamily\protect\idxbs#1}}}
\newcommand{\nxLcs}[1]{\mbox{\normalfont\ttfamily\bs#1}}
```

\Lmcs makes a main index entry for places where one defines or really talks about a command.

```
\newcommand{\Lmcs}[1]{\mbox{\normalfont\ttfamily\bs#1}\xLmcs{#1}}
\newcommand{\xLmcs}[1]{\index{"#1@{\ttfamily\protect\idxbs"#1}|idxbf}}
```

The \Lcsextra command is for producing a subentry for a command name.

```
\newcommand{\Lcsextra}[1]{\mbox{\normalfont\ttfamily\bs#1}\xLcsextra{#1}}
\newcommand{\xLcsextra}[2]{\index{#1@{\ttfamily\protect\idxbs#1}!#2}}
\newcommand{\Lmcsextra}[1]{\mbox{\normalfont\ttfamily\bs#1}\xLmcsextra{#1}}
\newcommand{\xLmcsextra}[2]{\index{#1@{\ttfamily\protect\idxbs#1}!#2|idxbf}}
```

For flagging a range of pages covered by a definition, we use the "rangel" (start of range), and "ranger" (end of range) construct.

```
\newcommand{\xLcsextrarangel}[2]{\index{"#1@{\ttfamily\protect\idxbs"#1}!#2|(}}
\newcommand{\xLcsextraranger}[2]{\index{"#1@{\ttfamily\protect\idxbs"#1}!#2|)}}
```

Table 1: Examples of generic tags used to reference command sequences

Chapter	2	3	4	5	6	7	8	9	10	11	12	13	14	A1
Number of pages	36	36	18	40	16	58	44	16	36	36	26	50	18	36
\finallongpage	0	3	1	0	3	10	4	2	3	0	4	9	7	4
\finalshortpage	0	5	4	4	0	2	10	0	0	8	6	0	0	2
\finalforcedpage	1	0	0	2	2	0	1	0	0	1	0	1	0	0
Page length change	1	8	5	6	5	12	15	2	3	9	10	10	7	6
Average per page	.03	.22	.29	.15	.33	.08	.34	.13	.08	.25	.38	.2	.39	.17
\finalpagebreak	4	5	2	4	3	7	12	1	0	6	4	5	3	6
\finalnewpage	0	1	0	0	0	0	0	0	0	0	0	1	0	0
Page break change	4	6	2	4	3	7	12	1	0	6	4	6	3	6
Average per page	.11	.17	.11	.1	.19	.12	.27	.06	0	.17	.15	.12	.17	.17
\finalfixedskip	4	3	4	11	0	8	2	2	0	14	6	10	7	3
Average per page	.11	.08	.22	.28	0	.14	.05	.13	0	.39	.23	.2	.38	.08
Sum	9	17	11	21	8	27	29	5	3	29	20	26	17	15
Average per page	.25	.47	.61	.53	.5	.47	.66	.31	.08	.81	.77	.52	.94	.42

Table 2: Manual work—some numbers (from Goossens, Mittelbach and Samarin (1994))

Typesetting the Holy Bible in Hebrew, with TEX

Yannis Haralambous
Centre d'Études et de Recherche sur le Traitement Automatique des Langues
Institut National des Langues et Civilisations Orientales, Paris.
Private address: 187, rue Nationale, 59800 Lille, France.
Yannis.Haralambous@univ-lille1.fr

Abstract

This paper presents *Tiqwah*, a typesetting system for Biblical Hebrew, that uses the combined efforts of TEX, METAFONT and GNU Flex. The author describes its use and its features, discusses issues relevant to the design of fonts and placement of floating diacritics, and gives a list of rare cases and typographical curiosa which can be found in the Bible. The paper concludes with an example of Hebrew Biblical text (the beginning of the book of Genesis) typeset by *Tiqwah* .

Introduction

The *Tiqwah* system uses the possibilities of TEX, METAFONT and GNU Flex to typeset Biblical Hebrew. This is not a simple task: (a) special fonts had to be created, described in the section 'Fonts for typesetting the Holy Bible in Hebrew' on page 177; (b) several levels of diacritics are required; they have to be entered in a reasonable way (see 'Vowels' on page 176, and 'Masoretic accents and other symbols' on page 176), and placed correctly under or over the characters (see 'An algorithm for placing floating diacritics' on page 179). The Bible being the most demanding Hebrew text (from the typographical point of view), *Tiqwah* can trivially be used to typeset any other Hebrew text, classical or modern; in addition to Tiberian vowels, Babylonian and Palestinian vowels are also included in the font, as are special characters for Yiddish.

This paper is divided into three parts: the first one, more pragmatic, describes the requirements and use of the *Tiqwah* system; the second one discusses the design of the fonts and the algorithm of floating diacritics placement; finally, the third part gives a list of rare cases and typographical curiosa found in the Hebrew Bible, and the way to produce them through *Tiqwah*.

But first, for the reader not familiar with the Hebrew language, a short introduction to the Hebrew system of diacritization.

Diacritization. In Hebrew, as in other Semitic languages, only consonants and long vowels are written as letters: the reading process includes a permanent "guessing" of words out of the available data—the consonants and long vowels, as well as the grammatical, syntactic and semantic context.[1] To prevent misunderstandings, in cases where the short vowels cannot be guessed out of the context (for example in names or foreign words), or in cases where the text is extremely important and should by no means be altered (the case of holy texts, like the Bible), short vowels have been added, in the form of diacritics. This is the first level of diacritization; it can be applied to any text; at school, children first learn vowelized Hebrew.

A second degree of diacritization is the use of *cantillation marks* or *Masoretic marks* or *neumes.*[2] This method of diacritization applies only to the Hebrew Bible.

Finally, a third degree of diacritization and markup (less important in volume than the two previous ones) consists of using editorial marks for scholarly editions (locations where text is missing, diverging sources, etc.). For this purpose, mainly two signs are used: the *circellus* (a small circle) and the *asterisk.* Also a dot is sometimes placed over each letter of a word—it is called *punctum extraordinarium.*

[1] tr t rd ths t s wht I mn → try to read this to see what I mean.

[2] One reads in Levine (1988, pp. 36-37): "...unlike Psalmodic technique which reserves its motifs for a single syllable toward the phrase-end, Biblical chant assigns a motif to each word. It does this with signs called *neumes* (*te'amim* in Hebrew). ...The root of "neume" in Hebrew, *ta'am* has several meanings: 'taste'; 'accent'; 'sense'. Neumes impart taste (intonation) to Scripture through melody, accent through placement (above or below the stressed syllable), and sense (rhetoric) by their ability to create a pause or to run words together. In addition to these functions, neumes provide a means of memorizing the intonation, accentuation, and rhetoric of the handwritten scrolls read publicly, for only consonants appear on the scrolls. Vowels and punctuation—as well as neumes—appear only in *printed* editions of the Hebrew Bible."

It follows that printed Hebrew Biblical text can globally be subdivided into four strata:

4. Editorial marks
3. Cantillation marks
2. Vowels, semi-vowels and šewa
1. Text

The placement of diacritics falls into the following groups:

1. inside the letter: the *dageš* or *mappiq* dot;
2. over the letter: vowels (*ḥolem* in the Tiberian system of vowelization, and all Palestinian and Babylonian vowels), spirantization (*rafe*), cantillation marks (*zaqeph, rebia, gereš, garšayim*, etc.), editorial marks (*circellus, asterisk, punctum extraordinarium*);
3. under the letter: vowels (*ḥireq, ṣere, segol*, etc.), semi-vowels (*ḥateph-pataḥ*, etc.), absence of vowel (*šewa*), cantillation marks (*silluq, atnaḥ*, etc.);
4. before the letter (on its right): prepositive cantillation marks (*dehi, yetib*, etc.);
5. after the letter (on its left): postpositive cantillation marks (*segolta, sinnor*, etc.).

All strata of diacritics can be combined. It has always been a typesetter's nightmare (or delight, depending on the case) to produce fully diacriticized Hebrew text: sometimes the combinations of diacritics get even wider than the character that carries them; in these cases, diacritics will *float* under (or over) the immediately following letter, according to rules given in the section 'An algorithm for placing floating diacritics' on page 179. These actions can eventually change the appearance of the whole word. In that section we give an analytic approach of floating diacritic placement, and the corresponding algorithm used by *Tiqwah*'s TEX macros.

The reader can find more information on the grammar of Biblical Hebrew in Lettinga (1980); for an introduction to the modern edition of the Bible BHS (Biblia Hebraica Stuttgartensia), see Wonneberger (1990).

Using *Tiqwah*

Requirements. To typeset in Biblical Hebrew using *Tiqwah*, one needs a decent TEX system,[3] a relatively powerful machine (being able to run BigTEX) and the *Tiqwah* package,[4] consisting of a preprocessor written in GNU Flex, fonts written in METAFONT, and TEX macros. The preprocessor being written entirely in GNU Flex (without using any system-dependent subroutines), can be compiled in a straightforward manner on any platform having a GNU Flex executable and an ANSI C (preferably gcc) compiler.

Once *Tiqwah* has been installed, typesetting is done in two steps: an input file is prepared using the syntax described below; the preprocessor then reads this file, and produces a LATEX2ε (or plain TEX) file which can then be run through TEX in the usual way.[5]

Preparing the input file. If you wish to write your file in LATEX2ε, you have to include the line

```
\usepackage{tiqwah}
```

in the preamble. Plain TEX users will write

```
\input tiqwamac.tex
```

at the beginning. However, the author recommends the use of LATEX2ε, because of its powerful font selection scheme.

A *Tiqwah* input file contains text, TEX/LATEX macros, and *preprocessor directives*. The latter concern only Hebrew script. To type Hebrew text you need to enter *Hebrew mode*; this is done by the preprocessor directive <H>. To leave Hebrew mode, enter the directive </H>. For Yiddish, the directives are <Y> and </Y>. The directives <H> and <Y> are the only ones recognized by the preprocessor *outside* Hebrew/Yiddish mode.

Once you are inside Hebrew/Yiddish mode, you type Hebrew text in Latin transcription, from left to right. No special indication needs to be given to TEX about font or writing direction switching—this is done automatically. The following sections describe the transcription you have to use as well as all other features of the preprocessor.

Letters. The Hebrew transcription of letters (consonants and long vowels) is given in Table 2 of the appendix (page 187); the Yiddish one will be given together with all other features of the Yiddish part of *Tiqwah*, in a forthcoming paper, dedicated entirely to this language.

Here is a simple example of code producing non-vowelized Hebrew text:

[3] In this context, by 'decent TEX system' we mean a TEX implementation featuring Peter Breitenlohner's TEX--XET as well as a METAFONT implementation with user-configurable parameters (the internal parameter `max_font_dimen` of METAFONT has to take a value of at least 53, to be able to generate *Tiqwah* fonts).

[4] *Tiqwah* will be included in ScholarTEX; it is part of the long awaited version 1 of the latter, together with *new* Greek, Arabic, Estrangello, Serto, Chaldean, Coptic and Akkadian cuneiform fonts.

[5] An adaptation of the *Tiqwah* system to Ω (the TEX extension prepared by John Plaice and the author) is under preparation; it will allow typesetting in Biblical Hebrew, without a preprocessor.

<H>ym hm*lx hw*' hm*qwm hn*mw*k b*ywtr b*'wlm</H> will produce

ים המלח הוא המקום הנמוך ביותר בעולם.

Some notes concerning the transcription of letters of Table 2: there is no distinction between medial and final forms; these are automatically applied by TEX. The asterisk * transcribes the *dageš*, *mappiq* or *šureq* dot.[6] The broken lamed ל is used automatically whenever no upper diacritic is present;[7] this feature can be turned off by the command line option -nobroken of the preprocessor. The character ﭏ is a ligature of the letters *aleph* and *lamed*; a variant form of it is ﭏ. This ligature is not used in the Bible, and hence is not applied automatically by the preprocessor: it has to be explicitly requested by the code '/l (instead of 'l which will produce the normal אל).

In the same table, the reader will also encounter the symbol ׆; it is called "*nun invers*" and is used in Nu 10:35-36 and Ps 107. The "broken waw" ו is used in Nu 25:12. See the section 'Inverted and broken letters' for more details.

Vowels. Hebrew vowels and their transcriptions are displayed in Tables C and D (p. 188). Table 3 displays the three systems of vowelization available: Tiberian (the most frequent one), Palestinian and Babylonian. Tiberian vowels are used by default. To switch to Palestinian or Babylonian, one uses the directives <PALESTINIAN> and <BABYLONIAN>. The directive for Tiberian is <TIBERIAN>. The same text can be typeset in any one of the three systems just by adding/removing one of these directives; here is an example of the same text, written in the three vowel systems:

ימלא פי תהלתך היום תפארתך
ימלא פי תהלתך היום תפארתך
ימלא פי תהלתך היום תפארתך

Most vowels can be entered in two different ways: either by a "phonetic" one- (or two-) letter code (a for *pataḥ*, A for *qameṣ*, etc.) or by a three-letter code in uppercase form, surrounded by < and > (<PAT> for *pataḥ*, <QAM> for *qameṣ*, etc.). Both methods are equivalent and can be arbitrarily mixed.

Vowels are entered *after* letters, except in the case of the *pataḥ furtivum*, where the code <PTF> has to be entered before x, h*, or ' (ח, ה and ע are the only letters which can take a *pataḥ furtivum*[8]). The *rafe* accent can be found in table 5.

Below is the same example of simple Hebrew text with its transcription, this time vowelized:

```
<H>yAm ham*Elax hw*' ham*Aqw^om han*Amw*k"

b*"yw^oter b*A'w^olAm</H>
```

will produce

יָם הַמֶּלַח הוּא הַמָּקוֹם הַנָּמוּךְ בְּיוֹתֵר בָּעוֹלָם

Masoretic accents and other symbols. Tables E and F (p. 189, 190) display Masoretic cantillation marks and miscellaneous symbols: the Sephardic *varika*, and punctuation marks *maqqeph*, *setuma*, *petuḥa*, *soph pasuq*. Two styles of Masoretic accents are provided: oldstyle (as found in BHK[9] and Holzhausen Bible (1889), Lowe and Brydone Bible (1948)) and modern (as in BHS[9]). The distinction is made at the TEX level, by macros (\modernmasoretic and \oldstylemasoretic), which can be used inside or outside Hebrew mode; the default style is oldstyle. Table 6 shows the glyphs of modern Masoretic accents. The same remark as in the previous section, concerning alternative input of codes, applies in this case also.

Masoretic accents are entered after the letter to which they belong; they can be placed before or after vowels belonging to the same letter—their order is not important. Prepositive accents are placed before the first letter of the word. Postpositive accents, such as *pašṭa*, placed *inside* a word, will be typeset *between* letters.

Finally, Table 7 (page 190) displays a collection of typographical curiosa: symbols used in various contexts and for various purposes. The single and double primes ′ and ″ are used for numerals and abbreviations. The upper two dots diacritical mark is also used for numerals: it indicates thousands. The asterisk * is used both as an editorial mark (like the circellus, but apparently with slightly different meaning), and as a replacement character for missing letters (see the section 'Missing letters' on page 183). The zero-like symbol **0** is used to indicate a missing word in Jdc 20:13 (Holzhausen Bible (1889), Lowe and Brydone Bible (1948) only). The isolated *dageš* is used to indicate a missing letter with *dageš*, in Jes 54:16 (BHS only). The "tetragrammaton" ה׳ is a symbol for the name of God; it can be obtained by the directives <YYY> or <TETRAGRAMMATON>. The dotted circle ◌ is used in textbooks as a basis for diacritics.

Other preprocessor directives. A few directives do not produce glyphs, and hence are not included in the tables:

[6] Following advice by Philippe Cassuto, we will attempt to differentiate the *dageš* and the *šureq* applied to the letter *waw*, in the next version of *Tiqwah*.

[7] With one exception: the *ḥolem*.

[8] The combination "letter *'ayin* with *pataḥ furtivum*" is not displayed in the table because it is graphically indistinguishable from the normal *'ayin* with *pataḥ*. This can be changed if there is a demand for differentiation of the two *pataḥ* types.

[9] Throughout this paper, BHS will be the *Biblia Hebraica Stuttgartensia* BHS (1987), and BHK the *Biblia Hebraica* BHK (1925), edited by Rudolf Kittel.

1. <NIL> placed after a letter will prevent the final form to be applied to it. For example, for numerals or stand-alone letters which have to be in medial form:
   <H>k<NIL> k</H> will produce כ ך.
2. <EOW> placed after a letter will force it to be in final form, even if other letters follow. For example, in Jes 9:6 one reads לְםַרְבֵּה; this word has been entered as
   <H>l"ma<EOW>r"b*e<AZL>h</H>). See the section 'Letters not obeying rules of contextual analysis' on page 182 for more details.
3. <EMPTY> will produce an invisible character of normal width. It can be used as a basis for stand-alone diacritics in the case of missing words (see 'Missing words' on page 183).
4. <SMALL> and <BIG> will produce small and big letters, see section 'Bigger and smaller letters' on page 181 for more details; they act only on one letter at a time.

More features may be added to the preprocessor if necessary.

Running the preprocessor. Once you have prepared the input file, for example genesis.inp, you run the preprocessor by writing
tiqwah *options* < genesis.inp > genesis.tex
where *options* can be the following:

1. -h displays a few lines describing the command line options;
2. -p produces plain TEX instead of LATEX output (typesetting with *Tiqwah* in plain TEX is not recommended);
3. -l followed by a number, indicates the maximum line length of code produced by the preprocessor; default is 80. This applies only to commands inside Hebrew/Yiddish mode, the remainder of the file is not modified;
4. -nobroken disables the automatic broken *lamed* insertion. With this option,
   <H>w"l<SIL>'^o=yAla<RBM>d"t*iy</H> will produce וְלֹא־יָלַדְתִּי Jes 23:4; without it, you would get וְלֹא־יָלַדְתִּי. It should be noted that the *ḥolem* vowel fits on the broken *lamed*: a special "broken-lamed-with-ḥolem" glyph is provided in the font (לֹ);
5. -d produces debugging output sent to the stderr stream, for those who want to modify the code of the preprocessor.

Running TEX/LATEX. As usual, TEX has to be TEX--XET, otherwise you will get an error message about the unknown commands \beginR and \endR.

If you are using LATEX2ε, you have to include the line \usepackage{tiqwah} in the preamble; if you are using plain TEX (not recommended), write \input tiqwamac.tex instead.

We have completed the description of the preprocessor's use and features. Now we will turn ourselves to issues concerning the design of fonts and the placement of floating diacritics.

Fonts for typesetting the Holy Bible in Hebrew

Designing fonts for Biblical typesetting is quite a challenge: on the one hand, one has to face centuries of tradition, and the inevitable comparison with masterpieces of typography; on the other hand, unlike Western typography, there is no room for innovation: modern Hebrew typefaces are widely used in Israel and elsewhere, but certainly *not* for Biblical text! Working under such tight restrictions can be compared to composing fugues or painting Byzantine icons: there are very strict rules to struggle with, and you can't avoid being hooked by the masterpieces others have done and which fatally are out of reach...

Fortunately, digital font creation does not always need to be original and innovative (although at the end it always *will* have new features, since the phototypesetting machines are fundamentally different from the traditional presses). After all, we are in the age of *reproduction*...

The author started with the idea in mind to reproduce as faithfully as possible the most beautiful Hebrew font he could find. There seems to be a consensus among a large group of scholars that one of the most beautiful Hebrew types ever done was the one of the *Biblia Hebraica*, edited by Kittel and printed in Germany in the early twenties. Unfortunately the molds were lost in the bombing of Leipzig, so only printed copies of that book could be studied by the author to get the necessary information for reproducing the font.

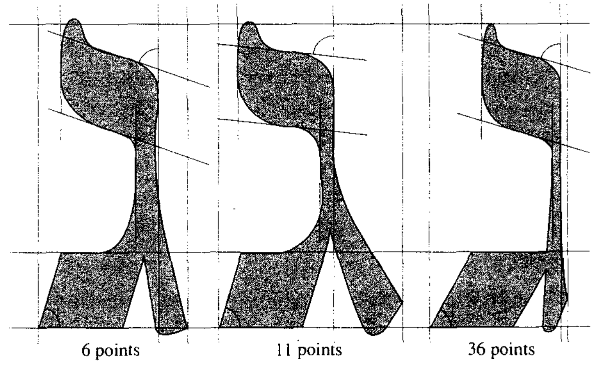


Figure 1: The letter ג at point sizes 6, 11, 36

Doing this, and studying other books as well, such as a Haggadah by Saul Raskin (Raskin 1941), printed in New York in 1941, and old Talmudic books

printed in Vienna in the late XIXth century, the author realized that Hebrew fonts have a fascinating feature: there is a remarkable deviation between different point sizes (in TEXnical terms: they have a high degree of metaness). While for Latin typefaces the changes between small and large point sizes affect mostly the width of strokes, in Hebrew, letter shapes often change considerably. And what's even more unusual: changes that occur when going from small sizes to the normal size often occur the other way around when going from normal to large size: for example, one can see, in Fig. 1, the letter *gimel* at 6, 11, and 36 points (magnified so that they all have the same physical size). While the right tail of the letter moves more to the right when going from 6 to 11 points, it retracts again when going from 11 to 36, and almost becomes vertical.

Here is a (possible) explanation for this behaviour: the reasons for metaness in the small-to-normal range are different than those in the normal-to-large range. In the former case, the problem to solve is legibility. As a matter of fact, many Hebrew letters look quite similar in normal size: compare *samekh* and final *mem*, or *kaph* and *bet*, at 11 (or higher) points in the Table of Appendix A. Their distinctive features are so discrete that they could well disappear if the normal size was reduced linearly; a well-drawn small point Hebrew font has to bring these distinctive features to the foreground. Compare these letters again at 6 points: *kaph* and final *mem* are round while *samekh* and *bet* remain quadratic. On the other hand, when going from normal to large, one follows purely esthetic criteria: elegance is the main goal. In this context, Hebrew letters follow "Bodoni-like" esthetics: they have very important fat strokes and very fine thin ones. Hebrew letters use —even more than Latin letters— the effect of contrast between fat and thin strokes.

Being hooked by the beauty of this script the author decided not only to produce a most decent Hebrew font, but also to cover the whole range of optical METAmorphoses of the types. On table 1 of the appendix (page 186), the reader can see the first results of this adventure; they are by no means final! The author hopes to be able to improve these characters to meet the level of the Hebrew typographical tradition.

Technical details. Drawing a font with such a high degree of metaness is a process not far from *morphing*, a technique used more and more in video and cinema.[10] Nevertheless there is an important difference between METAFONT "morphing" and the usual morphing we see in movies. To morph two images, we are not changing the grayscale (or color) weight of each pixel, but the coordinates of Bézier curve control points. Interpolation becomes very uncertain, since it is by no means trivial that the set of interpolated Bézier curves will still produce a decent character shape.

The solution to this problem is to detect "tendencies" in the letter shape metaness and to be guided by these while morphing: for example, the lower left stroke of the letter *aleph* has the tendency of protruding to the left when point sizes become small. This has to be taken into account for all paths of this stroke, so that the transformation is homogeneous. The best way to do this is to determine "centers of gravity" which will move during the transformation; then it suffices to define all the important control points of the stroke with respect to a center of gravity: in this way the movement of the latter will produce an homogeneous move (and hence, transformation) of the whole stroke.

An important precaution is to limit the metaness of certain quantities to a certain range of point sizes. For example, the width of fat strokes can vary arbitrarily (after all, it is directly related to the letter point size), but other characteristics should not "vary too much"; in other words, they should remain stable outside of a certain point size range. That is the case, for example, of the "hanging left stroke" of letter final *pe*, in small point sizes; this stroke extrudes already to the left at point size 8; for point sizes lower than 8, the amount of extrusion remains stable, otherwise the character shape would be deformed; same phenomenon for the height of the intersection point of the vertical and the oblique stroke of letter final *ṣade*: after point size 24 the intersection height remains stable, since at this point size it has reached an extremal point. The idea of this paragraph could be stated as: "morphing should be applied only for interpolations inside the regular range; for extrapolations, the usual metaness (stroke widths, etc.) is applied."

One of the most important parts of many Hebrew letters is the "flame" (or "crown"). Figure 2 shows the different METAFONT reference points and paths used for the definition of a standard METAFONT "flame"-subroutine.

Rashi. Besides the "quadratic" Hebrew font, which is shown in table 1 of the Appendix, the author has also developped a *Rashi* font. This type was used in Synagogue books for comments on the Biblical text. Synagogal books, which are often masterpieces of typography, combine several point sizes of *Rashi* and *quadratic* in various page setups. On the other hand, Rashi is not used in scholarly editions. Rashi is not diacriticized (neither vowels, nor cantillation marks);

[10] *Morphing* is the continuous interpolation between two pictures; it has been used in special effects, for example to show faces being transformed into other faces.

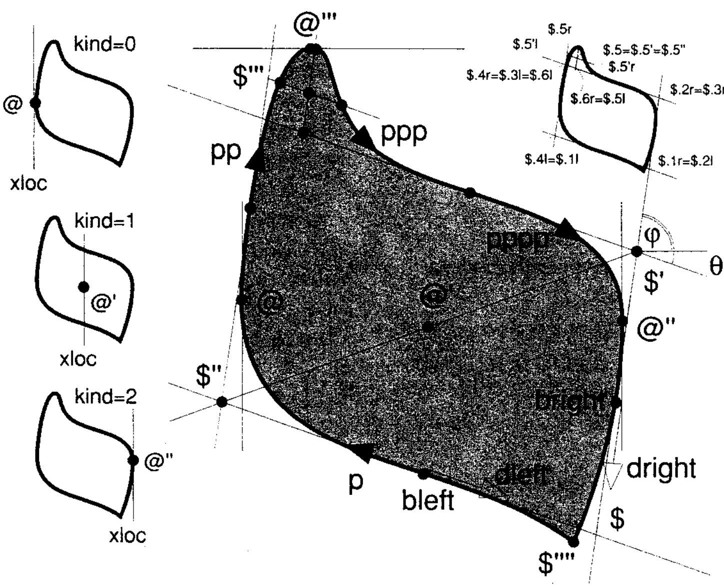


Figure 2: The METAFONT description of the character part "flame"

nevertheless, the author has included vowels, semi-vowels and šewa in the Rashi font, just in case somebody wants to break that rule...

The Rashi font has only the usual metaness (optical scaling through variation of stroke width). Here are the characters of point size 10, in their real physical size, compared with the quadratic font at the same size:

Rashi: אבגדהוזחטיכךלמםנןסעפףצץקרשת

Quadratic: אבגדהוזחטיכךלמםנןסעפףצץקרשת

In a forthcoming article, the author will give examples taken from Synagogal books with comments in Rashi.

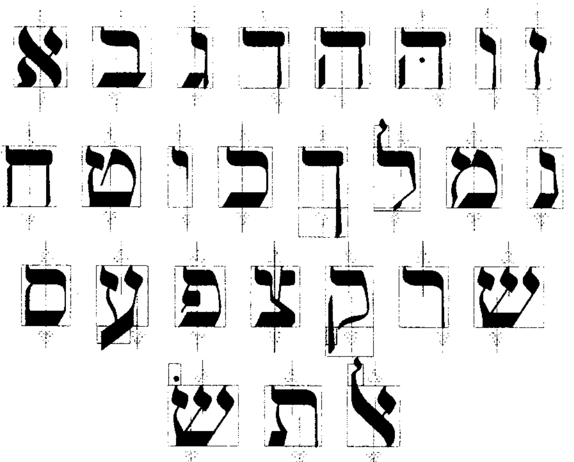


Figure 3: Upper and lower symmetry axes for Hebrew characters

An algorithm for placing floating diacritics. After a close study of the typesetting of diacriticized text in the Hebrew Bibles, and numerous discussions with Johannes de Moor, the author was led to the following considerations:

1. We divide the set of diacritics into two categories: *primary* and *secondary*. Primary diacritics are vowels, semi-vowels and šewa (stratum 2); all others are secondary ones (strata 3–4). Secondary diacritics are always appended to the left of primary diacritics belonging to the same letter.
2. Every consonant has an upper and a lower symmetry axis, on which diacritics are centered: these are not necessarily identified with the symmetry axis of the character's box: for example, the lower symmetry axis of character ך is going through the middle of the *vertical stroke*, and not through the middle of the imaginary character box (as its upper symmetry axis).

   In Fig. 3, the reader can see the choice of upper and lower symmetry axes for each character, as well as the "forbidden zones", which should be avoided by diacritics[11].
3. Suppose that:
   - we have a letter L followed by letter L';

[11] The reader will notice two letters *he* in Fig. 3, with different lower symmetry axes: the second one shows the axis used for the *pataḥ furtivum*. The same process is applied to the letter *ḥet*.

- letter L carries a primary diacritic P and a secondary diacritic S, both under it, S being necessarily appended to the left of P;
- the lower symmetry axis of letter L is A and the one of L' is A'.

Then we have *three possible choices, in the following order of preference*:

(a) P is centered on A (see Fig. 4, 1);
(b) the group of diacritics SP is centered on A as a whole (see fig. 4, 2);
(c) the group of diacritics SP is centered on A as a whole, furthermore a kern is added between L' and L (see fig. 4, 3) so that diacritic S does not overlap on L' or its diacritics.

Once these choices as well as their order of preference have been determined, the algorithm for placing diacritics under (or over) a word is the following:

```
for (letters of the word starting from the left)
{
    try choice (a)
    if ((a) not successful) {
        try choice (b)
        if ((b) not successful) {
            apply choice (c)
            }
        }
    go to next letter
    }
```

where the criterium of "success" is the fact that the diacritics of the current letter do not overlap with the following letter (if this letter has a descender part) or its diacritics, or its lower symmetry axis. Here is an example of such a situation. The reader can see a few (imaginary) words illustrating the three choices described above:[12]

בבֶ֑ז (2) בבֶ֑ב (1)

בבֶ֑׃ (4) בבֶ֑זֶ (3)

In case (1), we have three letters *bet*, the medial one having a primary diacritic *segol* and a secondary diacritic *atnaḥ*. On the left side there is no diacritic, and the lower symmetry axis of the left *bet* is far enough from the *atnaḥ* of the medial letter to allow placement of the diacritics according to choice 1: the *segol* is centered under the letter, and the *atnaḥ* concatenated to it.

In case (2), instead of *bet* we have placed a *zayin* at the end of the word. There is no diacritic under that letter, but its lower symmetry axis is much closer to the medial *bet* than it was in case (1), so that now, the diacritics of the medial letter, placed as before, would inevitably touch the symmetry axis of the *zayin*. TEX automatically switches to choice 2, and checks that, without additional kerning, the diacritics remain indeed inside the authorized area.

In case (3), we add a diacritic *segol* to the letter *zayin*. Choice 2 is not valid anymore, and TEX automatically kerns letters *zayin* and the *bet* so that the *atnaḥ* is at a safe distance from the *segol* to the left of it. This is choice 3, and it always works, because there are no limits set on TEX's operation of kerning.

Word (4) has been included to show TEX's reaction in front of a punctuation mark: (a) TEX does not float the diacritic under the punctuation mark as in case (1), and (b) it does not switch either for choice 2, like in (2). The reason is that both operations (a) and (b) are reserved for letters which are considered as part of a whole (the *word*); the punctuation mark belonging to a different entity must be placed independently, and should not participate in the algorithm of floating diacritic placement. As the reader can see in (4), TEX kerns between the punctuation mark and the letter until the *atnaḥ* is clearly not under it anymore.

[12] These words are displayed in a magnified 8-point font, so that diacritics are larger, relative to characters, and the three choices become more obvious.

NOTES:

1. Certain characters have descenders: ק ך ן ף ץ or ascenders: ל; these parts of characters are considered "forbidden zones"—no diacritic should overlap or even touch them (forbidden zones are visible on fig. 3 as shaded areas).
2. The algorithm only concerns diacritics that are *centered* over or under the character with respect to the symmetry axes shown in Fig. 3; uncentered diacritics (like the *ḥolem*) obtain fixed positions before applying the algorithm. The region they occupy becomes a *forbidden zone*, just like letter descenders or ascenders.[13]
3. If there are both upper and lower diacritics, the algorithm has to be applied twice, once for each case. Choices are independent, but a possible kerning due to application of choice 3 to one of the two parts could modify the choice applied to the other part.
4. If there is already a kern between two letters, it must be taken into account before applying the algorithm.
5. While inside a line, TEX is typesetting by counting blank space with respect to character boxes (and not diacritic boxes), at the beginning of a line the maximum between the width of diacritic box and the width of character box must

[13] An exception to this rule is the letter וֹ (*waw* with *ḥolem magnum*), where the right dot is sufficiently below the standard diacritic height for additional diacritics to be placed as if the dot was not there.

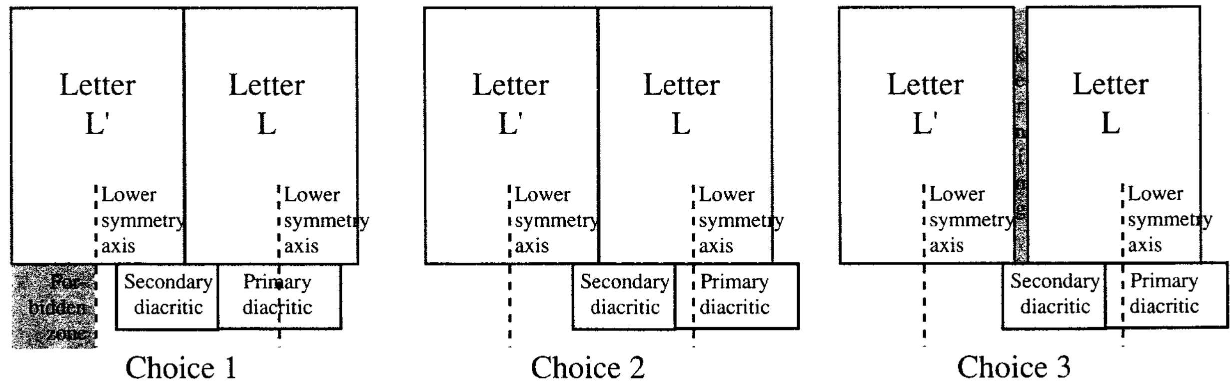


Figure 4: Three possible choices of diacritics positioning, in order of preference.

be taken into account (so that diacritics do not protrude over the beginning of a line).

Rare cases and typographical curiosa

It is forbidden—and has always been forbidden—to change the text of the Hebrew Bible. The Masorets and other Bible commentors have proposed some modifications to the text, which had to be made apparent without changing the text itself. For this reason, many (typo-)graphical tricks have been used to indicate potential modifications of the text. These may differ from one Bible edition to the other (although they seem to be quite stable between rabbinical editions), and may not appear in modern study editions of the Bible, like the BHS. Here is a list of such curiosa, after a short search by the author,[14] as well as the way to achieve them with *Tiqwah*.

Bigger and smaller letters. These are letters bigger or smaller than ordinary text. They can appear at any location inside a word. They are vertically justified at the upper bar of Hebrew letters (and not at the baseline), so that big letters are protruding downwards only, and small letters are "hanging". The eventual *dageš* dot belongs to the point size of the letter itself (bigger or smaller than ordinary text), while the eventual diacritics are typeset in the same size as ordinary text. In the case of big letters, lower diacritics are lowered so that they keep the same distance to the letter as in the case of ordinary letters; in the case of small letters they are *not* raised, and remain at their default position.

Here are all possible occurrences the author could detect:

בְּרֵאשִׁית Gn 1:1,
בְּהִבָּרְאָם Gn 2:4,
וְלִבְכֹּתָהּ Gn 23:2,
קַצְתִּי Gn 27:46,
וּבְהַעֲטִיף Gn 30:42,
הַכְזוֹנָה Gn 34:31,
שִׁלֵּשִׁים Gn 50:23,
נֹצֵר Ex 34:7,
אַחֵר Ex 34:14,
וַיִּקְרָא Lv 1:1,
מוֹקְדָה Lv 6:2,
עַל־גָּחוֹן Lv 11:42 [big *waw*],
וְהִתְגַּלָּח Lv 13:33,
יִגְדַּל־נָא Nu 14:17 [big *yod*],
שְׁמַע ... אֶחָד Dt 6:4,
מַמְרִים Dt 9:24,
וַיַּשְׁלִכֵם Dt 29:28,
הַלְיהוָה Dt 32:6,
כְּכֹחִי Jos 14:11,
צִפּוֹ Jes 56:1,
וְצִוְחָת Jer 14:2,
בְּסוּפָה Na 1:4,
לַשָּׁוְא Ps 24:4 [small *waw*],
וְכַנָּה Ps 80:16,
מִשְׁלֵי Prv 1:1 [small final *nun*],
וְנִרְגָּן Prv 16:28,
אָדָם Prv 28:17,
הַב | הַב Prv 30:15,
וְגִישׁ Hi 7:6,
שִׁבְטוֹ Hi 9:34,
עַל־פְּנֵי־פָרֶץ Hi 16:14 [small final *ṣade*],
חַף Hi 33:9,

[14] The author would be grateful for any help or suggestions on completing this list.

שִׁיר Cant 1:1,
לִינִי Ru 3:13,
לוֹא Thr 1:12,
טָבְעוּ Thr 2:9,
טוֹב Qoh 7:1,
סוֹף Qoh 12:13,
חוּר Est 1:6,
פַּרְשַׁנְדָּתָא Est 9:7,
פַּרְמַשְׁתָּא Est 9:9,
וַיְזָתָא Est 9:9 [big *waw* and small *zayin*],
וַתִּכְתֹּב Est 9:29,
בִּשְׁפַרְפָּרָא Da 6:20,
אָדָם 1Ch 1:1.

To produce big and small letters, one uses the preprocessor directives <BIG> and <SMALL> respectively. These affect only the first letter following them, e.g., to obtain בִּשְׁפַרְפָּרָא Da 6:20, one writes <H>b*iS"<SMALL>par"<BIG>p*ArA<TIP>'</H>.

Raised letters. At three locations in the Bible, the author encountered raised *'ayin* letters, and at one location, a raised *nun*. Contrary to small letters as described in the previous section, these are typeset in the regular point size. The diacritics remain under the normal baseline except in the case of a *pataḥ* diacritic, which was raised as well, in BHS and BHK.

Here are all occurrences of raised *'ayin* the author could find:

מִיָּעַר [in Holzhausen Bible (1889), Lowe and Brydone Bible (1948)] or
מִיַּעַר [in BHS and BHK] Ps 80:14,
רְשָׁעִים Hi 38:13,
מֵרְשָׁעִים Hi 38:15.

The raised *nun* was encountered in
בֶּן־מְנַשֶּׁה Jdc 18:30.

Both raised letters are regular characters of the *Tiqwah* font. The raised *'ayin* can be produced by the input code '/ ('/a in the case of raised *'ayin* with *pataḥ*). The raised *nun* with *pataḥ* can be obtained by the input code n//a (n followed by a single slash n/ produces the inverted *nun*, see section 'Inverted and broken letters').

Letters *aleph*, *resh* and *'ayin* with *dageš* dot. The author has found three locations in the Hebrew Bible, where the letter *aleph* takes a *dageš* dot: in BHS the dots are placed in the lower part of the letter; in BHK they are ignored; while in Holzhausen Bible (1889), Lowe and Brydone Bible (1948) they are placed in the upper or in the lower part of the letter. Here are these occurrences, as they appear in Holzhausen Bible (1889), Lowe and Brydone Bible (1948):

וַיָּבִאּוּ Gn 43:26,
תָּבִיאּוּ Lv 23:17,
וַיָּבִיאּוּ Esr 8:18.

At a single location in the Bible the author found the letter *'ayin* with *dageš*: בַּעְּפֹלִים 1S 5:12. This letter appears in Holzhausen Bible (1889), Lowe and Brydone Bible (1948) but not in BHS. In BHK a large dot is placed *over* the character.

Finally, the letter *resh* with *dageš* occurs in מָרַּת Prv 14:10.

To produce these letters with *Tiqwah*, use codes '*, '/, r* and '* as shown in table 2 of the appendix.

Letters not obeying rules of contextual analysis. In some cases a letter does not appear in final form as it should, and conversely a letter inside a word is written in final form (for example to indicate a contraction of two words). Here are two cases the author has detected: לְםַרְבֵּה Jes 9:6, with a final *mem* inside the word, and מִנ הַסְּעָרָה Hi 38:1, where the *nun* of the first word is not in final form.

To impose a final form one uses the preprocessor directive <EOW> (EOW stands for "end of word"), after the letter: <H>l"ma<EOW>r"b*e<AZL>h</H> to obtain the example above. To avoid a final form one uses the directive <NIL>, after the letter as well: <H>mi<MER><NIL></H> for the example. More technically, in the first case, the preprocessor considers it is at the end of a word and treats the two parts of the word as distinct—but concatenated—words; in the second case an invisible character of zero width makes it think it is *not* at the end of the word.

Letters with more than one vowel. Again because of contractions or other grammatical phenomena, a letter can carry more than one vowel. Here is an example: כַּאֲשֶׁר Ez 9:11, where the letter *kaph* carries both a *šewa* and a *ḥolem*. Input of such letters is straightforward.

Isolated *dageš*. The author encountered an isolated, vowelized *dageš* in BHS: הִן Jes 54:16.

To obtain this character with *Tiqwah*, use the directive <DAGESH>. The invisible box of this character is sufficiently wide to carry vowels and/or other diacritics. It is treated as any other letter, so you have to use the directive <EOW> (see the section 'Other preprocessor directives' on page 176) to obtain our unique example (otherwise the letter *nun* will not be final). Here is its *Tiqwah* code: <H>hin<EOW><DAGESH>e<MEH></H>.

Unusual letters. In Nu 10:35–36 as well as in Ps 107, one encounters the horizontally inverted letter *nun* ׆. In the critical apparatus of BHK one can read "נ invers: [editio Bombergiana Jacobi ben Chajjim anni 1524/25] בנסע et כמתאננים". Both in BHS and BHK the types used for this character are not very satisfactory, while in Holzhausen Bible (1889), Lowe and

Brydone Bible (1948) a type of the same quality as the ordinary *nun* is used.

This character is obtained by the code n/. It seems that other inverted characters may exist (an inverted *lamed* seems to be hidden in the Bible text...). They will be added to the *Tiqwah* system, whenever necessary.

In Nu 25:12, there is a "broken" *waw* with right *ḥolem*, in the word שָׁלוֹם. This character is obtained by the code w/^o.

Finally, there is a variant form of the letter *qoph*, in Ex 32:25, בְּקָמֵיהֶם, and in Nu 25:12, עַל־הַפְּקֻדִים. This letter is mentioned in the Masorah as "*qof* joined and without *taggim*" (see Yeivin 1980, §46); it can be obtained by the code q/.

Missing letters. The treatment of missing letters is typical of the work and restrictions of Masorets: they were not allowed to add letters that were missing, so while vowelizing the consonants they did so also for the missing letters, and by that action made their existence apparent.[15]

In the Holzhausen Bible (1889), and Lowe and Brydone Bible (1948), an asterisk is used to denote a missing letter. This asterisk is vowelized just like any ordinary letter. In BHS and BHK different methods are used: in some cases, empty space is left; in other cases no empty space is left and the diacritics of the missing character are just squeezed between those of (not missing) letters (a phenomenon occuring also in Holzhausen Bible (1889), Lowe and Brydone Bible (1948); for example in the word יְרוּשָׁלִָם Ps 137:6 where a *ḥireq* is squeezed between the *lamed* and the final *mem*).

Here are the missing letters detected by the author, as printed in Holzhausen Bible (1889) and in Lowe and Brydone Bible (1948):

יִשְׁאַל־* 2S 16:23,
עַל־* 2S 18:20,
וַ*תִּגְעַשׁ 2S 22:8,
*שְׂבָכָה 1R 7:20,
וְאֶת־תַּ*מֹר 1R 9:18,
*מֶּלֶךְ 1R 15:18,
מֵעַל* 1R 20:41,
מֵאַ*ן 2R 5:25,
לְ*גִּיד 2R 9:15,
*מֶּלֶךְ 2R 11:20,
בֵּית־*מֶּלֶךְ 2R 15:25,
וְ*כַּרְמֶל Jes 32:15,
*תַּחַת Jes 55:13,

[15] The reader can compare this with the glasses or gloves worn by the invisible man in H. G. Wells's homonymous novel.

*אֶרֶץ Jer 10:13,
*עָם Jer 17:19,
וְ*הָיוּ Jer 18:23,
*דָּבָר Jer 40:3,
*רָשָׁע Ez 18:20,
וְקִרְיָתָ*מָה Ez 25:9,
*לְשָׁכוֹת Ez 42:9,
בַּיַּרְכָתָ*ם Ez 46:19,
*יִשָׁנוּ Prv 4:16,
*יוֹלַדְתֶּךָ Prv 23:25,
*עַד Hi 2:7,
יִגְ*אָל Ru 3:13,
כָל־*עַמִּים Thr 1:18,
*לֹא Thr 2:2,
*אֵין Thr 5:3,
הִזְ*מִנְתּוּן Da 2:9,
*דִּי Da 2:43.

In *Tiqwah* one writes <AST> to obtain the letter-like asterisk (warning, the ASCII asterisk * is used only for the *dageš*, *mappiq* and *šureq* dot!). If one prefers to leave an empty space, one can use the directive <EMPTY>. Unlike <NIL>, this one produces an invisible character with *non-zero* width; it can be vowelized just like any character. Finally, <NIL> can be used if we want to squeeze the diacritics of the missing character between the existing characters/diacritics.

Here is an example: <H>b"<AST>ag*iyd</H> will give the (imaginary) word בְּ*גִּיד; by replacing <AST> by <EMPTY> in the code, one would get בְּ גִּיד and finally, by using <NIL> instead of <AST> or <EMPTY>, the result would be בְּגִּיד.

Missing words. To indicate the location of missing words, all combinations of the preceding techniques are used. In BHK and BHS, empty vowelized characters are used; in Holzhausen Bible (1889), Lowe and Brydone Bible (1948), a single asterisk, in the middle of the diacritics of the missing word is used. In a single case, a digit zero is used instead of asterisk. Here are the missing words detected by the author, as printed in Holzhausen Bible (1889), Lowe and Brydone Bible (1948):

0 Jdc 20:13,
* 2S 8:3,
* 2R 19:31,
* 2R 19:37,
* Jer 31:37,
* Jer 50:29,
* Ru 3:6,
* Ru 3:17.

To obtain the digit zero in *Tiqwah*, use the directive <ZERO>. For the remaining examples, the directives explained in the section 'Missing letters' on page 183 are used in a straightforward manner.

Conclusion

As hinted by its name (*Tiqwah* means "hope" in Hebrew), the author has made this system hoping that it will lead to a revival of Biblical Hebrew typography. Its three main axes (fonts, typesetting, user interface) are based on three powerful programming languages: METAFONT for font creation, TeX for typesetting, and GNU Flex for preprocessing. The openness and flexibility of these languages guarantees the platform independence and consistency of the *Tiqwah* system.

The author would like to express his gratitude to Prof. Johannes de Moor of the Theologische Universiteit van de Gereformeerde Kerken (Kampen) for his constant and friendly guidance and support. Also he would like to thank Jean Kahn (Paris) for his help in the hunt for rare cases and typographical curiosa, and Alan Hoenig (New York), Daniel Navia (Paris) and Reinhard Wonneberger (Mainz) for their warm response and friendly advice. Last, but not least, many thanks to those who have fetched the preprint of this paper on the net and have generously provided suggestions and corrections: Abe Stone (Princeton), Aaron Naiman (Maryland), Scott Smith (MIT), Malki Cymbalista (Weizmann Institute, Israel).

References

BHK. תורה נביאים וכתובים *Biblia Hebraica, edidit Rud. Kittel, Professor Lipsiensis (Editio altera emendatior stereotypica).* Stuttgart, 1925.

BHS. תורה נביאים וכתובים *Biblia Hebraica Stuttgartensia.* Stuttgart, 1987.

Holzhausen Bible. תורה נביאים וכתובים. Vienna, 1889.

Lettinga, J.P. *Grammaire de l'hébreu biblique.* E.J. Brill, Leiden, 1980.

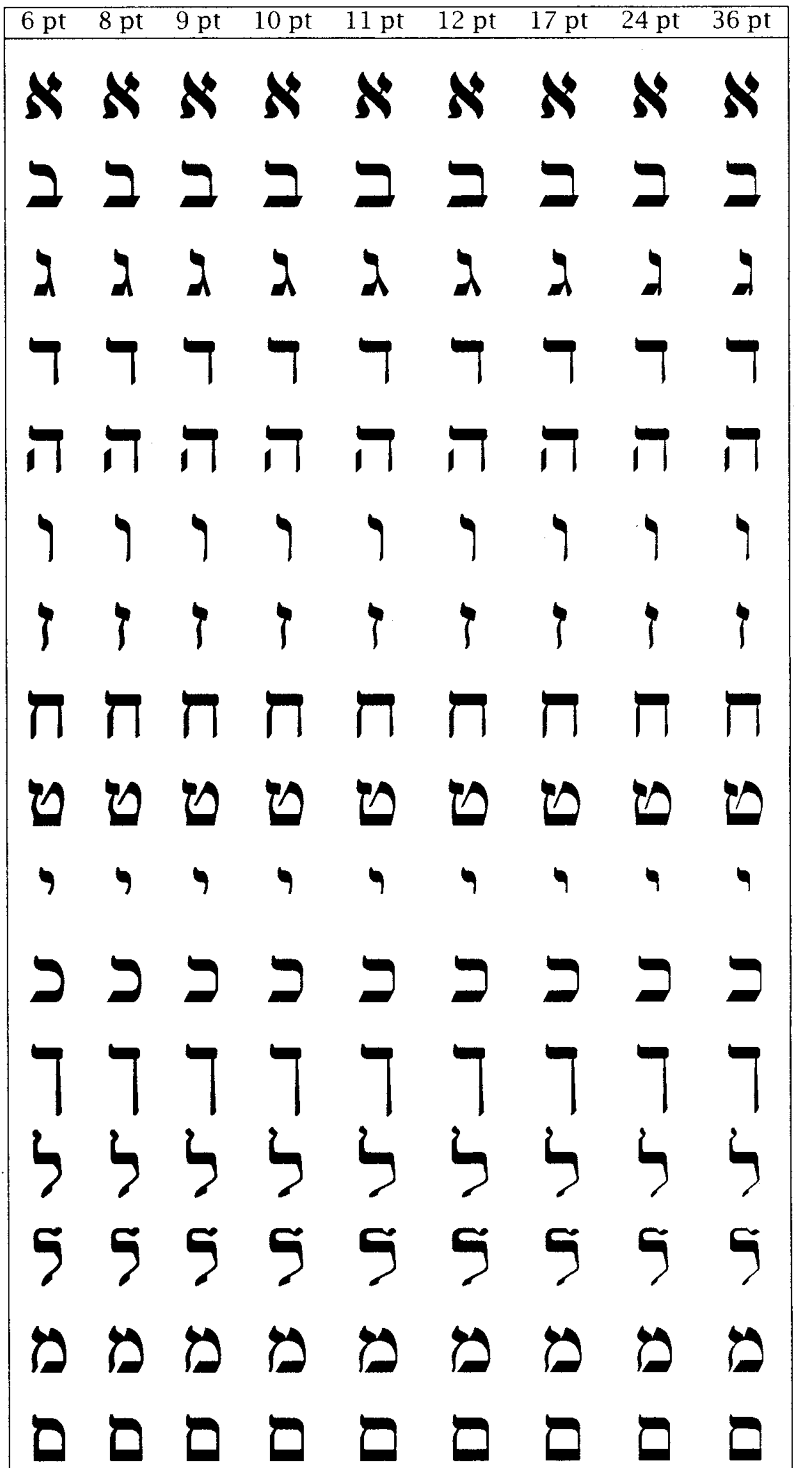
Levine, J.A. *Synagogue Song in America.* White Cliffs Media Company, Crown Point, Indiana, 1988.

Lowe and Brydone Bible. תורה נביאים וכתובים. London, 1948.

Raskin, S. *Haggadah,* הגדה של פסח. Academy Photo Offset Inc., New York, 1941.

Wonneberger, R. *Understanding BHS. A Manual for the Users of Biblia Hebraica Stuttgartensia.* Editrice Pontificio Instituto Biblico, Roma, 1990.

Yeivin, I. *Introduction to the Tiberian Masorah.* Scholars Press, Missoula, Montana, 1980.



6 pt	8 pt	9 pt	10 pt	11 pt	12 pt	17 pt	24 pt	36 pt
א	א	א	א	א	א	א	א	א
ב	ב	ב	ב	ב	ב	ב	ב	ב
ג	ג	ג	ג	ג	ג	ג	ג	ג
ד	ד	ד	ד	ד	ד	ד	ד	ד
ה	ה	ה	ה	ה	ה	ה	ה	ה
ו	ו	ו	ו	ו	ו	ו	ו	ו
ז	ז	ז	ז	ז	ז	ז	ז	ז
ח	ח	ח	ח	ח	ח	ח	ח	ח
ט	ט	ט	ט	ט	ט	ט	ט	ט
י	י	י	י	י	י	י	י	י
כ	כ	כ	כ	כ	כ	כ	כ	כ
ך	ך	ך	ך	ך	ך	ך	ך	ך
ל	ל	ל	ל	ל	ל	ל	ל	ל
ל	ל	ל	ל	ל	ל	ל	ל	ל
מ	מ	מ	מ	מ	מ	מ	מ	מ
ם	ם	ם	ם	ם	ם	ם	ם	ם

Table 1: Hebrew characters in point sizes 6-36 (part A)

6 pt	8 pt	9 pt	10 pt	11 pt	12 pt	17 pt	24 pt	36 pt
נ	נ	נ	נ	נ	נ	נ	נ	נ
ן	ן	ן	ן	ן	ן	ן	ן	ן
ס	ס	ס	ס	ס	ס	ס	ס	ס
ע	ע	ע	ע	ע	ע	ע	ע	ע
פּ	פּ	פּ	פּ	פּ	פּ	פּ	פּ	פּ
ףּ	ףּ	ףּ	ףּ	ףּ	ףּ	ףּ	ףּ	ףּ
צ	צ	צ	צ	צ	צ	צ	צ	צ
ץ	ץ	ץ	ץ	ץ	ץ	ץ	ץ	ץ
ק	ק	ק	ק	ק	ק	ק	ק	ק
ר	ר	ר	ר	ר	ר	ר	ר	ר
שׁ	שׁ	שׁ	שׁ	שׁ	שׁ	שׁ	שׁ	שׁ
ת	ת	ת	ת	ת	ת	ת	ת	ת
ﭏ	ﭏ	ﭏ	ﭏ	ﭏ	ﭏ	ﭏ	ﭏ	ﭏ

Table 1: Hebrew characters in point sizes 6–36 (Part B)

א	ב	ג	ד	ה	ו	ז	ח
'	b	g	d	h	w	z	x
ט	י	כך	ל	מם	נן	ס	ע
T	y	k	l	m	n	s	‘
פף	צץ	ק	ר	שׁ	שׂ	ש	ת
p	Y	q	r	W	S	W/	t
אּ	בּ	גּ	דּ	הּ	וּ	זּ	חּ
'*	b*	g*	d*	h*	w*	z*	x*
טּ	יּ	כּךּ	לּ	מּםּ	נּןּ	סּ	עּ
T*	y*	k*	l*	m*	n*	s*	‘*
פּ	צּ	קּ	רּ	שּׁ	שּׂ	שּ	תּ
p*	Y*	q*	r*	W*	S*	W/*	t*
ל	לּ	ﭏ	ﭏ	א̈	׆	וֹ	ייי
L	L*	'/l	'/L	'/	n/	w/^o	<YYY>

Table 2: Hebrew letters and their input codes

Name	Ḥireq	Ṣere	Segol	Pataḥ	Qameṣ	Ḥolem	Qibbuṣ
Tiberian							
Palestinian							
Babylonian							
Input	i	e	E	a	A	o	u
Alt. input	<HIR>	<SER>	<SGL>	<PAT>	<QAM>	<HOL>	<QIB>

Table 3: Hebrew vowels (Tiberian, Palestinian & Babylonian) and their input codes

Šewa	Ḥateph- Pataḥ	Ḥateph- Segol	Ḥateph- Qameṣ	Pataḥ furtivum		Mater lectionis	
				ח	ה	ו	א
"	Ha	HE	HA	+x	+h	w^o	'^o
<SWA>	<HPA>	<HSE>	<HQA>	<PTF>x	<PTF>h*	w<RHO>	'<RHO>
	ה	ש	ס	ה	ש	ס	
	<UHE>	<USH>	<USA>	<LHE>	<LSH>	<LSA>	

Table 4: Special Hebrew vowels, special characters and their input codes

Silluq	Atnaḥ	Ṭiphḥa	Mereka	Munaḥ	Mehupak	Mereka kepula	Darga
<SIL>	<ATN>	<TIP>	<MER>	<MUN>	<MEH>	<MEK>	<DAR>
Xsil	Xatn	Xtip	Xmer	Xmun	Xmeh	Xmek	Xdar

	Zaqeph		Rebia				Pazer
Galgal	parvum	magnum	magnum	Gereš	Garšayim	Pazer	magnum
<GAL>	<ZQP>	<ZQM>	<RBM>	<GER>	<GAR>	<PAZ>	<PZM>
Xgal	Xzqp	Xzqm	Xrbm	Xger	Xgar	Xpaz	Xpzm

Azla	'Ole weyored	'Illuy	Šalšelet	Ṣinnorit	Circellus	Tebir	Rafe
<AZL>	<OLE>	<ILL>	<SHP>	<SIN>	<CIR>	<TEB>	<RAF>
Xazl	Xole	Xill	Xshp	Xsin	Xcir	Xteb	Xraf

		Postpositivi					
Punctum extraordinarium	Varika	Circellus	Lineola (paseq)	Segolta	Sinnor, Zarqa	Pašṭa	Teliša parvum
<PUN>	<VAR>	<PCR>	<LIN>	<SEG>	<ZAR>	<PAS>	<TLP>
Xpun	Xvar	Xpcr	Xlin	Xseg	Xzar	Xpas	Xtlp

Praepositivi							
Rebia mugrash	Dehi	Yetib	Teliša magnum	Setuma	Petuḥa	Maqqeph	Soph pasuq
<REM>	<DEH>	<YET>	<TLM>	<SET>	<PET>	=	:
Xrem	Xdeh	Xyet	Xtlm	Xset	Xpet		

Table 5: Oldstyle Hebrew masoretic accents and their input codes

Atnaḥ	Ṭiphḥa	Mereka	Mereka kepula	Darga	Galgal	Gereš	Garšayim
<ATN>	<TIP>	<MER>	<MEK>	<DAR>	<GAL>	<GER>	<GAR>
Xatn	Xtip	Xmer	Xmek	Xdar	Xgal	Xger	Xgar

			Postpositivi		Praepositivi		
Pazer	Azla	Ṣinnorit	Tebir	Sinnor, Zarqa	Pašṭa	Rebia mugrash	Dehi
<PAZ>	<AZL>	<SIN>	<TEB>	<ZAR>	<PAS>	<REM>	<DEH>
Xpaz	Xazl	Xsin	Xteb	Xzar	Xpas	Xrem	Xdeh

Table 6: Modern Hebrew masoretic accents and their input codes

'	''		*	*	0	·	
!	!!	<MIL>	<AST>	<ASA>	<ZERO>	<DAGESH>	<XXX>
		Xmil	Xast	Xasa	Xzer	Xdag	Xxxx

Table 7: Miscellaneous symbols and their input codes

בראשית

GENESIS

Caput I. א

בְּרֵאשִׁית בָּרָא אֱלֹהִים אֵת הַשָּׁמַיִם וְאֵת הָאָרֶץ׃ וְהָאָרֶץ
הָיְתָה תֹהוּ וָבֹהוּ וְחֹשֶׁךְ עַל־פְּנֵי תְהוֹם וְרוּחַ אֱלֹהִים
מְרַחֶפֶת עַל־פְּנֵי הַמָּיִם׃ וַיֹּאמֶר אֱלֹהִים יְהִי אוֹר וַיְהִי־אוֹר׃
וַיַּרְא אֱלֹהִים אֶת־הָאוֹר כִּי־טוֹב וַיַּבְדֵּל אֱלֹהִים בֵּין הָאוֹר
וּבֵין הַחֹשֶׁךְ׃ וַיִּקְרָא אֱלֹהִים ׀ לָאוֹר יוֹם וְלַחֹשֶׁךְ קָרָא
לָיְלָה וַיְהִי־עֶרֶב וַיְהִי־בֹקֶר יוֹם אֶחָד׃ פ

וַיֹּאמֶר אֱלֹהִים יְהִי רָקִיעַ בְּתוֹךְ הַמָּיִם וִיהִי מַבְדִּיל בֵּין מַיִם
לָמָיִם׃ וַיַּעַשׂ אֱלֹהִים אֶת־הָרָקִיעַ וַיַּבְדֵּל בֵּין הַמַּיִם אֲשֶׁר
מִתַּחַת לָרָקִיעַ וּבֵין הַמַּיִם אֲשֶׁר מֵעַל לָרָקִיעַ וַיְהִי־כֵן׃
וַיִּקְרָא אֱלֹהִים לָרָקִיעַ שָׁמָיִם וַיְהִי־עֶרֶב וַיְהִי־בֹקֶר יוֹם
שֵׁנִי׃ פ

וַיֹּאמֶר אֱלֹהִים יִקָּווּ הַמַּיִם מִתַּחַת הַשָּׁמַיִם אֶל־מָקוֹם אֶחָד
וְתֵרָאֶה הַיַּבָּשָׁה וַיְהִי־כֵן׃ וַיִּקְרָא אֱלֹהִים ׀ לַיַּבָּשָׁה אֶרֶץ
וּלְמִקְוֵה הַמַּיִם קָרָא יַמִּים וַיַּרְא אֱלֹהִים כִּי־טוֹב׃ וַיֹּאמֶר
אֱלֹהִים תַּדְשֵׁא הָאָרֶץ דֶּשֶׁא עֵשֶׂב מַזְרִיעַ זֶרַע עֵץ פְּרִי
עֹשֶׂה פְּרִי לְמִינוֹ אֲשֶׁר זַרְעוֹ־בוֹ עַל־הָאָרֶץ וַיְהִי־כֵן׃ וַתּוֹצֵא
הָאָרֶץ דֶּשֶׁא עֵשֶׂב מַזְרִיעַ זֶרַע לְמִינֵהוּ וְעֵץ עֹשֶׂה־פְּרִי
אֲשֶׁר זַרְעוֹ־בוֹ לְמִינֵהוּ וַיַּרְא אֱלֹהִים כִּי־טוֹב׃ וַיְהִי־עֶרֶב
וַיְהִי־בֹקֶר יוֹם שְׁלִישִׁי׃ פ

Figure 5: The book of Genesis, as printed in an 1889 Viennese Bible

Adaptive character generation and spatial expressiveness

Michael Cohen
Human Interface Lab, University of Aizu 965-80, Japan
mcohen@u-aizu.ac.jp

Abstract

Zebrackets is a system of meta-METAFONTs to generate semi-custom striated parenthetical delimiters on demand. Contextualized by a pseudo-environment in LaTeX, and invoked by an aliased pre-compiler, *Zebrackets* are nearly seamlessly invokable in a variety of modes, manually or automatically generated marked matching pairs of background, foreground, or hybrid delimiters, according to a unique index or depth in the expression stack, in 'demux,' unary, or binary encodings of nested associativity. Implemented as an active filter that re-presents textual information graphically, adaptive character generation can reflect an arbitrarily wide context, increasing the information density of textual presentation by reconsidering text as pictures and expanding the range of written spatial expression.

Adaptive Character Generation: Zebrackets

Zebrackets [Cohen 92] [Cohen 93] takes a small-scale approach to hierarchical representation, focusing on in-line representation of nested associativity, extending parentheses (also known as "lunulae" [Lennard 91]), and square brackets (a.k.a. "crotchets"), by systematically striating them according to an index reflecting their context.

Functionality. Table 1 crosses three of the dimensions currently supported by *Zebrackets*, using a LISP function (which performs a generalized "inclusive or") as a scaffolding.

index is the semantic value of the pattern being superimposed on the delimiters:

- **unique** generates a unique, incremental index for each pair of delimiters
- **depth** calculates the depth of the delimited expression in an evaluation stack, useful for visualizing expression complexity

encoding scheme refers to the way that the index is represented visually:

- **demux** named after a demultiplexer, or data selector, which selects one of n lines using $\lg_2 |n|$ selectors, puts a 'slider' on the delimiter. Such a mode is useful for establishing spatial references, as in (top)(middle)(bottom).
- **unary** creates a simple tally, a column of tick marks
- **binary** encodes the index or depth as a binary pattern, the most compact of these representations

The `demux` encoding mode always has exactly one band or stripe, but the `unary` and `binary` encodings have variable numbers, and use an index origin of zero to preserve backwards compatibility. Since the striations are adaptively chosen, the complexity of the delimited expression determines the spacing of the streaks. Without NFSS, the maximum number of stripes for a self-contained face is $\lg_2(\frac{256}{2}) = 7$. Otherwise, for overly rich expressions that exceed visual acuity, *Zebrackets* can be limited to a fixed striation depth, wrapping around (repeating) the indexing scheme if the delimiters exhaust the range of uniquely encodable values, as seen in the `unique` × {`demux`|`unary`} sextants.

type controls the style of the striations superimposed on pairs of delimiters:

- **background** bands drop out segments from the delimiters
- **foreground** explicitly put in black ticks, which are more legible if less inconspicuous
- **hybrid** combines these two styles, dropping out bands at all the possible slot locations, and then striping the actual index

Eventually perhaps, greyscale striations (not yet implemented) might interpolate between these approaches, causing the ticks to disappear at normal reading speed, but be visible when doing a detailed search.

Foreground *Zebrackets* only work well with thinner faces, and background *Zebrackets* only with bolder faces. Figure 1 exercises *Zebrackets* through an obstacle course of less common fonts, showing some of the legibility problems, even with figure/ground modes chosen to flatter the filigrees.

encoding	type	index: unique	index: depth
demux	background	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))
	foreground	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))
	hybrid	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))
unary	background	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))
	foreground	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))
	hybrid	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))
binary	background	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))
	foreground	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))
	hybrid	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))	(DEFUN ANY (LST) (COND ((NULL LST) NIL) ((CAR LST) T) (T (ANY (CDR LST)))))

Table 1: index:{unique, depth} × encoding:{demux, unary, binary} × type:{background, foreground, hybrid} (10 pt. cmtcsc *Zebrackets*, selected to match size and font [small caps] of text)

ACTIVE INGREDIENT: Hydramethylnon [tetrahydro-5, 5-dimethyl-2(1*H*)-pyrimidinone(3-[4-(trifluoromethyl)phenyl]-1-(2-[4-(trifluoromethyl) phenyl]ethenyl)-2-propenylidene)hydrazone]

ACTIVE INGREDIENT: Hydramethylnon [tetrahydro-5, 5-dimethyl-2(1*H*)-pyrimidinone(3-[4-(trifluoromethyl)phenyl]-1-(2-[4-(trifluoromethyl) phenyl]ethenyl)-2-propenylidene)hydrazone]

ACTIVE INGREDIENT: Hydramethylnon [tetrahydro-5, 5-dimethyl-2(1*H*)-pyrimidinone(3-[4-(trifluoromethyl)phenyl]-1-(2-[4-(trifluoromethyl) phenyl]ethenyl)-2-propenylidene)hydrazone]

ACTIVE INGREDIENT: Hydramethylnon [tetrahydro-5, 5-dimethyl-2(1*H*)-pyrimidinone(3-[4-(trifluoromethyl)phenyl]-1-(2-[4-(trifluoromethyl) phenyl]ethenyl)-2-propenylidene)hydrazone]

Figure 1: Application of *Zebrackets* to a chemical formula (sans serif bold extended with background, sans serif with hybrid, sans serif demibold condensed with hybrid, "funny face" [negative inclination] with foreground)

Implementation. The implementation of *Zebrackets* comprises two aspects: a filter to generate permuted invocations of the underlying delimiters, and the delimiter glyphs themselves. The filter is composed of (an *ad hoc* collection of) `csh` and `sh` shell scripts and C and `perl` [Wall & Schwartz 91] programs. The two-pass filter parses selected text, invoked explicitly with editor utilities like Emacs' `shell-command-on-region` command [Stallman 88], or implicitly as a precompiler. In the latter case, sections of the document set off by the LaTeX [Lamport 86] pseudo-environment
`\begin{zebrackets}{<parameters...>}`
⋮
`\end{zebrackets}`
are replaced by zebracket invocations. This pseudo-environment is interpreted by a precompiler, like a macro processor, that replaces vanilla delimiters with zebracketed, and emits METAFONT [Knuth 86] source that will be invoked at image time.

The first pass parses the expression using a stack, establishes the maximum number of stripe slots needed, and generates the necessary METAFONT files. For the unique index mode, the maximum number of striations is the number of bits needed to represent its highest index, which is equal to $\lceil \lg_2 |\text{delimiter pairs}| \rceil$. Using the context established by the first pass, the second pass replaces each delimiter with LaTeX code invoking its respective zebracketed version by effectively traversing the underlying tree. As seen in Figure 1, different styles of delimiters (like rounded parentheses and square brackets) are handled separately, and the respective striation slots are spaced out evenly along the height of the delimiter.

For example, invoking the aliased precompiler/compiler on a document containing the contents of Figure 2 runs the `zebrackets` filter on "`(a * (b + c))`" (with arguments that mean "automatically generate (uniquely) indexed foreground-striated binary-encoded 10pt. delimiters using `cmr` base parameters"), determines that only one potential striation is needed, encodes the indices as binary patterns, replaces the source text with that in Figure 3,[1] and generates the `zpfbcmr10.mf` source,[2] as well as the appropriate `.tfm` and `.pk` files, which together yield "(a * (b + c))" at preview (`TeXview` [Rokicki 93] via `TeXMenu` [Schlangmann 92] on NextStep) or printing (`dvips` [Rokicki 92]) time.

By having indirected the glyphs one extra level, *Zebrackets* implements a meta-METAFONT. Dynamic fonts [Knuth 88] [André & Borghi 89] [André & Ostromoukhov 89] employ what is sometimes called "dynamic programming," which is basically lazy evaluation of a potentially sparse domain. Although each *Zebrackets* character is essentially determined at edit-time, and the actual specification involves human-specified ranges for zebracketing, because of the communication between document and METAFONT, character generation is context-sensitive and adaptive, since the automatic specification can be conceptually lumped together with the compilation (via `latex`) and imaging.

Currently the size of the delimiters and the name of the Computer Modern model font are

[1] Idempotency of font declarations is finessed by the `\ifundefined` condition [Knuth 84], pages 40, 308.

[2] The syntax of METAFONT terminates a token upon encountering a digit, so no numbers can be used directly as part of a font name. Therefore, the number of striations is mapped to an alpha character ('a'⇒0 stripes, 'b'⇒1 stripe, ...), which becomes, after 'z' [for *Zebrackets*], 'b' or 'p' [for parentheses, or brackets], and 'b', 'f', or 'h' [for back- or foreground, or hybrid], the fourth character in the font name.

```
\documentstyle[zebrackets]{article}
\begin{document}
⋮
\begin{zebrackets}{f,-1,-1,binary,10,cmr}
(a * (b + c))
\end{zebrackets}
⋮
\end{document}
```

Figure 2: Sample (LaTeX pseudo-environment) input

```
\documentstyle[zebrackets]{article}
\begin{document}
⋮
\ifundefined{zpfbcmr}\newfont{\zpfbcmr}{zpfbcmr10}\fi
{\zpfbcmr\symbol{0}}a * {\zpfbcmr\symbol{1}}b + c{\zpfbcmr\symbol{3}}{\zpfbcmr\symbol{2}}
⋮
\end{document}
```

Figure 3: Sample (*Zebrackets* filter) output

explicitly passed as parameters to the pseudo-environment. A more elegant approach would be to code the *Zebrackets* filter directly as a *bona fide* LaTeX environment, which could determine delimiter size and font at compile time (writing information to an `.aux` file and using something like "`\immediate\write18`"to escape to the operating system to create and invoke `mf` files). *Zebrackets'* implementation as a precompiler insulates the characters from useful positional and contextual information, like page position and current font and size. Otherwise, *Zebrackets* is compatible with (perhaps redundant) LaTeX dimensions, as overstated by Figure 4.

The *Zebrackets* filters slow down document compilation considerably. However, since they are usually image-level compatible, a document may be previewed quickly in a *Zebrackets*-less mode, while the cycle-intensive *Zebrackets* run in the background, eventually seamlessly strobing into the previewer without any layout change or page motion.

Spatial Expressiveness

The notion of a fixed alphabet font is inherently limited, even one extended into a family by techniques like weighting, italicization, emboldening, and local contextual tools like ligature and kerning. Computers offer the potential of "chameleon fonts," altered, depending on their context, to heighten legibility (readability, balance, or proportion) or evoke emotions that complement, reinforce, or amplify the words and ideas.

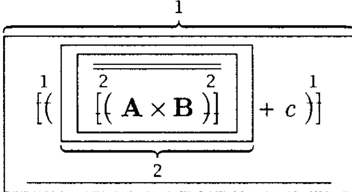


Figure 4: Celebration of nesting hyperbole: Round and rectangular tagged *Zebrackets* reinforcing interleaved (to the limits of TeX's semantic nesting stack size) tagged over- and underbraces, framing, over- and underlining, emboldening, italicization, case, natural operator precedence, and canonical left→right reading order

Zebrackets is a focused realization of adaptive character generation, useful in certain contexts, but ultimately less important than its conceptual ambitions. The logical extension of typography is arbitrarily tuned characters, calculated globally and generated uniquely. Adaptive character generation is the destiny of electronic publishing, glyphs adjusted in arbitrarily subtle ways to carry information and fit space.

***Zebrackets* and Multiple Master Typefaces.** Reading publications like Baseline and Emigre, one might think that the only computer-driven typographic innovations are on the Macintosh, using tools like Fontographer. This imbalance is perhaps because the formalization of a meta-language (and its corollary meta*-languages) is less accessible to artists than graphical techniques.

The notion of a meta-font language can be likened to Adobe Multiple Master Typeface [Adobe 92] [Spiekermann & Ginger 93] with an arbitrary number of axes, or dimensions, each corresponding to a parameter. (Selection of a base font can be thought of as setting lower level parameters.) Usually the glyph space is thought of as continuous, and the arguments, or components of the index vector, are floating point.

Zebrackets' integral characterization of the text yields a quantized specification of a font; real numbers would allow for continuous variation (within, of course, the resolution of finite precision encoding), expanding even further the ability to custom-tailor a font for a context. Such variety might manifest as arbitrarily soft typefaces, perhaps employing greyscale or dynamic effects, or tuned by the reader, to match visual acuity.

Quantification of dimensionality. Visual languages combine textual and graphical elements. Spatial expressiveness is achieved not only via effects like *Zebrackets*, but any kind of systematic control of document presentation—explicit parameters like margins, but also implicit global characteristics, like consistency or contrast of typographic features.

Words have different expressive qualities than pictures, but treating text as pictures, interpolating between 1D textual streams and 2D graphical representations, enables some of the best qualities of both. Table 2 attempts to align this spatial expressiveness with computer languages and communication modalities, suggesting that typeset documents have a dimensionality somewhere between 1 and 2.[3]

It is amusing to try to estimate the value of this non-integral dimension. We can assume that a document composed entirely of (captionless) pictures is fully 2-dimensional, and a document stripped of graphical cues, denuded ASCII, to be entirely one-dimensional, and that the interpolation between is (linearly) proportional to the fraction of the respective components, as shown in Figure 5.

Using an information theoretic assumption that a metric of a vector is proportional to its length, and that languages are Huffman encoded, so that clichéd expressions are terse, then the most expressive will be the longest, and a heuristic for spatial expressiveness is simply to compare the magnitude of a graphical file with that of the underlying text:

$$S = \frac{|graphics| - |text|}{|graphics|} + 1 \qquad (1)$$

where $|text|$ is the length of the text substrate, $|graphics|$ is the length of the graphical file (which includes all the text), and S is the dimension of spatial expressiveness. In particular, the character counts of the PostScript (`.ps`) file and the `detexed` LaTeX (`.tex`) and bibliography (`.bbl`) files can be used to characterize the relative weights of the respective components. Using this formula, a simple shell script, shown in Figure 6, calculates the dimension of this document to be about 1.94. This value is inflated by including the font encoding in the ps file, but such a dilation is an expected consequence of sharpening the granularity of the document rendering. A (perhaps not undesirable) consequence of such a definition is that dimension varies with output resolution.

In contrast, we would expect the spatial expressiveness of a graphically-challenged document, `detexed` source embraced by a minimalist compiling context to be closer to unity. As seen in Table 3, and corresponding with this intuition, small documents are mostly graphical, with dimensionality near 2, but as their textual component lengthens, they become more vector-like, with dimension closer to 1. The same text, but zebracketed, yields, as expected, higher values of spatial expressiveness, except for the lowest character counts, where the heuristic manifests artifacts of `detex` idiosyncrasies. Anyway, the test files used to generate these metrics are more than a little artificial, because empty lines are needed to prevent TeX's paragraph buffer from overflowing, and *Zebrackets'* wrap-around restriction currently makes it impossible to generate even a contrived document in which every character is unique.

The usefulness of such a metric is bounded by the validity of its model; particularly suspect is the assumption of equivalence of graphical information, yielding artifacts of an over-simplified characterization. It seems intuitive that the information in text scales according to length ("A = B" has roughly half as much information as "A = B = C"), but does, for example, a PostScript (macroscopic) `moveto` carry, on the average, the same amount of information as a same-length fraction of a (microscopic) font encoding in a document prelude? Only arguably, in a relaxed, informal sense of "spatial expressiveness." The data must be regarded as preliminary, and further analysis is indicated.

Paradigm shift: the end of fonts. As adaptive character generation becomes increasingly intricate, compressed encodings become less relevant (since each font is disposable), and the distance between the bitmap and the next least abstract representation

[3] Of course time must be considered another dimension, or design axis. Temporal techniques, like `Emacs'` flashing pairs of parentheses, will become important in ways difficult for us to imagine now, and cinematographic techniques will start to infiltrate books (as in World-Wide Web). Perhaps rotating colors through letters, or gently inflating/deflating them, will make them easier to read. And, of course, (hyperlinked) video is inherently temporal.

spatial expressiveness (dimensionality)	computer language	communication modality
1	ASCII	
	JIS, EUC [Lunde 93]	email
	Unicode	
	Rich Text Format (rtf)	typewriting
	TEX/LATEX/METAFONT	typesetting
	+ *Zebrackets*	
	device independent (dvi)	handwriting
2	PostScript (ps)	drawing & photography
	texture maps	
		painting
	Renderman (rib)	
3		sculpture

Table 2: Correspondence of dimension with hierarchy of languages and media as richness of expression

dimensionality slider	
1	2
2D partition: graphical layout, typography, ...	1D partition: textual substrate

Figure 5: Partitioning of document into graphical and textual components: the fraction of a document's graphical content determines its dimensionality

grows. Fonts as we know them will become singletons, eclipsed by transformations and geometric manipulations. Document manipulation will be organized as filters—not only conventional idioms like emboldening, underlining, size and color contexts, but also legibility sliders, path-following and space-filling constraints, visual overtones, and temporal effects. Not only will characters be morphed, but characteristics will be crossed and composed. Such promiscuous intermingling of these filters, dancing to graphical rhythms that reverberate through the document, will legitimize an intermarriage between perspectives and multiple inheritance of eclectic legacies.

An example of such local manifestation of global context, inspired by the notion of a cross-reference as a back-traversable hyperlink, can be seen in this paper's references section, whose (superimposed demux-style) zebracketed keys indicate the pages of all the respective citations. An extension to *Zebrackets* (the intricacies of which deserve another paper) automatically uses `.aux`, `.bbl`, and `.idx` files to striate the bibliographic tags for back-references, each of the delimiter slots representing a page of the document. (The body of a paper, excluding the bibliography, can be at most seven pages long, since only up to seven striations are currently encoded by *Zebrackets*.) The left delimiter points to the `\cites` and the right indicates the `\nocites`. Notice, for instance, that [Knuth 86] gets two explicit citations (one of which is here) and one invisible one.

Conclusion. The handwritten "publishing" of pre-Gutenberg scribes was arbitrarily subtle, with its attendant human caprice (and mistakes). Printing can be thought of as having rigidified this information transmission. The research described here loosens some of that determinism, not by randomizing the presented information, but by softening the digitized boundaries, thereby expanding the range of expression. Contextual fonts like *Zebrackets* indicate evolving modes of written representation, algorithmic descriptions driving adaptive displays, as style catches up to technology.

References

[Adobe 92] Adobe. Adobe Type 1 Font Format: Multiple Master Extensions. Technical report, Adobe Systems, February 1992.

[André & Borghi 89] J. André and B. Borghi. Dynamic fonts. In J. André and R. Hersh, editors, *Proc. Int. Conf. on Raster Imaging and Digital Typography*, pages 198–203, Lausanne, Switzerland, October 1989. Cambridge University Press. ISBN 0-521-37490-1.

[André & Ostromoukhov 89] J. André and V. Ostromoukhov. Punk: de METAFONT à PostScript. *Cahiers GUTenberg*, 4:123–28, 1989.

[Cohen 92] M. Cohen. Blush and Zebrackets: Two Schemes for Typographical Representation of Nested Associativity. *Visible Language*, 26(3+4):436–449, Summer/Autumn 1992.

```
#!/bin/sh
INPUTFILENAME='basename $1 .tex'
GRAPHICS='dvips -o "!cat" $INPUTFILENAME.dvi | wc -c'
TEXT='cat $INPUTFILENAME.tex $INPUTFILENAME.bbl | detex | wc -c'
echo 'echo 3 k $GRAPHICS $TEXT - $GRAPHICS / 1 + p | dc'
```

Figure 6: Shell script to estimate spatial expressiveness (dimensionality) of a compiled LaTeX document

target characters	pages	without *Zebrackets*			with *Zebrackets*		
		TEXT	GRAPHICS	S	TEXT	GRAPHICS	S
1	1	14	4073	1.996	49	4090	1.988
10	1	41	4193	1.990	76	6427	1.988
100	3	311	6001	1.948	346	30194	1.988
1000	23	3011	21009	1.856	3046	84223	1.963
10000	228	30011	154907	1.806	30046	226971	1.867
100000	2273	300011	1501805	1.800	300046	1661796	1.819

Table 3: Quantification of spatial expressiveness: stripped down and striped up

[Cohen 93] M. Cohen. *Zebrackets: a Pseudo-dynamic Contextually Adaptive Font.* *TUGboat: Communications of the TeX Users Group*, 14(2):118–122, July 1993. ISBN 0896-3207.

[Knuth 84] D. E. Knuth. *The TeXbook.* Addison-Wesley, 1984. ISBN 0-201-13448-9.

[Knuth 86] D. E. Knuth. *The METAFONTbook.* Addison-Wesley, 1986. ISBN 0-201-13444-6.

[Knuth 88] D. E. Knuth. A punk meta-font. *TUGboat: Communications of the TeX Users Group*, 9(2):152–168, August 1988. ISBN 0896-3207.

[Lamport 86] L. Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley, 1986. ISBN 0-201-15790-X.

[Lennard 91] J. Lennard. *But I Digress: Parentheses in English Printed Verse.* Oxford University Press, 1991. ISBN 0-19-811247-5.

[Lunde 93] K. Lunde. *Understanding Japanese Information Processing.* O'Reilly & Associates, 1993. ISBN 1-56592-043-0.

[Lupton & Miller 90] E. Lupton and J. A. Miller. Type writing. *Emigre*, (15):i–viii, 1990.

[Rokicki 92] T. G. Rokicki. dvips 5.491, 1992.

[Rokicki 92] T. Rokicki. *NeXTTeX 3.141*, 1993.

[Rokicki 93] T. Rokicki. TeXview 3.0, 1993.

[Schlangmann 92] H. Schlangmann. TeXmenu, 1992. 4.1.

[Spiekermann & Ginger 93] E. Spiekermann and E. Ginger. *Stop Stealing Sheep & find out how type works.* Adobe Press, 1993. ISBN 0-672-48543-5.

[Stallman 88] R. M. Stallman. *GNU Emacs Manual.* Free Software Foundation, 1988.

[Wall & Schwartz 91] L. Wall and R. L. Schwartz. *Programming perl.* O'Reilly & Associates, Inc., 1991. ISBN 0-937175-64-1.

Humanist

Yannis Haralambous
Centre d'Études et de Recherche sur le Traitement Automatique des Langues
Institut National des Langues et Civilisations Orientales, Paris.
Private address: 187, rue Nationale, 59800 Lille, France.
Yannis.Haralambous@univ-lille1.fr

Abstract

The goal of *Humanist* is to *Human*ize LaTeX. It is a concept, a document markup syntax and a package of programs, macros and fonts. The concept of *Humanist* is the use of word processors as "rich" TeX input devices. The user shall input, edit and store a document in the most friendly and natural manner (in other words, *without a single LaTeX command*), and be provided with syntactically correct and platform-independent LaTeX output. The document input, markup and editing is done using any word processor with RTF output and TrueType (or PostScript) screen display possibilities (for example Word and WordPerfect for Mac, Windows, NeXT, X-Window etc.). *Humanist* will convert the RTF output into LaTeX code.

Paper withdrawn by the author.

Problems of the conversion of METAFONT fonts to PostScript Type 1

Basil K. Malyshev
Institute for High Energy Physics, IHEP, OMVT, Moscow Region, RU-142284 Protvino, Russia
malyshev@mx.ihep.su

Abstract

The paper describes problems pertaining to the automatic conversion of METAFONT fonts into the PostScript Type 1 font format. Several methods of conversion are discussed. A short description of the *Paradissa Fonts Collection* is presented. It contains Computer Modern fonts (used in (LA)TEX) in ATM compatible PostScript Type 1 format. The use of the collection and the problems related to it are discussed.

This paper will be published in the next issue of *TUGboat*.

The (Pre)History of Color in Rokicki's `dvips`

James Lee Hafner
IBM Research Division, Almaden Research Center, K53/802, 650 Harry Road, San Jose, CA 95120-6099, USA
hafner@almaden.ibm.com

Abstract

In this paper I give an abbreviated history of the current scheme for using color with Rokicki's `dvips` program up to the end of 1993. The real story begins in early 1990, when a local user asked if I could add to the fledgling FoilTEX project support for color to take advantage of our new color printers. This started a major effort, in collaboration with Tom Rokicki, to find an efficient and simple method for specifying color in TEX documents.

Introduction

> The `\special` command enables you to make use of special equipment that might be available to you, e.g., for printing books in glorious TEXnicolor.
>
> *D. E. Knuth*

As the quote above indicates, the grand wizard himself expected that color could (would?) be incorporated into TEX. He expected that this would be done through the use of `\special` commands to the dvi driver. In spite of this, not much was done with color for many years. Even SLITEX, where color is very desirable, was written to handle color in a rather cumbersome way.

In this paper, I will describe the efforts that went into the design and development of the current color support in Tom Rokicki's `dvips` program. I consider this the prehistory of true color support because only some of the real color issues were addressed (and many of these were done via simple hacks). Before we go into `dvips`'s method, let me set the stage.

The availability of color PostScript printers created a need for a better method to handle color. In stepped a number of people, including Leslie Lamport who wrote `color.sty` and Timothy Van Zandt who wrote PStricks. These all use literal PostScript commands passed to the dvi driver and then to the output PostScript file to create color effects. Unfortunately, there are problems with the use of literal PostScript. Namely, since each page is generally a self-contained graphics object, color effects on one page would not readily pass over to the next. Furthermore, effects at the end of a current page might trickle into the footnotes or page footer. This forces the use of these color utilities to be limited to very small parts of documents, e.g., single boxes. On the positive side, most dvi-to-PostScript drivers handle these kinds of literal PostScript `\special`s, so usability/portability was not an issue.

My character enters the story in early 1990, when a local user asked if I could add to the fledgling FoilTEX project support for color to take advantage of our new color printers. (At the time, I was not aware of the two packages mentioned above.) Using `dvi2ps` and `dvialw`, I massaged some primitive color support into these programs but certain obstacles came immediately to light. For example, in `dvialw`, large characters and all rules are placed immediately on the first pass of the page, and then the graphics environment is set up for the main characters. This is efficient for memory use but not for consistent color. If one tries to set a large square root sign in color, the opening check mark is fine but the long rule above the enclosed formula will always be black. Similar splits of colors occur for large brackets.

Finally, I came across Rokicki's `dvips` and determined that this is very well suited for color. Some of the reasons for this are stated below. This started a collaboration with Tom about how one could achieve the desired effects. In the next section I discuss the relevant issues. Later I talk about the first real attempts at getting at the problem. Finally, we describe the current system in some detail and discuss some of the limitations.

The Issues

There were a number of issues that we had to deal with at three different levels of the process. At the TEX-level (i.e., for the user macros) we wanted them to work across formats so that they could be used in FoilTEX as well as Plain TEX and LATEX, for example. We had two somewhat conflicting requirements at this level as well. We wanted to allow the naive user to specify colors without having to know a specific color model (do you know what RGB=`(1,.5,.2)` will look like? do you even know what RGB stands for or the notation `(1,.5,.2)`?). At the same time we wanted enough functionality in the underlying system to let sophisticated color experts use a broad range of color models and effects. Furthermore, the macros should lend themselves to the kind of effects one would expect with regards to TEX's grouping. For example, it should be possible to nest colors with the expected results.

Equally important from our point of view, the macros should be *device-independent*. In particular, they should not be written so that only PostScript printers could handle them. This means that the `\special` keywords should not invoke literal PostScript, but be generic. The transformation from these generic keywords to the device language (e.g., PostScript) should be handled by the driver itself. As far as I know, at present only `dvips` and TeXview on a NeXT handle these `\special`s. Hopefully, in the next era in this history (see Rokicki's article in these proceedings), more drivers will be added to this list.

Furthermore, nesting information should not get lost at each new page or other structural break, nor should the order of the pages matter when processing. This of course requires careful handling at the driver level. It must track this nesting information and be able to restore state for any specific page. Structural breaks include but are not limited to margin paragraphs, footnotes, headers and footers.

On the other hand, it is important to note when dealing with color that different rendering devices (even if they are POSTSCRIPT devices) can produce dramatically different perceptual colors on the same input. For example, on a Tektronix wax printer, green is dark and rich whereas on an X-display the same color is much lighter and even phosphorescent. Ideally the driver should be able to customize itself for this discrepancy, at least on named colors.

`Dvips`'s prescanning processes and its ability to modify its behavior for different printers were ideally suited to these ends. (Besides, it is well written code and so easy to dive into to add modifications.)

There is one issue that we did not address. That is the issue of "floats". By floats, we mean anything that appears in some place other than at the current point where TEX encounters it. This includes the obvious floats like figures and tables as well as the more subtle issue of footnotes (particularly long footnotes that might get split across pages) and saved boxes. The problem here is that color attributes at the time the float is processed may conflict with color attributes at the time the float is placed in the document. For example, a float that is encountered when text is blue and background is yellow may float to a page that has a yellow background. There are two possible approaches to this, namely, the float picks up the attributes on the page on which it is set or it takes its attributes (and the surrounding attributes) with it to the float page. In this case, the float may have a boxed background that differs from the main page on which it is set. As should be obvious, this problem is very subtle and it is not clear what approach is the best to take. Some local grouping *à la* the current scheme may provide a partial solution to the problem using the first approach, though we have not experimented with it at all.

A First Pass

In the first attempts at addressing this problem of color, we ignored the device-independence of the `\special` keywords and attempted to find a solution that required very minimal (if any) changes to the original `dvips` code.

We used literal PostScript strings in `\special` macros. There were two kinds of macros. Ones that just set the color state, and another that tried for nesting. This saved the current color state on the PostScript stack, set the color and at the end of the grouping, restored the color state from the stack. For example,

```
\def\textRed{%
      % set color to Red
    \special{ps:1 0 0 setrgbcolor}}
      % save current color
\def\Red#1{\special{ps: currentrgbcolor}
      % set color and typeset #1
            \textRed #1
      % restore old color
            \special{ps: setrgbcolor}}
```

To help with changes across page boundaries, we made a small modification to the bop (BeginOfPage), eop (EndOfPage), and `start` in the header files. Basically, eop saved the current color, bop restored the color and `start` initialized the color on the PostScript operand stack. Tom suggested that we do this on a separate color stack, an idea we never implemented because we soon abandoned this approach. We realized that this method was inherently flawed because it was too much tied to PostScript and it only worked if the document was processed front to back with all pages printed. We thought about storing more of the color stack information in the PostScript itself, but this still suffered from a number of limitations, not the least of which is the first one mentioned above.

The Current Scheme

After realizing that any attempt to do this work in the PostScript code was either doomed or too costly in terms of PostScript resources, we determined that it would be best to have `dvips` track everything internally, primarily during the prescan and then when a color is changed (either by starting a new color region or closing one), simply output a "set color" command in PostScript.

Below are the basic features of this scheme.

The `\special` Keywords. All color `\special`s begin with the keyword `color` (with one exception). The "parameters" to this keyword and their desired effects are described below:

ColorName
: Declare the current color to be the specified color. Furthermore, drop all current color stack

history, and tell the driver to set the new color state to *ColorName* (see the section on Header Files).

Model Parameters
: Same effect on the color stack history as above. The new color state will have model determined by *Model*, e.g., rgb, cmyk, cie, lab, etc., and parameters *Parameters* in that model.

push *ColorName*
: Append or save the current color to the color stack, set the new color to *ColorName*.

push *Model Parameters*
: Same as above but build the new color state from the model and parameters.

pop
: Take previous color off the color stack and tell the driver to use this for the new color state (used for closing of group).

There is one additional color special keyword. This is background. It is used with either the options *ColorName* or *Model Parameters* to tell the driver to set the background color of the current page and subsequent pages.

Some things to note about this system. First, it is completely generic, with no reliance on PostScript. Second, it assumes that some color names are known to the driver or are defined in the output file. For example, in PostScript, dvips could predefine Red to be `1 0 0 setrgbcolor` or `0 1 1 0 setcmykcolor`. (In fact it uses the latter.) The user might also be able to use the drivers' literal strings mechanism to predefine their own color names. Third, there are two types of color settings. The first is just a "set color and forget the stack." The other "push"es the current color on the stack, sets a new color, and (presumably at the end of a group) "pop"s the last pushed color off the stack to restore. This is the basic nesting mechanism. It is limited only by the resources dvips uses. Fourth, the parsing of the flags is in a hierarchical order. First comes the color keyword to indicate a color special. Next is either a known color name or a color model. After the color model are the parameters for the chosen color. This is in slight contrast to PostScript itself which is more stack oriented and expects the parameters first. We felt that if the driver didn't understand a particular model it should recognize this in the order it parses the \special string.

This functionality of being able to specify the model and parameters allows sophisticated color users a simple option to get special effects.

Header Files. As mentioned above, we assume that a certain set of color names is already known either to the driver internally or is passed to the output file for the printer interpreter. In dvips this is done in the second manner via the color.pro header file. This is prepended automatically to the output stream as soon as any color special is encountered. In this file, two things are defined. First, the PostScript command setcmykcolor is defined in terms of setrgbcolor in order that the output can be processed by some old PostScript interpreters, i.e., ones that do not recognize this function. More precisely, this is done only if the current interpreter requires a definition for this function. Other color models could also be defined here if necessary. Second, the predefined color names are defined in terms of the CMYK (Cyan, Magenta, Yellow, Black) color model. The reasons for this choice are that most color printers use this physical model of printing. This is a subtractive color space as opposed to the additive color (RGB — Red, Green, Blue) of most displays. Another reason is that I had a good template for matching color names to parameters in the CMYK color space for a particular printer.

These colors are only conditionally defined in color.pro. If they are known by the userdict, then no new definition is added. The reason for this is that a particular device might need to have different parameters set. Dvips's configuration file mechanism can then easily be used to customize the color parameters for a particular device by inclusion of a special device header file.

I emphasize at this point the distinction between physical device and output data stream. The latter is PostScript or HPGL or PCL or some such. The former is the actual physical device. These devices can vary widely even under the same data stream. An analogy is the difference between a write-white or a write-black printer and the need for finely tuned bitmap fonts for each. They may both be PostScript printers, but they print differently. The driver should be able to compensate for the physical characteristics of a given device, if at all possible.

The Color Macros. The color macros, defined in the style file colordvi.sty, come in basically two flavors. One kind sets a new color by issuing a \special{color *ColorName*} or \special{color *Model Parameters*}. The second is a combination of pushes, sets and pops for nested local colors. Furthermore, there are user definable colors of both types, where the user declares the color model and the parameters. Finally, there is a \background macro for setting the background color. For example, the revised version of the \textRed and \Red macros defined above are:

```
      % set color to Red
\def\textRed{\special{color Red}}
      % save current color and set Red
\def\Red#1{\special{color push Red}
      % typeset #1 and restore old color
            #1 \special{color pop}}
```

These are described in more detail in Hafner (1992) as well as in the documentation for dvips and for FoilTEX. Note, that these macros are completely device independent, hence the name of the style file. The macros are all in plain TEX form, so that they can

be used in any format or macro package. In other words, they are not LaTeX specific.

As for the color names, we used most of the names from the Crayola crayon box of 64 colors with a couple of additions and some deletions. Additions were the named Pantone colors not already included (e.g., RubineRed) and a couple of other well-rendered colors which we named ourselves (e.g., RoyalBlue). Deletions were mostly for colors that did not render well on our printers. In particular, the new fluorescent colors were eliminated. We chose these color names over, say, the X11 colors for a couple of reasons. First, we originally tried the X11 colors but they suffered from bad rendering on all devices tested. They just did not match their names (at least to me on my display or printer). Second, we could match the crayon names to the Pantone tables for a particular printer, and so give an approximate Pantone match to the color names as well as a good set of parameters. This information could (should?) be used at printer setup time to fine tune the parameters of the predefined colors to nearly match a Pantone (via special header files as mentioned above). In this way, output devices can be approximately calibrated to produce similar and expected results. The color names were also very descriptive of the actual color and very familiar to (at least) the North American TeXies. So, naive users have some idea of what to expect from certain color macros.

Tracking the Color Stack. The color stack or history is tracked by `dvips` in an internal structure. During the prescan which always goes from front to back on the dvi file, the color stack is tracked and a snapshot is taken at the beginning of every page. During the output pass, regardless of what pages are being processed, the driver knows the state of the stack at the beginning of every page. First one outputs both the background color (if necessary) and the top color on the color stack (i.e., the current color active at the beginning of the page) for the page being processed. Then color pushes just augment the color stack. Color pops just drop colors off the stack. Skipped pages are handled in the same way. This tracking keeps everything consistent from page to page.

The Remaining Issues. We have discussed almost every issue that was raised in the beginning. These included the simplicity of the macros themselves so they can work with any format, can be used by naive users with simple and generally recognizable names (Crayola crayons), still fully support arbitrary color models (if the driver can handle them), and their independence of the particular output data stream. We also discussed, in our implementation in `dvips`, how nesting and crossing of page boundaries are handled in a clean way. Furthermore, the implementation also can be easily customized for device-dependent differences, even within the same data stream.

The only remaining issue is how other structural problems, like margin paragraphs, headers and footers, itemize tags, floats and the like deal with color changes. Other than floats, these can be handled with simple redesign of the basic macros that deal with these page layout areas. Namely, they simply need to protect themselves with some local color macros. Unfortunately, most formats were written before this issue of color came up, so certain problems can arise. As far as I know, FoilTeX is the only format that has the color macros integrated into it. For example, the header and footer macros have their regions wrapped in a local color command that defaults to the root color of the document. So, for example, if the root color is blue, and there is some green text that gets split across page boundaries, then the text will resume green at the beginning of the next page and furthermore, the footer of the current page and header of the the next page will still be set in blue.

The mechanism described here is basically a hack to deal with these problems. A more structural approach at the driver level will be described by Tom Rokicki. At the user level, there is now some color support (e.g., `\normalcolor` and the color package by David Carlisle) in the new version of LaTeX that is designed to help deal with some of these problems in the context of existing drivers. It should be noted that the user interface in the color package is very different from the one we have presented.

Concluding Remarks

The story doesn't end here, of course. We don't claim to have solved all the problems (there are still many) nor to have provided the functionality that a professional publisher might want (refer to the paper by Michael Sofka on that point). The next era in the story is for Tom Rokicki to write (see his paper in this proceedings). Hopefully, Rokicki's new developments will provide a basis for a very powerful mechanism for setting color and one that can be easily integrated into plain TeX and the next generations of LaTeX (and other formats).

Acknowledgements

We thank Tom Rokicki for his comments on this paper as well as for his acceptance of support for color in `dvips` and his continued interest in the subject. Thanks also to Sebastian Rahtz for the invitation to participate in this session.

References

Hafner, James L., "FoilTeX, a LaTeX-like system for typesetting foils", *TUGboat*, **13** (3), pages 347–356, 1992.

Driver Support for Color in TeX: Proposal and Implementation

Tomas Gerhard Rokicki
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303
Internet: rokicki@cs.stanford.edu

Abstract

The advent of inexpensive medium-resolution color printing devices is creating an increasing demand for flexible and powerful color support in TeX. In this paper we discuss a new implementation of color support and propose an initial standard for color and color-like specials. We first discuss the difficulties that are presented to the driver writer in supporting color, and other features, by presenting a number of hard examples. Second, we present an implementation of a driver that provides a solution to many of the problems discussed. Best of all, this solution includes modular C code that is easily integrated into other drivers, automatically translating the higher-level special commands into existing low-level special commands.

Introduction

This paper has two parts: a collection of difficulties, and a proposed partial solution. The collection of difficulties is by far the easier part to write and to read; it is always easier to criticize than to originate. Nonetheless, it includes some subtle conclusions. The proposed solution does not come near to solving all of the problems raised in the first section, but it attempts to solve at least one, as one step towards a more general solution for the remaining ones.

Our perspective is that of a dvi driver writer. We care not for the user; let the macro programmers provide a convenient interface. Rather, we attempt to provide the primitive functionality from which specific effects can be accomplished.

For driver writers, on the other hand, we have untold sympathy. We will even do much of the work for them, by providing a set of C routines that implement the new functionality.

In order to understand why each problem is difficult, and what conclusions we can draw from each problem, we need to understand the limitations of TeX and of the various device drivers. While there is only one TeX, there are many different types of device drivers, each with its own requirements and capabilities. We can divide the drivers into four categories according to their style of operation.

The first kind is a driver that scans the entire dvi file (or at least up to the last required page) before generating any output. This prescan phase usually determines what fonts and what characters from each font need to be downloaded. This type of driver is typically necessary for laser printers.

The second type of driver does not perform this prescan phase, usually because the output device does not support downloaded fonts; this is typically the case for dot-matrix printers or FAX machines. This type of driver must render the entire page before shipping even the first row of pixels; it too buffers information, but at the page level instead of the document level.

Both of these types of drivers typically process the pages in the order they are given in the dvi file. A previewer, our third type of driver, does no such thing; instead, the pages are processed in some random order, and quick access to each page is desired.

The fourth and last type of driver we recognize is the driver that generates a dvi file as output. These include programs that do pagination tricks, like dvidvi and dviselect, and programs that expand virtual fonts, like dvicopy, and even the dvicolorsep program that does color separation.

Because TeX does not support color directly, we conclude that any such support must come through specials. Thus, the task of the device driver writer is two-fold: to recognize and parse the specials that direct his rendering, and to perform the rendering appropriately. This paper is primarily concerned with the first task. Color rendering and imaging is incredibly complex, so other than a few minor points, we shall not yet concern ourselves with these issues. Instead, we adopt the current

solution, as described in Dr. Hafner's paper in these proceedings.

Part One: The Problems

Now we are ready to present some sample difficulties and draw some conclusions from each.

Colored text and rules. Our first example is the most basic; we want to specify that some text or rules in our document be red. Because TEX does not allow us to attach color information directly to text or rules, this must be implemented as a change of state for our abstract rendering engine. Since we are using specials to implement colors, this change of state must occur at the point in the `dvi` file that the special is emitted. Therefore, *specials that indicate state changes must be used to implement colors.*

Even at this early stage, problems arise. It is not always obvious to the user where a special will be emitted. In general, it occurs in the same place in the linear stream of text that the user types, but occasionally this is not the case. Consider, for example, LATEX 2.09's `list` environment. Placing a special immediately after an `\item` command causes the special to occur in the `dvi` file before the bullet, coloring the bullet; this is not the intuitive result. (Technically, this happens because the special does not cause a switch to horizontal mode and is thus simply attached to the current vertical list; the bullet is inserted at the head of each paragraph, which starts with the switch to horizontal mode.) On the other hand, this can be considered simply a side-effect of the way the list environment is implemented; adding a `\leavevmode` command before the special command works around this difficulty. LATEX 2_ε solves this particular problem using color nesting, but similar problems can arise in other situations and with other macro packages.

If the state change occurs at the point at which the special occurs, then how shall we define the range of the color command? One alternative is to define the range to be that sequence of `dvi` commands enclosed between two specials. A second is to define it to be until the end of the enclosing TEX box. A third is to define it to be until the end of the enclosing TEX group. A fourth is to use some combination of these.

Unfortunately, the box solution fails in a number of ways. First, there is no real notion of boxes at the `dvi` level. Indeed, this can make it difficult to color a paragraph red—that paragraph might be split across several pages, and thus several boxes, with no overall enclosing box.

The first solution subsumes the third. Groups are not visible at the `dvi` level, but TEX's aftergroup command can be used to make specific groups visible. Therefore, *the range of color commands must be from special to special.*

Nested colors. The next question is whether to nest colors. In other words, should we be able to color a word red, without having to figure out and restore the color of the enclosing paragraph? Somehow it seems more consistent with TEX to allow nesting of colors, and in many situations, nesting colors solves some important problems. For instance, nesting is used in the previous version of color support in `dvips` and in the current version of LATEX 2_ε to allow headlines to work correctly. Certainly it is not hard to implement. Thus, *we should allow the nesting of colors.*

Should the driver be responsible for maintaining the color stack, or should the TEX macros? Either is easily implemented, and since the color stack should never nest deeply, the resources consumed by either should be negligible. If we use TEX, we can always make the current color available to the user of the macro package, provided that we standardize on some representation. On the other hand, we might not want to require that the color stack be provided by the macro package—and implementing a color stack is easy enough that we might as well provide one at the `dvi` driver level. Providing one at the driver level does not require the TEX macros to use it. In any case, backwards compatibility with the current color implementation requires a color stack. *The driver should implement color stacking, and some macro packages might also maintain the color stack for their own purposes.*

Should we also include a command to set the current color, independent of state changes? If we are using a set of simple macros that just set the color and ignore the stacking capability of the driver, this might cause the stack to get increasingly deep. And just issuing a pop stack command before each color command fails with the first color. Since it is a pretty easy feature to provide, we might as well. *The driver should implement non-stacked color changing.*

Colored text split across pages. Now imagine the word "example," in red, split across two pages. At the `dvi` level, the "begin red" special will occur near the end of one page, and the "end red" special will occur near the beginning of the next. Thus, `dvi` *drivers must maintain the color stack information across pages.*

In the context of page reversal, page selection, and random page access, this requires that the `dvi` driver store the contents of the color stack for each page it might need to revisit, and set up the output device state appropriately. This is not hard to implement once the requirement is understood.

Page break in colored region with black headline. There is a danger that a color region split across pages might also cause some headlines or footers to become contaminated with color. There is nothing in the `dvi` file indicating that some text is a headline or footline, so a straightforward nested color implementation will have this problem. The only real solution to this is to have the output routine put that information in the `dvi` file. Similar problems arise with footnotes, figures, and marginal notes. *The TEX output routine must indicate the origin of text in order for the color to be maintained correctly.*

Alternatively, the output routine can simply reset the color to black in regions such as headlines, footlines, marginal note, and floats; this is the solution currently adopted in LATEX 2ε.

Split footnote with colored regions. It might be desired to color headlines or marginal notes. Indeed, footnotes might have colored regions that are split across pages. A single page break might split both a pagebody colored region and a footnote colored region. Therefore, *the driver should actually maintain separate and independent color contexts, each with its own color stack, and the output routine should issue the necessary commands to switch among them.*

In the case of marginal notes, it may not be clear what the enclosing color context is. A marginal note might occur inside of a float or inside of a normal pagebody paragraph. Therefore, *the driver should maintain a stack of color contexts.*

Such contexts make it easy to do things like color all headers red; simply invoke the header context, push or set the color red, and then return to the previous context.

It is not clear how many different sources of text there might be, so the color stacks should be dynamically allocated by name inside the driver.

Footnotes within a colored region. Floats pose an interesting problem. If an entire section of a document is colored, should the included footnotes be colored as well? What happens if the floats move into the next section? As a logical consequence of the color context idea, they should (by default) not be colored, since they are from a different stream of text. On the other hand, to provide just this functionality if it is desired, it is easy to provide a global context that is always used for attributes not set in the current context. This global context will provide functionality backwards compatible with the current FoilTEX color model, and it will allow setting the color of entire regions of a document. On the other hand, it will not allow floats or footnotes that have portions on pages after the end of the color region to have the appropriate color; the color contexts must be used to obtain that effect. *A special "global" color context should be used as a default for parameters not set in the current context.*

To summarize, all stack push and pop commands affect the context on the top of the context stack; this is the current context. Colors (and other items) are always searched for first in the current context and then, if not found, in the global context.

An alternative, and perhaps preferable, implementation is to search in the current context, and then in the next context on the context stack, etc. This may be more natural, but it undoes the "defaulting" that we currently get if we set the pagebody to red and draw a marginal note. We believe this defaulting is more important, so we have implemented evaluation to only search the current and the global context, rather than all of the ones on the context stack.

Everything we have described so far is easy to implement. At the beginning of each page, we have a particular stack of contexts, which we save away in case we ever need to render that page again. In order to generate that data structure for a particular page, we must scan the `dvi` file from the front to that page. In other words, in the presence of color, it is no longer possible to read the `dvi` postamble and skip backwards on the previous page pointers in order to quickly find a page. On the other hand, the processing required to skip pages is negligible. *In order to properly render any page, all previous pages must be scanned.*

Because it is trivial to write out specials to set the stacks to any desired state, page reversal is also implementable. Indeed, it is easy to eliminate the stacks altogether using a `dvi` to `dvi` translator, thus allowing the use of simpler drivers, or translating the specials to a form recognized by a particular driver. The only trick is to use a syntax that allows the `dvi` to `dvi` program to easily distinguish those specials it must manipulate from those that it must leave alone.

Changebars. The color mechanism we have described will also help with tasks other than color.

For instance, changebars are also complicated by the asynchronous nature of TEX's output routine. Defining changebar on and changebar off to be color-type commands gives us the full nesting and state saving capabilities we used for color. Indeed, we can use the context switching commands to give us a vertical reference position, and define some changebar parameter to give a horizontal offset from that position, allowing dual-column changebars. This solves the problem of having changebars span inappropriate figures and not span appropriate ones.

The current implementation does not yet support changebars, but the author feels that the changes should be straightforward. Indeed, as with color, it is possible for a `dvi` to `dvi` program to convert a `dvi` file that specifies changebars into one that uses explicit rules. *Color and color context specials are appropriate for tasks other than color.*

Colored backgrounds. Another use of color, especially for slides, is in setting the current background color. Instead of modifying characters and rules between specials, this affects the entire page background before anything is drawn. There is no reason not to allow this to nest just like other color commands do, even though the primitives are at a different level. Thus, *we must be able to specify the background color.*

Colored background with headline on first page. Because of the way specials are sent out, headline text is emitted before any specials attached to the page contents. Thus, if the first page has a headline, that headline will occur in the `dvi` file before any page content such as specials. Therefore, the page global attribute values in effect at the beginning of a page, or before the first character or rule in the `dvi` file, might not be what is intended.

To solve this problem, we define that the page globals in effect at the *end* of the page are what define the values for the page background, orientation, or other page globals. This has two effects. The simple one is that page globals must be syntactically distinguishable from non-page-global color information. Indeed, this last requirement also allows us to distinguish a page-global rotation from a local rotation. *Page globals must be syntactically different from local attributes.*

A more important effect is that either pages must be fully prescanned before rendering can begin, or the driver must be prepared to restart the rendering of a page if a page global is encountered with different values from those currently in effect. Currently, many drivers prescan anyway. For those that do not, they cannot send out the first row of pixels until the entire page has been scanned anyway (a character at the top of the page might be the last character rendered in the `dvi` file), so rerendering when necessary is not terribly inconvenient. Thus, *to support page globals, pages must be prescanned or possibly rerendered.*

Paper size specification. One important page global is the specification of the paper size. Indeed, the lack of a standard for this information makes the driver's job much more difficult; knowing the job is intended for A4 paper can allow the driver to either request the appropriate paper, or shift or scale the page to fit. Certainly paper size is a typesetting-level and not a print-level option. Paper size should be specified as a page global on the first page. *The desired paper size should be specified in the* `dvi` *file.*

Imposition of pages with colored backgrounds or varying paper sizes. One function of `dvi` to `dvi` programs is page imposition—where pages are laid out in a specific order and orientation so that the folded signatures contain them in the proper order. When pages are imposed, the semantics of the page global options such as paper size and background color change slightly; this is simply a complexity that must be dealt with by the `dvi` to `dvi` program. It is possible to approximate some of these combinations using the appropriate `dvi` commands; for instance, page background commands can be converted into commands to draw a large background rule in the appropriate color.

Envelope/media selection. Page globals, such as paper size, might change in a particular job. For instance, many modern printers include an envelope tray; it would be convenient to have a media-selection page global that would allow a standard letter style to properly print the envelope, or select a sheet of letterhead for the first page of a long letter. *Drivers should support different paper sizes within a single document.*

Coloring the backgrounds of boxes. Occasionally a user might want to color the background of a particular TEX box. There are several problems with this. The first is that the box information simply is not available at the `dvi` level. The second is that the box dimensions tightly enclose the contents; does the user really intend to have the italic "f" protruding from the colored region? Finally, this is something that is easy to do at the TEX macro level by simply drawing a rule of the appropriate size

and color before setting the box. *Many things still should be implemented at the TeX level.*

Colored table backgrounds. One of the more common uses of color is to decorate the backgrounds of tables—each column gets a distinct shade or color. This is quite difficult to implement, although Timothy Van Zandt has had success with his `colortab.sty`. The primary difficulty is obtaining the column dimensions—height and width—before rendering the text of the columns. *Many common requirements still defy easy solution.*

Included graphics and other objects. It should also be possible to include graphics and do other rendering with specials, in the way they were intended. The main requirement is that *these types of specials be syntactically different from the color specials, so that* `dvi` *to* `dvi` *programs know which specials to manipulate and which to leave alone.*

As an aside, it is important that the mechanism for including graphics respect the `dvi` magnification and any rotation and scaling commands, so that imposition and scaling work correctly. In addition, it would be convenient to be able to easily calculate the size of the enclosing rectangle from just the special arguments so that, if nothing else, an outline can be drawn. *The* `dvi` *magnification should be respected in scaling graphics, and some standard for sizing/scaling included objects should be defined.*

And while we are off the topic, there is no excuse for not rendering PostScript graphics and fonts with previewers and non-PostScript drivers. The fine freely-available programs GhostScript and ps2pk do all the hard work of rendering for virtually any platform; a few dozen or hundred lines of interface code is usually all that is necessary for a fail-safe interface. *If you can't fully use PostScript in your TeX environment, it is time to complain.*

White on black. It is not necessary to wait for a color device to support color. Even black and white printers should support the two colors black and white, including being able to render white text on a black background. This is useful in itself and for color separations. *Even black and white devices need "color" support.*

Dithered text. When approximating gray text on the screen or to a low-resolution printer using dither patterns, the resulting image is often impossible to read. This is because the dither pattern sacrifices the high resolution needed to render characters for the ability to approximate gray levels.

In professional printing, spot colors, rather than four-color separations, are used to render small text and other single-color highlights. It is important to be able to specify what colors are intended to be spot colors. *We need a standard for specifying and using spot colors.*

For previewing or rendering on low-resolution printers, it is often useful to disable dithering for small fonts in order to end up with something that is readable.

Fountains. Another comment request is fountains. These are smooth graduations of color over an area. For instance, many slides are rendered with a background that is deep blue at the bottom and lighter blue at the top. A rainbow can also be considered a fountain. Fountains are normally approximated by drawing hundreds or thousands of narrow rules, each of a color midway between its neighbors. Whatever color model is chosen for TeX, it would be extremely nice to be able to render fountains.

There are many more examples, including clipping paths, character fountains, chokes, spreads, and the complexities of color vision, color rendering, and color models, that we will not address here.

Part Two: Some Solutions

This second part proposes a solution and implementation for some of the problems listed above. This implementation is used in both `dvips` and `dvidvi`, and the code is freely available to be used in any manner whatsoever.

First we will discuss a categorization of specials. Next, we will define a syntax, and finally, we will describe some keywords and what they mean.

Before we delve into the technical details, let us dispose of one objection: why not just introduce a little language for specials? In essence, that is essentially what we are doing; some might ask why not give it variables, types, and control flow as well. Of course, TeX is already a language; any processing that can be done at the special level is probably better and more portably done using TeX. Also, we would rather people spend their time learning a more practical language, such as PostScript. Indeed, some may consider what we are proposing as already unnecessarily complex—and they may be right.

With even a very simple language, implementing things such as change bars, colored table backgrounds, and much more would be straightforward.

We are not ready yet to define such a language, but we see it as an extension of what we propose here.

Syntax and parsing. Specials are case-sensitive. Words are defined as sequences of characters delimited by any of tabs, spaces, commas, equal signs, or open or close parentheses. If one of the delimiting characters is an equals sign, then the word on the left of the equals sign is associated with the word on the right.

The first word of the special is the keyword. The remainder of the special are its optional arguments.

If a double quote occurs, everything up until the next double quote is considered a single argument.

If a left quote occurs, the following argument is treated as a string without the left quote. If such an argument is opened as a file name, the argument is treated as a command to be executed, and the output from that command is read as the input from the file.

The types of words are string, number, and dimension. Strings or keywords are sequences of numbers, digits, or any character other than delimiters. Numbers consist of an optional negative sign followed by a sequence of digits, optional decimal point and additional sequence of digits. Dimensions are numbers, followed by an optional `true`, followed by one of `in`, `pt`, `bp`, `dd`, `cm`, or `mm`. They are interpreted exactly as in TeX.

Categories of specials. We divide specials into five categories: context switches, foreground state changes, background state changes, document globals, and objects.

1. Context switches push and pop contexts onto the context stack by name. If the context named does not exist, it is created. The default context at startup is `global`.

   ```
   context <push/pop> <name>
   ```

2. Foreground state changes set, push, or pop a foreground state item, such as a color.

   ```
   attribute <push/pop/set> <name>
      [<value>]*
   ```

3. Background state changes set, push, or pop a background state item, such as a background color or paper type.

   ```
   attribute <push/pop/set> page
      <name> [<value>]*
   ```

4. Document globals set some resource requirement or provide some other information. These specials must always occur somewhere on the first page.

   ```
   attribute <push/pop/set> document
      <name> [<value>]*
   ```

5. Objects are everything else, including snippets of PostScript code and included graphics.

   ```
   psfile=foo.ps llx=72 lly=72
      urx=452 ury=930 rwi=500
   ```

With the above syntax, it is easy to syntactically identify the type of a special without needing to understand the specific instances.

Interpretation. We have introduced the idea of a `dvi` color context that can be saved and restored in a non-nested fashion. We allocate contexts dynamically as they are encountered; a macro package might define one for each of footnotes, pagebody, figures, headers, marginal notes, and global. The output routine will then issue the appropriate 'switch context' commands at each point.

```
context push header
<header stuff>
context pop
context push pagebody
<pagebody>
context push figure
<figure>
context pop
<more pagebody>
context push margin-note
<margin note>
context pop
<more pagebody>
context pop
```

Default values for attributes are more troublesome. Consider a document that, on page ten, sets a specific special attribute woomp to the value there-it-is, and this value remains set for the rest of the document. If this document is reversed, the set would then occur at the beginning of the new document—but something must be done to undo the special at the place where page ten now occurs.

The solution is straightforward. If a context stack does not have an entry for a particular attribute when a set occurs, the set is interpreted as a push; otherwise, the set is interpreted as a pop followed by a push. Thus, for a flat sequence of sets, the first will allocate an entry on the context stack for the attribute, and all others will modify that attribute. If it becomes necessary to reset an attribute to its default value, a pop will suffice.

The following implementation effectively flattens all contexts into a simple sequence of set attributes and pops. Pops are only issued to reset

attributes to their defaults; there are no corresponding pushes except the implicit ones introduced by the sets.

Thus, with the provided C code, it is trivial to integrate color contexts into an existing driver.

Implementation. Implementing these specials is straightforward. The key idea is that we need to maintain the stack states for each page and restore them appropriately. In addition, an implementation can choose between always prescanning the first time a page is encountered, either on a page or document basis, or possibly re-rendering the page if it should turn out to be necessary. Our implementation supports both possibilities.

Essentially, the code provided flattens all context specials and attribute settings to a simple sequence of attribute sets and pops. All page specials are moved to the very beginning of a page, and all document specials are moved to the very beginning of a document. The `dvidvi` program provided does this from the provided `dvi` file; for all other drivers, this special translation and movement happens dynamically.

When a new `dvi` file is started, the driver is responsible for calling `initcontexts()` to initialize the various data structures. At the beginning and end of each page, the driver should call `bopcontexts()` and `eopcontexts()`. These need not come in matched pairs; if page rendering is interrupted for any reason (such as the user selecting the next page before rendering is completed) the driver must not call `eopcontexts()` but should instead simply call `bopcontexts()` for the next page.

The exception to this, of course, is that each page must be fully scanned at least once, and `eopcontexts()` called, before any subsequent page can be rendered.

The driver must provide the subroutine `dospecial()` that is responsible for parsing and understanding specials in the normal manner. Typically this already exists in almost all drivers. But rather than calling this subroutine every time a special is encountered, the driver should instead call the supplied routine `contextspecial()`. This subroutine will check if the special is one of the context specials described here, and if so, translate it to the appropriate flat specials, calling `dospecial()` for each one. If the special is not a context special, then the driver's `dospecial()` routine is invoked.

If the special was a page special or a document special, and this is the first time this page has been encountered, `contextspecial()` will return the special value `RERENDER` indicating that the driver should consider rerendering the page from the beginning after performing (finishing) a prescan. If the driver has not yet rendered any characters or rules, or if the driver is scanning rather than rendering, this return code can be ignored.

To identify pages, the driver should also provide a routine called `dviloc()` that returns a long value indicating the byte position in the `dvi` file.

The call to `bopcontexts()` at the beginning of a page may cause the driver's `dospecial()` routine to be invoked many times, once for every outstanding page attribute and local attribute.

To handle document global specials, the entire first page must always be fully prescanned.

The way the code works is as follows. At the beginning of each page that has not been previously encountered, the full stack contents of each context are saved and associated with the `dvi` file location for that page. If the page has been encountered, then the stack contents are restored, issuing any necessary set attribute specials for current attributes in the global context. In addition, any page attribute values are set. The context stack is set to hold just the global context.

When a push context special is encountered, the context associated with that name is found. If none exists, one is allocated. If the context stack has more than just the global context, then the attribute values from the context on top of the context stack are hidden. In any case, the attribute values for the context being pushed are made visible.

Attribute values are hidden by searching for the same attribute in the global context. If one exists, then its value is emitted with a flat set attribute special. Otherwise, the value is reset with a pop attribute special.

Attribute values are made visible by simply executing a flat set attribute special for each value.

When a pop context special is encountered, the context stack is checked to make sure it has at least two entries. If not, an error routine is called. Otherwise, the top context is popped, and all attribute values in that context are hidden. If the resulting context stack has more than just one context, then the attributes in that context are made visible.

When an attribute push special is encountered, then the attribute name and value pair are added to the current context, and the new value is made visible.

When an attribute set special is encountered, if the context on top of the context stack has such an attribute, than that attribute is changed and the new value made visible. Otherwise, the set attribute

special is treated precisely as though it were a push attribute special.

When an attribute pop special is encountered, the context on top of the context stack is searched for that attribute. If the context has no such attribute, an error is reported. Otherwise, the attribute is hidden, and the attribute/value pair is popped from the context. If the same attribute exists in the current context (further down on the stack), then that attribute value is made visible.

Note that attributes do not need to nest "correctly"; the following sequence is legal:

```
attribute push color red
   <text>
attribute push changebar on
   <text>
attribute pop color
   <text>
attribute pop changebar
```

In addition, pushing and popping contexts simply makes them visible and hidden; it does not affect their values. Thus, assuming that the global context is on the context stack, after the following sequence, the color in the global context will be green:

```
attribute push color red
context push header
context push global
attribute set color green
context pop
context pop
```

Backwards compatibility. For backwards compatibility, existing `dvips` specials are fully supported. Most specials fall into the object category and are automatically passed through to `dospecial()`. These specials include those for EPSF inclusion and literal PostScript code.

The existing color macros are trivially supported by translation. The existing color macros never change contexts (they always use the implicit global context), so the semantics are unchanged with one exception. The explicit color set macro is now legal even when there are colors on the color stack; only the topmost entry on the stack is affected.

The four specials `header`, `papersize`, `landscape`, and `!` are document global specials and are translated as such. The next release of `dvips` will also allow `papersize` and `landscape` specials to apply on a page basis.

The code implementing these color specials, along with documentation describing how to use the code, is available in both `dvips` and `dvidvi` on `labrea.stanford.edu`.

Future work. We plan to continue the development of special capabilities using this form of interface. In particular, we hope to add support for colored box backgrounds, changebars, and similar things through a simple language. As we or others enhance the released code, any drivers that use this will automatically get the new capabilities. And, the `dvidvi` program will provide full support for these specials for those drivers that don't use the code.

Acknowledgments. The ideas in this paper are primarily derived from discussions with James Hafner, David Carlisle, Leslie Lamport, Frank Mittelbach, Sebastian Rahtz, and Tim Van Zandt. The confusion and complexity is attributable to me. I can only hope that this code will evolve quickly and stabilize into a useful and powerful base for using color in TEX.

DVISep — A colour separator for DVI files

Angus Duggan
Harlequin Ltd., Barrington Hall, Barrington, Cambridge CB2 5RG, United Kingdom
angus@harlequin.co.uk

Abstract

This paper describes a simple colour separator for DVI files written by the author, and explores the implementation and limitations of such a colour separator. The colour separator recognises the colour support `\special` commands used by `dvips`. It produces multiple DVI files as output from a single input file, each containing a single separation from the input colour.

Introduction

DVISep is a simple colour separation program for DVI files. It reads a single DVI file, outputting separation files for each process colour and spot colour found in the input file. DVISep recognises the colour `\special` commands used by Tomas Rokicki's `dvips` driver, but is not limited to use with `dvips`. DVISep is part of the second release of the author's DVIUtils package of DVI manipulation programs.

Colour printing presses and printers normally use four *process* colours (cyan, magenta, yellow and black) printed on top of each other to create the illusion of many colours. Cyan ink absorbs red light, reflecting only blue and green. Magenta ink absorbs green light, reflecting red and blue, and yellow ink absorbs blue light, reflecting red and green. Black is also used as a process colour because the cyan, magenta, and yellow inks do not have perfect absorbtion, and a combination of solid cyan, magenta and yellow usually looks a muddy brown colour instead of truly black.

Colour separation is the process of splitting the image into appropriate proportions of the process colours for the colours desired. DVISep generates separate files for each of the process colours, containing only the parts of the image with the appropriate colours in them.

In addition to process colours, printed pages may use *spot* colours. Spot colours are used in several circumstances; when there are only a couple of colours in a document, it may be cheaper to print it with spot colours rather process colours; special inks (*e.g.* textured, metallic, neon colours or luminescent colour) are required sometimes; and sometimes it is necessary to provide an exact colour match, for instance in a paint or ink catalogue.

Having decided which colours to use, there are still different ways of combining those colours on the page. Each object on the page may *knockout* or *overprint* other objects. Knockouts cause blank areas to appear on other separations; this may be useful when an ink should not be combined with other ink on the page; for instance, a spot colour should probably not be printed on top of a process colour background.

Figures 1 and 2 illustrate the difference between overprinting and knockout. If knockouts are set, the shapes drawn last erase the corresponding areas of shapes drawn before them; if overprinting is used, the colours of previously drawn shapes are combined with the latterly drawn shapes.

Using DVISep

The simplest use of DVISep takes an input filename:

```
dvisep file.dvi
```

This generates the output files `Black.dvi`, `Cyan.dvi`, `Magenta.dvi`, `Yellow.dvi`, and *spot*`.dvi` for each spot colour used in the input file[1].

The `\special` commands currently recognised by DVISep are a subset of those defined by Tomas Rokicki's `dvips` driver. The `\special` commands start with either `color` or `background`, followed by a colour specification. The colour specification may be either a colour name, for example `Maroon`, or the name of a colour space (`rgb`, `hsb`, `cmyk`, or `gray`) followed by an appropriate number of numeric parameters (3 for `rgb` and `hsb`, 4 for `cmyk`, or 1 for `gray`). The numeric parameters are all within the range 0–1, indicating the intensity of each colour component. Note that for additive colour spaces (`gray`, `rgb`, and `hsb`) zero values of the parameters indicate that no light should be reflected from the page (*i.e.*, the page is marked black), whereas for subtractive colour spaces (`cmyk`) zero values indicate that no light should be absorbed by the page (*i.e.*, the page is left white). The

[1] On systems with restricted filename length the spot colour names are reduced using a heuristic rule which attempts to create a recognisable name. DVISep does not overwrite other files of the same name unless explicitly told to, so other files which accidentally match the reduced spot colour name will be preserved.

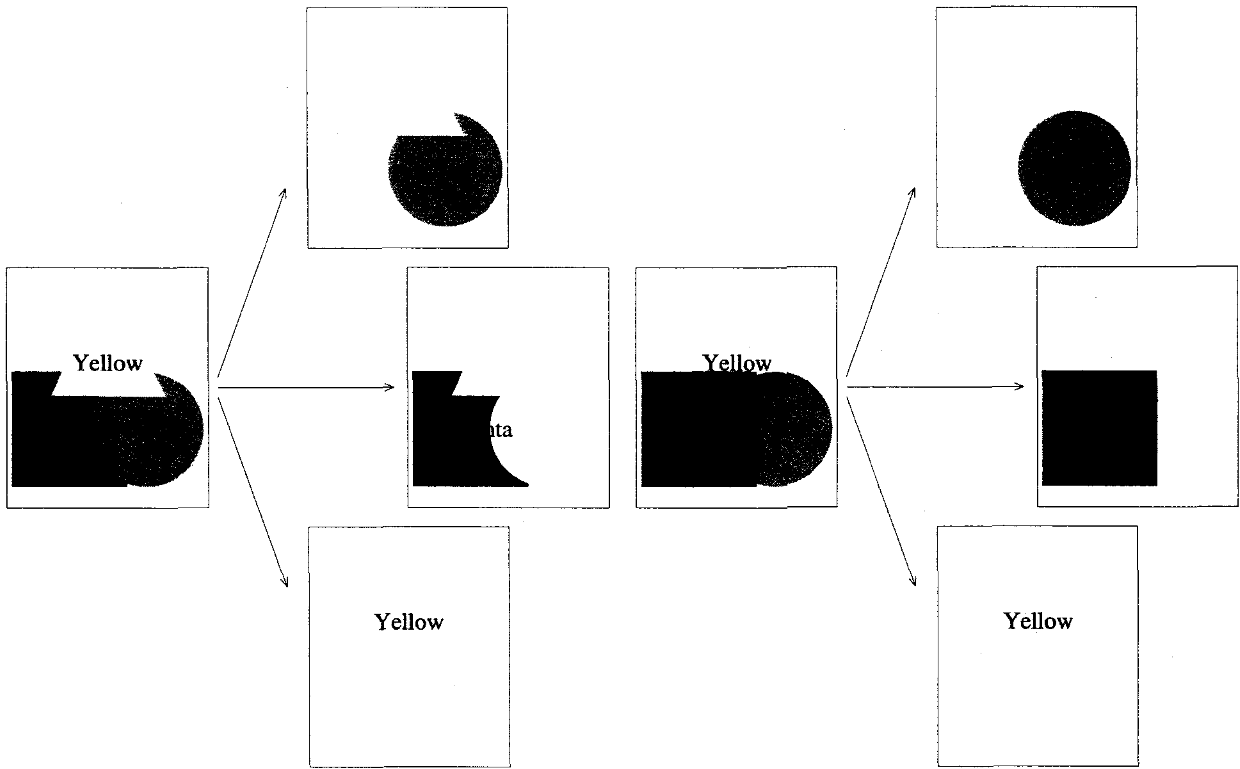


Figure 1: Colour separation with knockout

Figure 2: Colour separation with overprinting

double quote form of colour specification supported by `dvips` is not supported by DVISep.

A second form of `\special` command supported is `color push` followed by a colour specification; this saves the current colour and sets the curent colour to the new colour. The command `color pop` sets the current colour to the colour last saved with a `color push` command

Colours specified by a colour space and parameters are converted to CMYK process colours before use. There is an option to DVISep which affects this conversion process. The `-u` flag turns on undercolour removal and black generation. RGB and HSB colours are initially converted to CMY colours. Undercolour removal removes an equal amount of each of the CMY components, and black generation adds that amount to the black component, yielding a CMYK colour.

Colours specified by names are looked up in a colour definition table; DVISep reads a default colour definition file when it initialises, which tells it the name, colour space, and parameters for named colours. Spot colours are also defined by the colour specification files. The format of these colour specification files is very simple — each line begins with a colour name, followed by either a colour space and numeric parameters, or the keyword `spot` and a single numeric parameter for spot colours. Spot colours may also have one of the optional keywords `overprint` or `knockout` at the end of the line. Comment lines in the colour specification file are indicated by '#' as the first non-blank character on the line. Extra colour specification files can be loaded by passing the `-c` *file* option to DVISep. If a colour name in an extra colour specification file matches an existing name, the specification in the extra file is used. More than one extra colour specification files can be loaded by using several instances of the `-c` option.

The `-o` and `-k` options to DVISep indicate whether it is to overprint or knockout process colours; knockouts for process colours are the default. Whether spot colours overprint or knockout is determined by the colour specification file; they will normally knockout unless the colour specification contains the `overprint` keyword.

How DVISep works

Upon starting, DVISep reads a default colour specification file, and then processes its arguments, which may result in it reading more colour specification files. Colours specified in RGB or HSB space are transformed into CMY space when they are used, and then into CMYK through undercolour removal and black generation. The transformation from RGB and HSB

to CMY is very simplistic; the algorithms in Foley, van Dam, Feiner and Hughes (Foley et al.1990) are used. If undercolour removal and black generation are turned on, the amount of black generated will be equal to the minimum value of the other colour components.

In general, colour transformation is more complicated than this; the representable range of colours on devices differs, depending on the printing process and inks used. The same colour values will produce noticeably different results not only on different devices, but also on different calibrations of the same device type and on different printing surfaces. This problem is beyond the scope of DVISep — a colour matching system is required to solve these problems.

DVISep makes several passes over the input DVI file. The first pass is used to build up a structure containing information about the pages in the file. The start and end of each page is noted, and lists of the spot colours and fonts used are generated. DVISep then makes one pass over the input file for each process colour and spot colour, to write out the separation files. On each output pass, the initial colour is set to black, and a colour stack is maintained by noting the `color push` and `color pop` special commands. As each page is scanned, DVISep looks for colour changing `\special`s and commands that mark the page (characters and rules). The action taken for page-marking commands depends on whether the current colour is a process colour or a spot colour, and whether the colour is overprinting or knocking out other colours. The name of the separation files is generated from the process and spot colour names. DVISep avoids writing out pages which do not contain any ink.

Process colour handling. When page-marking commands are found during process-colour separating, the current colour is examined to see if it is a process colour. If the current colour is a process colour with a non-zero component of the separation colour being generated, a colour support `\special` command will be issued which sets the colour to a shade of gray corresponding to the amount of the component present. For example, if the CMYK colour (0.87 0.68 0.32 0) is specified in the input file, a command to set the gray level to 0.13 will be issued for the cyan separation (remember that the gray parameters are inverted with respect to CMYK parameters), a command for a gray level of 0.32 will be issued for the magenta separation, and a command for a gray level of 0.68 will be issued for the yellow separation. The colour command will only be issued before the first page-marking command after each colour change.

If the current colour is a process colour with a zero component of the separation colour, or a spot colour, then DVISep needs to decide whether to knockout existing objects on the page. If the knockout flag is set for process colours, or the spot colour was specified with the `knockout` keyword (or no extra keyword at all), then a command is issued to set the current colour to white, and the page-marking commands are written to the separation file. Knockouts done in this way will only work if the imaging model of the output device is a paint-and-stencil model like PostScript[2], where white areas painted over black areas erase them. If this imaging model is not assumed, knockouts have to be done by determining the difference of the shapes of the objects printed in the separation colour and the objects which are knocked out. The results of this process can not be represented in DVI format without generating custom fonts for the output resolution.

If the knockout flag for process colours is not set, or a spot colour is specified with the `overwrite` flag, then the page-marking commands up to the next change of colour are ignored. This allows different separations to have page-marking commands at the same position on the page, causing the inks to overprint and combine when printed. The page marking commands cannot be completely thrown away, however, because character and rule setting commands may move the current horizontal position. If characters are being ignored, DVISep reads and caches the TFM (TEX font metric) file for the current font, moving the current position right by the width of the character.

Spot colour handling. Spot colour separations are handled in a similar way to process colours, except that only one component is considered when deciding if the colour should be printed, knocked out, or ignored. The numeric parameter to the colour specification of a spot colour is a tint value, which indicates how much of the colourant should be applied, and hence the gray level for the following page-marking commands.

Background handling. The `background` colour command needs some special handling in DVISep. The last background command issued on each page is the one which takes effect (this is Rokicki's definition of the `background` special command), so this information must be stored in the page information during the initial scan. The background colour is treated much like other colours; if it is a process colour, then appropriate background commands will be set for each separation file, depending upon the amount of the separation component in the background colour. If the background colour is a spot colour, then the background command will only appear in the spot colour separation file.

[2] PostScript is a trademark of Adobe Systems Incorporated

Using DVISep

DVISep normally takes a single input DVI file, and produces a DVI file for each process and spot colour used in a document. There are several other options which control DVISep's behaviour.

The -c, -k, -o, and -u flags have already been mentioned; the first flag names additional colour specification files to read, the next two flags indicate whether DVISep is to knockout or overprint process colours, and the last flag indicates whether to apply undercolour removal and black generation to RGB and HSB colours when converting to CMYK.

The -s *colour* and -p options are often used together; -s selects a single separation to output, and -p causes this separation to be written to standard output. This allows separations to be filtered through other programs from within scripts, without having to know the filename that DVISep would create for the separation colour.

Normally DVISep will not overwrite existing files; the -f option forces it to do so.

The usual DVIUtils options of -v for version information and -q for quiet running also apply.

Using DVISep with dvips. DVISep can be used with any device driver which supports Rokicki's colour \special commands. Input CMYK, RGB, and HSB colours (and named colours which are specified in these colour spaces) are converted to explicit gray scale commands.

It is important that drivers should know which separation is being output, so that halftone screens can be generated at the correct angles. There is no standard method of communicating this information to the driver; DVISep issues a new \special command to indicate the separation. This special command has the keyword separation followed by the separation name.

If the output of DVISep is being output through dvips, header files should be used to set up the separation screen angles. Process colour screens are traditionally at 15° for cyan, 75° for magenta, 0° for yellow, and 45° for black. The eye is very good at picking out vertical and horizontal features, which is why the least noticeable colour (yellow) has the angle nearest to orthogonal.

A header file that sets the screen angle to 15° without changing the screen frequency or spot function[3] might be defined as:

```
%!
%%DocumentProcessColors: Cyan
currentscreen exch pop 15 exch setscreen
```

[3] Some high-end PostScript RIPs can be configured to ignore user settings of frequency and spot functions, because the user's settings are not always appropriate to the final output device.

The double quote form of colour specification is not supported by DVISep, because its parameter (an arbitrary PostScript string) does not give easily usable information about what colour is required.

DVISep supports an extended form of colour \special command, which allows spot colours to be specified from within TEX. This command takes the form spot *plate tint*, indicating the separation plate on which the colour is to appear, and its intensity. This colour command is not directly compatible with dvips; if the document is to be processed by dvips before separating (*e.g.*, for previewing) then spot colours should be specified by named colours, with a process colour approximation to the spot colour in the color.pro header file.

Limitations

DVISep has some limitations, which need to be borne in mind when using it. One of these limitations has been mentioned already; knockouts assume that printing in white can erase areas already printed in other colours. This limitation may be a problem for output drivers for a lot of devices, but fortunately not for PostScript drivers.

A more serious limitation from the user's point of view is that DVISep does not currently handle PostScript inclusions at all. There are reasons for this omission;

1. The \special commands used to insert PostScript code are output driver dependent.
2. The PostScript inclusion may reset the current colour. To a certain extent, this problem can be alleviated by putting the inclusion into each separation and redefining the colour-setting operators or transfer functions to set the colour to white for colours which should not appear on the current separation. This is not a total solution, because the inclusion can still access the original colour operators (and many applications produce PostScript which does this), and because overprinting cannot be done with inclusions.

Both of these problems are insurmountable in the general case; there is no way of hiding the original colour operators completely, because the dictionary in which they reside is read-only, and itself is impossible to hide. The second problem is insurmountable because PostScript does not have the concept of transparency, except in the limited case of image masks. Sampled images may contain a mix of colours, and there is no facility for making some of the image pixels overwrite existing objects on the page, and making others leave the existing page untouched.

DVISep does not yet provide registration marks for aligning pre-separated plates, or taglines for identifying separations. I hope to add these soon.

DVISep does not attempt to trap the separations at all (trapping is used to reduce registration errors, by enlarging or reducing areas painted on different separations so that they have a very small overlap).

Conclusions

Colour separation is not necessary for many printers, especially desktop printers, which use their own colour rendering techniques to print continuous tone data. Separation is necessary when going to press with higher resolution work. Separating the DVI file is quite easy, but some assumptions have to be made about the imaging model which will be used. If film or plates are being produced, separating the DVI file may save time and effort, by removing the empty pages from the separation files before printing.

The second release of the DVIUtils programs (including DVISep) will be available for anonymous FTP.

References

Foley, James D, Andries van Dam, Steven K. Feiner, and John F Hughes. *Computer Graphics, Principles and Practice.* Addison-Wesley, Reading, MA, USA, second edition, 1990.

Simple colour design, and colour in $\LaTeX2_\varepsilon$

Sebastian Rahtz
ArchaeoInformatica, 12 Cygnet Street, York Y02 1AG, United Kingdom
sebastian.rahtz@cl.cam.ac.uk

Michel Goossens
CERN, CN Division, CH1211 Geneva 23, Switzerland
michel.goossens@cern.ch

Abstract

This article reviews some basic principles underlying the use of colour. We start by a review of the functional use of colour, explaining how it can help to focus attention, explain relationships, guide the viewer/reader through the presented information so that its contents is easier to absorb and appreciate. Some common rules for optimizing communication using colour elements in documents are discussed. We then explain the colour support in $\LaTeX2_\varepsilon$ and give some examples.

Introduction

When considering the use of colour in a document, we should think about it as a tool, not a gadget to merely make the page look "pretty". The painter Eugène Delacroix wrote

> "La couleur est par excellence la partie de l'art qui détient le don magique. Alors que le sujet, la forme, la ligne s'adressent d'abord à la pensée, la couleur n'a aucun sens pour l'intelligence, mais elle a tous les pouvoirs sur la sensibilité, elle remue des sentiments."

This sentence summarizes perfectly the rôle that colour plays in the construction of the visual image. By choosing the right colour, the typographer can add an affective value to the message, thus helping it to be understood more clearly.

The world of colour

The Greek philosopher Aristotle had already, in the 4th century B.C., studied the mixing of colours by letting daylight shine through glasses of different colours. But it was only in the 17th century, thanks to experiments with glass prisms by Sir Isaac Newton, that the spectral theory of light was discovered, thus ending a period of almost 2200 years in which colours were ordered on a straight line from the lightest to the darkest colour, starting with white and ending with black. Newton ordered the colours on a closed circular ring, a representation still in use today[1].

Light can be decomposed into three "primary" components, from which one can build all possible colours. On a cathode ray tube, these colours are red, green and blue, and one of the more popular colour models is therefore called the **RGB** model. The printing industry does not use these primary colours, but rather their complements: cyan, magenta and yellow. This is because inks "subtract" their supplementary colours from the white light which falls on the surface, e.g., cyan ink absorbs the red component of white light, and thus, in terms of the additive primaries, cyan is white minus red, i.e., blue plus green. Similarly, magenta absorbs the green component and corresponds to red plus blue, while yellow, which absorbs blue, is red plus green. In fact, for practical purposes in the printing industry a process called "undercolour removal" takes places. In this procedure a fourth "colour", black, is added to the printing process, with an intensity equal to the equal amount of cyan, magenta and yellow present in the sample. This way one creates a darker black than is possible by mixing the three coloured inks. This colour model is called the **CMYK** model, where the final "K" stands for the black component. Color Example 1 gives a simplified view of the relation between the RGB and CMYK models.

Colour harmony

Harmonies are arrangements of colour that are pleasing to the eye. Scores of books giving the opinions of experts have been written on colour harmony, and the conclusions of many of these works are often

[1] Gerritsen (1988) gives an overview of the theory of colour from antiquity to the present. He reviews several models that have been proposed over the years in order to classify colours. He shows that choosing a suitable model depends clearly on the application area, e.g., mixing properties, human perception, hue values, gray levels.

contradictory. Reasons to explain this are not hard to find (Judd and Wyszecki 1963):

(a) Colour harmony is a matter of emotional response, likes and dislikes, and even the same person can change her/his mind over time, since old combinations can become boring, while frequent exposure to some new combination can make us appreciate it.

(b) Colour harmony depends on the absolute size of the areas covered by the colours as well as on the design and the colours themselves. For instance, a nice looking mosaic pattern can look completely unattractive when viewed magnified by a factor of ten.

(c) Colour harmony depends on the relative sizes of the areas as well as on their colours.

(d) Colour harmony depends on the shape of the elements as well as on their colours.

(e) Colour harmony depends on the meaning or interpretation of the design as well as on their colours. It is important to note that colour harmony for a portrait painter is quite a different subject from colour harmony in abstract design or typography.

Therefore one can only try and formulate a few principles for the construction of colour harmonies.

(a) Colour harmony results from the juxtaposition of colours selected to an orderly plan that can be recognized and emotionally appreciated.

(b) When comparing two similar sequences of colour, the observer will choose the one most familiar as the most harmonious.

(c) Groups of colours that appear to have a common aspect or quality are considered to be harmonious.

(d) Colours are perceived as harmonious only if the combination of colours has a plan of selection which is unambiguously recognizable.

Experimentally it has been observed that the eye prefers combinations where primary colours are in equilibrium with their complementary colours, and that our perception of a colour changes in relation to the environment in which it is embedded. Color Example 3 shows effects of *saturation* or *absorption* of the three primary colours with respect to white (leftmost column) or black (second column) and with respect to its complementary colour (third column) or a gray tone of the primary colour itself (rightmost column).

Constructing colour harmonies

To explain his theory of colours Itten, in his book *The Art of Colour* (Itten 1974), uses a model based on a harmonic colour circle subdivided into twelve equal parts (see Color Example 2). It contains the three primary colours yellow, red, and blue, 120° apart. Their complementary colours, purple, green, and orange, also called the secondary colours, are positioned diametrically opposite their respective primaries. The circle contains six more colours, intermediate between each primary and its adjacent secondaries. The harmonic colour circle is only a simplification. Indeed, all possible colours can be represented on the surface of a sphere, which has the harmonic colours at its equator, white at the north pole, and black at the south pole. Thus moving from the equator towards the south, respectively north pole yields darker, respectively lighter variants of a given colour. This also means that to each point on the colour sphere, there exists a diametrically opposed point with complementary characteristics, e.g., to light greenish blue one opposes dark orange red. Century long artistic experience has shown that a few simple basic rules allow artists to construct effective colour harmonies in their works. Following Itten, we shall discuss a few of them below.

Two-colour harmonic combinations. Complementary colours, lying at diametrically opposite points of the colour circle (sphere) define 2-colour harmonies, like the 2-tuples (red, green), (blue, orange), plus an almost infinite amount constructed using possible combinations on the sphere.

Three-colour harmonic combinations. When inside the colour circle one constructs an equilateral triangle, the colours at each edge form a 3-colour harmony. The most fundamental 3-tuple (yellow, red, blue) is well know in all forms of art, publishing, and the world of publicity, for its effectiveness, since it can be used in a wide variety of patterns, layouts, and in all kinds of light and dark combinations. The secondary colour 3-tuple (purple, green, orange) has also a strong character, and is often used. Other 3-tuples are also possible. One can also construct other 3-tuples by replacing the equilateral triangle by an isosceles one, or by working on the colour sphere and combining light and dark variants. As a special case, one can put one edge of the triangle at the white point (north pole), and create the harmony (white, dark greenish blue, dark orange red), or on the black point (south pole), yielding the harmony (black, light greenish blue, light orange red).

Four-colour harmonic combinations. One can construct a 4-colour harmony by taking the colours lying on the edges of a square, e.g., the 4-tuple (yellow, orange-red, purple, greenish blue). It is also possible to use a rectangle, combining two pairs of complementary colours.

Higher order harmonies (like six-colour) are equally easy to obtain using similar geometric models, by using the harmonic circle or the colour sphere. Note, however, that each combination has its own

character, and set of basic laws, and only a long experience will show which of the various sets of harmonies is most efficient for a given application.

Colour and readability

The readability of a message or sign is closely linked to how our visual system processes the information presented to it. Factors which influence the visibility of colours are:

(a) *intensity:* pure colours of the spectrum have the highest intensity;
(b) *contrast:* between the different colours;
(c) *purity:* pure colours are more visible than graded variants, where white is added, making them fainter, or black, making them darker.

Color Example 4 shows some of the most effective colour contrasts, which can be used for maximum readability or visibility, e.g., on slides, road signs, or publicity leaflets.

Colour in the printing industry

A detailed discussion of problems encountered when producing books in colour with TEX can be found in Michael Sofka's article in these proceedings. In this section, we merely present a short overview.

As already mentioned, the printing industry mostly uses the CMYK model to describe the colours on a page. Goossens and Rahtz (1994b, page 7) contains an example with the five Olympic rings and a multi-colour ellipse. It is shown how applying successively the coloured inks gives the picture its final colour. One starts with the cyan ink (top left), then applies the magenta (top right), yellow (bottom left), and finally the black inks (bottom right), to obtain the picture in full colour. The process is shown with the four separate stages, and the cumulative effect.

For high quality typeset output, the use of PostScript is now almost universal, and level 2 of the PostScript language offers full support for colour. In fact it not merely supports the RGB and CMYK models, but also the HSB (*Hue Saturation Brightness*), CIE (*Commission Internationale de l'Éclairage* standard) plus various special colour spaces; in industry and arts the Munsell and Pantone, and more recently the Focoltone and Trumatch systems, have become common for colour matching. The details of these, and algorithms for converting between colour models, are exhaustively discussed in Adobe Systems (1991, pp.176–199). Very useful discussions of using colour in PostScript are given in Kunkel (1990) and McGilton and Campione (1992), and our discussion is based largely on what we have learnt from these three books, and Reid (1986). It should be noted that full Level 2 PostScript provides a number of important new commands which considerably ease preparation of colour separations (see section 'Simple colour separation using `dvips`' on page 222 below).

Using colours with LaTeX2ε

With the release of LaTeX2ε, colour macros are now a standard supported package (together with graphics file inclusion, rotation, and scaling). These are, of course, dependent on the abilities of the driver in use, as all colour is done using `\special` commands. Hafner and Rokicki's `colordvi` package (see Hafner, this volume) was the first to try and address some of the complexities of colour support — TEX does not have internal support for colour attributes of text, and TEX 'grouping' across pages, floats, footnotes etc will not always yield the expected results. LaTeX2ε has extended support to cope with most situations, and it is hoped that more driver support will make this even better. Tomas Rokicki's paper in this volume discusses the problems in more detail. The LaTeX2ε colour package builds on the experience of `colordvi`, both in terms of the `\special` commands themselves (allowing for an extensible set of colour models), and in the macros.

One of the important features inherited from `colordvi` is the use of a layer of colour 'names' above the actual specification given to the printer; Hafner worked out a set of 68 CMYK colours which correspond to a common set of Crayola crayons, and these are predefined in the header files used by `dvips`, and the user calls them *by name*, allowing for tuning of the header files for a particular printer. New colours desired by the user can, of course, be defined using CMYK, RGB or other colour models, but in our examples we will use the Crayola names.

The LaTeX2ε colour support offers a variety of facilities for:

- colouring text;
- colouring box backgrounds;
- setting the page colour;
- defining new colour names

We will look at the interface, and then consider some uses for them.

The new LaTeX2ε commands There are two types of text colour commands, which correspond to the normal font-changing macros. The first one is a *command*:

`\textcolor{`*colourname*`}{`*text*`}`

This takes an argument enclosed in brackets and writes it in the selected colour. This should be used for local or nested colour changes, since it restores the original colour state when it is completed, e.g.,

```
This will be in black
\textcolor{Blue}{This text will be blue}
and this reverts to black
```

The second type of colour command is of the form:

```
\color{colourname}
```

This colour command takes only one argument and simply sets a new colour at this point. No previous colour information is saved, e.g.,

```
\color{Red} All the following text
        will be red.
\color{Black} Set the text colour
        to black again.
```

The command does however respect normal TeX grouping; if we write

```
We start in black, but now
{\color{red} all text
is in red, {\color{green} but this
should be in green} and this
should be back in red.}
And we finish in black
```

we will see[2]

We start in black, but now all text is in red, but this should be in green and this should be back in red. And we finish in black

The *background* of a normal LR TeX box can also be coloured:

```
\colorbox{colourname}{text}
```

This takes the same argument forms as `\textcolor`, but the colour specifies the background colour of the box. There is also an extended form:

```
\fcolorbox{colourname}{colourname}
```

This has an extra *colourname* argument, and puts a frame of the first colour around a box with a background specified by the second colour.

The normal `\fboxsep` and `\fboxrule` commands vary the line width, and offset of the frame from the text, as the examples in Color Example 5 show.

Defining new colours. The colour names 'white', 'black', 'red', 'green', 'blue', 'cyan', 'magenta' and 'yellow' are predefined by all driver files. New colour names can be defined with:

```
\definecolor{name}{model}{spec}
```

where *spec* is usually a list of comma-separated numbers needed by the *model*. Typically, drivers can cope with the models *gray*, *rgb* and *cmyk* (although the system is extensible), allowing, e.g.:

```
\definecolor{lightgrey}{gray}{.25}
\definecolor{cornflowerblue}{rgb}{.4,.6,.93}
\definecolor{GreenYellow}{cmyk}{.15,0,.69,0}
```

[2] The small examples of colour like this will be set using gray scales in this paper.

If you know that the driver has predefined colour models, you can access these directly. Thus `dvips` has a header file of CMYK colours configurable for different devices (as discussed above), and supports the extra *named* model. We can access these colours in the normal way:

```
\definecolor{Strawb}{named}{WildStrawberry}
```

It is also possible to use the `\textcolor` and `\color` macros with an explicit colour model and specifications, to avoid the overhead of defining new colors and using up TeX macro space:

```
\color[model]{specification}
\textcolor[model]{specification}{text}
```

This enables us to gray-out text like Expandafter by typing `\textcolor[gray]{0.5}{Expandafter}`.

Examples of colour in document design. The simple text colouring described in the preceding section is moderately easy to implement and use. Color Example 6 shows how a simple formal specification can be enhanced with coloured keywords. Shading the background of boxes is also a common requirement, a simple example to emphasize some text might be: ████. The grey-scale simulation of colour as printed here is also not ineffective. For more sophisticated use, the `PSTricks` package by Timothy van Zandt offers a more flexible set of commands (see the article by Denis Girou in the *Cahiers GUTenberg* Girou (1994) for a full discussion, and many examples, of `PSTricks`); the colour support in LaTeX2ε is compatible with `PSTricks`, so the same colour names and definitions can be used. The gradient colour fill in the background of Color Example 8 is an example of more complicated use.

A common requirement is to combine coloured text and shaded areas in a tabular format. This is surprisingly difficult to program in LaTeX, and cannot be undertaken lightly; however, another package by Van Zandt, `colortab`, takes care of almost all needs, utilizing the LaTeX2ε colour primitives. The full set of macros and syntax is described in the documentation (it works in plain TeX, with LaTeX's "tabular" tables, and in Carlisle's "longtable" documents). Color Example 7 shows the results, with a real table taken from a travel brochure (the code is given in Goossens and Rahtz 1994b). This example shows how colour is used to highlight similar structural elements of a table to allow reader to navigate more freely and effectively through the information. It also shows a basic principle of colour work, namely not to use more than two or three different colours, since the codification (the meaning associated to each colour) will be lost. In this case we have used red for the heading, the alternation white/yellow to outline rows, and cyan to draw attention to the price column. It also

shows the efficiency of the fundamental 3-colour harmony, the 3-tuple (red, blue, yellow), as discussed earlier. For an interesting discussion of the use of colour in publishing, we recommend White (1990).

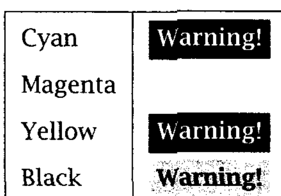
Another very common application area for coloured text or background is colour overhead transparencies. Color Example 8 shows a typical colour scheme for slides, using van Zandt's `seminar` package (see Goossens and Rahtz 1994b in *Cahiers GUTenberg* for more details), including the use of a graded colour fill for the slide frame. As well as coloured background, frame and lettering, we have used another colour for emphasis in the text, and highlighted the bullet lists with yet more colour. Most readers will probably agree that this represents distracting overkill, and that only one emphasis technique should be used at a time.

Simple colour separation using `dvips`

A document containing colour material can be typeset using LaTeX and run through `dvips` to create a colour PostScript document that can be previewed on screen, or printed on a colour printer. But if one wants to produce a "real" book using offset printing the printer will require four versions of each page, containing, respectively, the gray levels corresponding to the proportions of Cyan, Magenta, Yellow and Black. Colour work is usually typeset on special film, to high tolerances, since each page is overprinted four times, and registration must be exact. Some typesetting systems can produce the four separations automatically, but more commonly it is done with PostScript manipulation. A high-level professional quality requires sophisticated tools that are beyond the scope of this paper. Nevertheless, a TeX user can produce straightforward CMYK separations with `dvips`, using an approach that requires only PostScript Level 1 operators.

The principle of `dvips` separations is that each output page is produced four times (using the `-b 4` command-line switch, or `b 4` in a configuration file), and a header file which redefines the colour operators differently for each of the four pages. The header file (distributed with `dvips`, maintained by Sebastian Rahtz, and largely derived from Kunkel 1990 and Reid 1986), uses the `bop-hook` handle to increment a counter at the beginning of each page, and so check whether a C, M, Y or K page is being produced. The `setcmykcolour` operator is then redefined to produce just one of the four colours, in grey, and RGB colours are converted to CMYK before going through the same process. The `setgray` operator is only activated on the black ('K') page. A listing of the PostScript code is given in Goossens and Rahtz (1994b).

The output from separation can be seen (simulated) in Table 1 for the earlier example of ,



Cyan	Warning!
Magenta	
Yellow	Warning!
Black	Warning!

Table 1: Separation output

where the box is set in 'ForestGreen', whose CMYK value is '0.91 0 0.88 0.12'. Notice that the 'M' page will be blank, as neither the green box nor the black text need any magenta.

References

Adobe Systems. *PostScript Language Reference Manual second edition.* Addison-Wesley, Reading, MA, 1991.

Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice.* Addison-Wesley, Reading, MA, USA, second edition, 1990.

Gerritsen, Frans. *Evolution in Color.* Schiffer Publishing Ltd, West Chester, PA, USA, 1988.

Girou, Denis. "Présentation de PSTricks". *Cahiers GUTenberg* **16**, pages 21–70, 1994.

Goossens, Michel and Sebastian Rahtz. "Composition en couleur avec LaTeX". *Cahiers GUTenberg* **16**, pages 5–20, 1994.

Goossens, Michel and Sebastian Rahtz. "Préparer des transparents avec Seminar". *Cahiers GUTenberg* **16**, pages 71–82, 1994.
See also "Colour slides with LaTeX and `seminar`". *Baskerville* **4** (1), pages 12–16, 1994.

Itten, Johannes. *Art of Colour.* Von Nostrand Reinhold, New York, NY, USA, 1974.

Judd, Deane B. and Günter Wyszecki. *Color in Business, Science, and Industry.* John Wiley and Sons, New York, second edition, 1963.

Kunkel, Gerard. *Graphic Design with PostScript.* Scott, Foresman and Company, 1990.

McGilton, Henry and Mary Campione. *PostScript by Example.* Addison-Wesley, Reading, MA, 1992.

Reid, Glenn. *PostScript Language Program Design.* Addison-Wesley, Reading, MA, 1986.

Rokicki, Tomas. *DVIPS: A TeX Driver,* 1994. electronic distribution with software, version 5.55.

Van Zandt, Timothy. *PSTricks: PostScript macros for Generic TeX. User's Guide. Version 0.93,* 1993. electronic distribution with software.

Van Zandt, Timothy. *seminar.sty: A LaTeX style for slides and notes. User's Guide. Version 1.0,* 1993. electronic distribution with software.

White, Jan V. *Color for the Electronic Age.* Watson-Guptil Publications, 1990.

Printing colour pictures

Friedhelm Sowa
Heinrich-Heine-University, Computing Centre, Universitätsstraße 1, D 40225 Düsseldorf, Germany
sowa@convex.rz.uni-duesseldorf.de

Abstract

Printing colour pictures in a TeX document needs a driver program that is able to exploit the capabilities of a colour device. The driver must separate the colours of the picture into the basic colours used by the colour model supported by the output device. This was the purpose to develop the dvidjc-drivers for the Hewlett Packard inkjet printers and to upgrade BM2FONT to version 3.0.

The solution described in this article proposes a device independent approach to printing coloured TeX-documents, not only on expensive PostScript devices but also on cheap colour printers.

How it started

Good reasons. During the last few years the hardware industry has supplied the market with different kinds of colour printers. In particular ink jet printers with increasing quality and decreasing prices appeared on the market. So not surprisingly, more and more TeX users demanded dvi driver programs, that exploit the colour ability of these printers.

The answer to this question usually was the recommendation to use the colour package by Jim Hafner and Tom Rokicki and to print the formatted document via dvips using GhostScript. This was a good recommendation, but the supplementary megabytes, necessary for a GhostScript installation, could be a problem. Another problem is the quality of the output produced by GhostScript, which is way below what users expect of TeX output. Moreover the procedure is rather complicated and slow.

Yet it was not only the availability of good and cheap colour printers that brought the colour problem into the foreground. There were also the discussions and decisions on colour support in TeX by the LaTeX3 and $\mathcal{NTS}$ groups that made it clear that an interface will be designed similar to the known implementation of the dvips driver.

For all these reasons it was decided to write a driver program for the HP DeskJet family, hoping that it could be an example for other devices. It is even to be expected that it could be adapted to the final design of a graphics and colour interface, which is to be developed in the future.

An easy solution. The starting point of the project was a driver program for a dot matrix device and a program which could separate colours of a picture. The driver program was dvidot, written by Wolfgang R. Müller, which had to be extended in a way, that it could use the different inks of the HP DeskJet printers, support the \special-commands for coloured text and, the most important and difficult point, produce mixed colours from the primary colours cyan, magenta, yellow and black.

Colour separation could be done by BM2FONT, so that no special interface for the driver program had to be written. The four colour separations of the picture had to be stored in fonts, differentiated by their names, and then included into a document by overprinting each other.

This plan seemed to maintain the compatibility to dvips and the output it produces on paper, assuming that coloured text and pictures are handled similarly by PostScript. At the end of this paper it will be explained why this assumption was wrong.

Colour models

Before describing the dvidjc driver and the new features of BM2FONT version 3.0, some remarks have to be made on the different colour models we have to deal with in the real electronic world. The most important and mostly used models are based either on the primary colours red, green and blue, or on cyan, magenta, yellow and black.

The RGB-model. The colours red, green and blue are used for phosphorescent surfaces to produce mixed colours, a technique used in colour monitors. The following diagram shows what colours result from overlapping areas of fully saturated primary colours.

Digitized pictures mostly use the RGB model by storing the colour of the pixels in three bytes, each representing the intensities of the primary colours

in the range between 0 and 255, where 255 means a full saturation of the colour. Some economical picture formats use up to 256 colours by using a pixel index to a palette of such colour triplets like PCX or GIF. Others like TIFF or TIGA use 24 bits for each pixel and can reproduce up to 16,7 million possible colours.

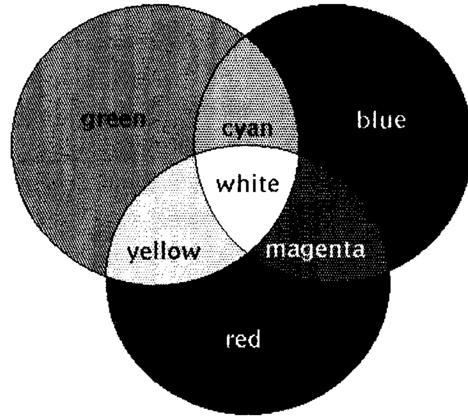


Figure 1: additive colour mix

The CMY-model. The primary colours cyan, magenta and yellow are used for reflecting surfaces like paper. Depending on the technique of a printer it is more or less difficult to position spots of primary colours on paper within a limited area to achieve a good quality picture.

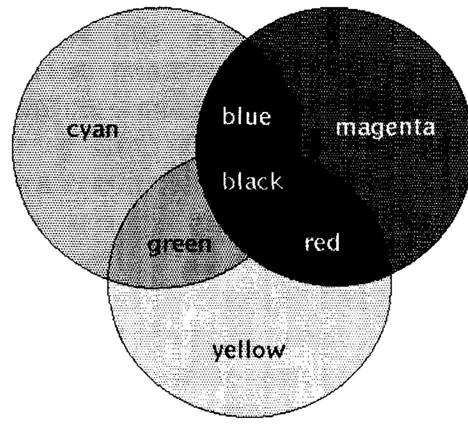


Figure 2: subtractive colour mix

The first problem is the saturation of the colour ink, because full saturation produces colours that are too intense. So usually a colour saturation of about 60% is used for printing each primary colour. The second problem is that areas of overlaying primary colours do not produce a solid black but a colour that looks like a mixture of dark brown and green. So in practice black ink, which extends the CMY-model to a CMYK-model, is used in addition for printing colour pictures.

The effect of mixing the primary colours of the CMY-model is demonstrated by a diagram similar to the one above, for the case of full saturation of cyan, magenta and yellow:

When producing the four different separations of a colour picture, screens of different angles should be used to avoid moiré effects. The common angles are 0° (yellow), 15° (magenta), 45° (black) and 55° (cyan).

Colour separation with BM2FONT

The way it works. To generate colour separations of a picture with BM2FONT, it was necessary to extend the program with the following features:

- converting RGB-colours to the CMYK colour model;
- correcting ink impurities;
- extracting the common black component of the colours.

The intensities of the complementary colours are calculated by subtracting the saturation of red, green and blue from 255, which means full saturation of this colour:

cyan = 255 − red
magenta = 255 − green
yellow = 255 − blue

This is done in BM2FONT3.0 for all bitmap formats, which come with a colour palette, and for TIFF files with RGB triplets, representing up to 2^{24} possible colours.

Correcting the impurity of the ink used is a very device dependent task. The process is known as *Undercolour Removal.* It removes a part of yellow intensity under magenta and partially magenta under cyan before finding out black:

cyan = cyan − $r_m$magenta
magenta = magenta − $r_y$yellow

BM2FONT uses 0.3 both for r_y and r_m, but further versions will read those values from the command line, because colour printers different from the HP DeskJet 5xxC might need other values to get good results.

The blackness of the pixel colour is the minimum value of the new CMY triplet (see figure 2). This blackness now has to be subtracted from the calculated primary colour intensities. The last step is not yet implemented in BM2FONT, because its effect depends on the generated screens, whose angles are different from those mentioned before. As the current solution gave good results on the HP

DeskJet printers, we released this version without implementing another routine, which still has to be tested for different devices.

All this is done in different runs, where the actual colour is given by the `-k` option on the command line. Each run produces one picture in the selected colour as one or more TeX fonts, which are to be overlaid in the document. Former versions of BM2FONT deleted the white space around the rectangle containing the picture, whose corners are determined by the first significant black pixel. This feature is not useful for colour pictures, so it should be switched off by using the `-j` option.

Input for TeX. The main input for TeX and the driver programs is created by BM2FONT in the form of the `.tfm` and `.pk` files, while the input files, generated in each run for the primary CMYK colours, construct the picture in the document. Unlike halftone pictures, where we deal with black and white pixels on paper, we now have to tell TeX, to write into the `dvi` file, which colour is to be used when printing the single parts of the colour picture.

This is done by a `\special` command, similar to the one which is defined in the `colordvi`-package, created by Jim Hafner and Tom Rokicki:

```
\special{color push #1}##1
\special{color pop}
```

Here we have in #1 the colour, for example "Apricot", and in ##1 the text that is to be printed in the desired colour. The `dvips` driver uses the file `color.pro` to tell the PostScript device how to produce this colour `\Apricot`:

```
/Apricot{0 0.32 0.52 0 setcmykcolor}
```

For efficiency reasons we did not implement inside the `dvidjc` driver a routine to read a device specific colour description from an ASCII file. Instead, the driver was supplied with a modified macro file `colordjc.tex/sty`, knowing that this was not the final solution. The difference is, that instead of the name of the colour the intensities of the primary colours cyan, magenta, yellow and black are written into the `\special` literal, while the new colour is defined as

```
\newColor Apricot {0 0.32 0.52 0 }
```

The current version 3.0 of BM2FONT provides no special TeX code to typeset colour pictures. This code is contained in the `colordjc` file. There a macro `\loadcmykpic[#1,#2,#3,#4]` is defined, which loads the descriptions of the coloured screens into the document. Then the complete picture is positioned within the text by using the command

`\cmykpic[`pic_c`,`pic_m`,`pic_y`,`pic_b`]`

A framed picture can be typeset by using the `\fcmykpic` command. Both commands expect in pic_n the names of the pictures, which are defined by the `-f` option in each of the four runs of BM2FONT.

The macros are written under the assumption, that a transparent imaging model is used, rather than an opaque imaging model. The opaque model replaces within the desired area on the page any colour, which was printed before, while the transparent model overprints the already coloured area. So it is possible to get mixed colours simply by overlaying the four planes of the picture in order to generate the final composite image. The original TeX does not know anything about imaging models, because it only expects black ink on white paper. But using colour for typesetting text as well as graphics requires the choice of an imaging model with respect to driver programs. Later on this problem will be discussed in connection with future developments.

The complete process for creating and typesetting a colour picture could look like the following, starting with the colour separation

```
bm2font picture.gif -jn -fpicb -kk
bm2font picture.gif -jn -fpicc -kc
bm2font picture.gif -jn -fpicm -km
bm2font picture.gif -jn -fpicy -ky
```

Inside the TeX document one should add the following commands

```
\loadcmykpic[picc,picm,picy,picb]
\fcmykpic[picc,picm,picy,picb]
```

It is important to mention that the colour macros used for typesetting colour pictures are the same that are used for typesetting colour text, when they are expanded by TeX. This means for the usage of a `dvidjc` driver, that we get the mixed colour of green in areas where two overlaying components of text in the primary colours cyan and yellow have coloured pixels at the same position.

The dvidjc-drivers

The old `dvidot` driver program, developed by Wolfgang R. Müller at the Computing Centre of the Heinrich-Heine-University, had to be extended by code to interpret the literals of the `\special` commands, which are written into the `dvi` file when using the macros of the `colordjc` style.

```
if (memcmp(comment,"color", 5)){
  fprintf(prot,"special »%s« ignored",comment);
  return;
  }
parm = comment+6;
if (!memcmp(parm, "pop", 3)){
```

```
  parm +=4;  colst-;
  if (colst < 0) {
    colst = 0;
    fprintf(prot," color stack underflow");
    }
  }
else {
if (!memcmp(parm, "push cmyk", 9)){
  parm +=10;  colst++;
  if (colst > smax-1) {
    colst = smax-1;
    fprintf(prot," color stack overflow");
    }
  }
if (sscanf(parm,"%lg%lg%lg%lg",
                &cy,&ma,&ye,&bl)<4){
  fprintf(prot," color: cmyk values <%s>
                incomplete ",comment);
  return;
  }
```

This code handles the most simple case, where some text has to be printed in a given mixed colour, which is made of CMYK intensities. Those intensities are defined by the `\newColor` macro. So the driver program knows how to handle the operation codes, enclosed by `color push cmyk` and `color pop`.

Colour production. Producing the desired mixed colour by the `dvidjc` driver is relevant only for printing coloured text, not for colour pictures. The screens made by BM2FONT already have the cyan. magenta, yellow or black pixels at those positions, where for example a blue sky or a red nose should be shown. Producing mixed colours for text is a task which has to be done by the program.

This is implemented by defining a 4 times 4 grid for each primary colour, which contains in certain positions a threshold value, that indicates whether a pixel is to be coloured or left white. This `maskmat` table is used to generate the bitmap for the current primary colour. The colour intensity is transformed to a value between 1 and 16, then a mask is generated with bits turned on in positions where the transformed intensity is less than the threshold value. The colour bitmap is then built up by switching on the bits which are black in both the mask and the originally black bitmap of the page. Color Example 11 shows the result of that process.

Different printers. The available `dvidjc` driver, released in January 1994, supports the colour HP DeskJet printers 500C and 550C and all monochrome printers with PCL support. When compiling the source it depends on a compiler definition, specifying the printer, to be supported by the generated program.

The monochrome version converts coloured areas within the bitmap of a page into different levels of grey. Colour versions distinguish between the additional availability of black ink or the primary colours cyan, magenta and yellow. In the latter case, black — or better a very dark colour — is produced by printing overlaying pixels in primary colours.

Of course good colour reproduction depends on the printer, its resolution, the purity of the ink, and on the kind of paper. All those factors influence some parts of the program:

- the generation of printer control sequences;
- the allocation of memory for colour bitmaps;
- the positioning of the threshold values for intensities of the primary colours.

The necessary code is written in the `hardcopy` procedure or preceded by a `#ifdef DJCOLOUR`, `DJ500C` or `DJ550C`. We mention this to invite *everybody* to extend the `dvidjc.c` source to support printers other than the HP DeskJet.

In the future the information about the CMYK intensity of a defined colour like "Apricot" should be read from a device specific file instead of getting that information by a `\special` command when reading the `dvi` operation codes. So the `dvi` file contains a `color Apricot` and the description file specifies how the mixed colour "Apricot" is realized on that device.

Previewing. The `dvidjc` driver package contains a previewer `dvivgac` for MS-DOS, which supports colour output. The driver starts in halftone mode, showing in the left part of the graphics screen the first page selected in the document, while in the right screen information about the usage of the program is given. When pressing a cursor key a rectangle appears on the left, showing the area of the page, which is magnified on the right.

Color Example 12 shows the hardcopy of a screen corresponding to the titlepage of the `dvidjc` documentation.

The next step

One of the next jobs will be to make the colour driver more generic as mentioned before and to distinguish between different imaging models. The authors of the package can not do this on their own. The result of further efforts will be a summary of the work of other people with access to different colour printers, with programming experience and some enthusiasm for TeX.

The most important job for the future can not be done only by hacking code. Writing a colour

driver requires a complete recommendation how to include graphics, both monochrome and coloured, into a TEX document. In spite of the fact that the described method works, a standard is necessary, because there are some problems left.

Problem with PostScript. The PostScript colour model works under the default assumption that a coloured region on the page is printed after the colour in an overlapping part of another region is removed. This *knockout* mode is unknown to the HP DeskJet printers, and was the reason why the first attempt to check the compatibility between the dvidjc drivers and the combination of dvips and GhostScript was discouraging: the pictures looked ugly, cyan text within a yellow box was not green. But reading the PostScript manual helped.

Simulation of knockout mode can be done by TEX, to achieve compatibility. If, for example, a yellow coloured text is to be printed inside a cyan box, then the text simply has to be positioned by using the colour \White before using the yellow colour. This was the cheapest method for the dvidjc drivers to simulate an opaque imaging model.

Gray in Black

This example should also make it clear that overprint mode is not useful when using colours with equal intensities of primary colours.

Anyway it is advisable not to mix the ink of a region containing text and the ink of a background area of the text. Overprinting that region would not produce the desired colour for the text. But what is correct for text is not necessarily correct for pictures. Here we need mixed colours, when the screens for the primary colours are overprinted.

A solution for that problem could be a \special, that tells the driver to switch into overprint mode or back into knockout mode. A probably better way could be to design a graphics interface, that enables driver programs to get information about the kind of picture included. Depending on the characteristics of the included graphic, the driver could switch to the appropriate mode automatically.

Problem with BM2FONT. The main disadvantage of BM2FONT is the usage of TEX fonts to include graphics into a document. This is the reason why the number of pictures printed in a document is limited. When printing colour pictures, up to four times more fonts than for a monochrome picture are generated.

The conversion of graphics into an easy to handle rectangular box consisting of characters, is an advantage compared to existing DVI drivers. But this method makes it impossible for colour supporting drivers to distinguish between text and graphics. A solution by marking that part of a page with enclosing \special commands would be contrary to the aim of BM2FONT to print included pictures with any device driver.

It looks like the expansion to colour support signals the beginning of the end of BM2FONT. We are confident that the long standing demand of a graphics standard and the problems connected with global colour support in TEX will lead to a solution that is similar to the one adopted by the driver family of the emTEX package and by dvips. External files will contain the graphics, and the typographic information like metric and colour will be derived from a description file.

Colour Pictures

Color Example 9 shows the locations of European home pages in the World Wide Web (WWW). The picture comes from //s700.uminho.pt/europa.html.

Color Example 10 is a picture of the campus of the University of Dortmund, Germany, while Color Example 11 shows the same picture separated in its four colour components cyan, magenta, yellow and black.

Color Example 12 is a screen dump of the dvivgac previewer in colour mode. The original picture was scanned from page 304, *The TEXbook*. Colours were added with the Xpaint program.

Bibliography

Clark, Adrian F. *Practical Halftoning with TEX*, *TUGboat* **12** (1), pages 157-165, 1991

Hilgefort, Ulrich, *Farbe aufs Papier*, c't **4**, pages 132-139, 1994.

Color Book Production Using TeX

Michael D. Sofka
Publication Services, Inc., 1802 Duncan Road, Champaign, IL 61821 USA
mike@psarc.com

Abstract

When typesetting a color book the goal is to produce a separate printer plate for each of the colors. The process of splitting the printed output into separate plates is called *color separation*. There are two color separation methods commonly used. *Custom* color separation selects colors from a standard pallet. A different plate is created for each color in the book. *Process* color separation separates the colors into the subtractive color components cyan, magenta and yellow, and creates a separate black plate by a process called undercolor removal. These four plates are used by a printing press to mix the colors on paper when the book is printed. Color separation is a more involved process than simply assigning RGB values to a desktop color printer. This article addresses the issues of professional color separation, and demonstrates how TeX with a suitable dvi driver can be used to produce quality custom and process color books.

Introduction

There has been recent interest in using color with TeX. This is evident by macro packages such as Foil-TeX (Hafner 1992), the discussions about color on NTS-L (New Typesetting System List), and the new standard color support in LaTeX2ε (Goossens, Mittelbach, and Samarin 1994). This interest was most likely initiated by the availability of low cost desktop color printers, and the desire to make use of these printers[1] with TeX.

Foil-TeX, LaTeX color styles, and other macro packages provide an easy way for the owners of desktop color-printers to use color with TeX. Their goal is a simple method, using macros and specials, to select color output on a desktop printer. This is different from what is required in color book production. When typesetting a color book the goal is to produce separate plates for each color used by the printing press. An example will help to clarify this. Imagine that you are typesetting a book that will have red section headings. The final product required by the printer[2] is two sets of negative film or camera-ready copy.[3] The first set will be the "black" film, which contains only the black text. The second set will be the "red" film and will contain only the red section headings. But, both sets of film will be printed on a black and white imagesetter because *it is the responsibility of the printer to provide the correct color ink to the printing press.*

The process of dividing the pages into separate printer plates is called *color separation*. Color separation is a more involved process than assigning RGB values to a desktop color-printer. The question addressed by this article is: "can color separation be done with TeX?", and the answer is: "yes, with an appropriate dvi driver." At Publication Services we have been typesetting color books in TeX since 1987. In 1993 we typeset our first process color book. This was done using a collection of specials that provide information about the current color, its type (process or custom) and its marking model, or how the color interacts with other colors placed on the paper underneath it.

[1] There are unfortunately 3 potential uses of the word "printer" in this article. To avoid confusion, I will use the term *desktop printer* to refer to a low resolution device, color or otherwise, which is used to print files. The term *printer* will refer to a person or corporation that prepares printing press plates and uses those plates to print books. Finally, *imagesetter* will be used to describe a high resolution printer that images to photographic paper or film. An imagesetter is also called a typesetter, but I will use that term for a person who sets type (electronically or otherwise).

[2] The person who makes plates and runs the printing press (see footnote 1).

[3] Film is clear acetate which is used to expose plates for an offset printing press. Film is printed with a negative image, that is transferred once to produce a positive plate. Camera ready copy, or CRC, is photographic film exposed by an imagesetter. Before transferring to a plate, a negative must be made of CRC. Because of the loss of quality when shooting the negative from CRC, negative film is usually requested for color books.

This article will explain the color separation process and describe how TeX can be used to support electronic color separation. It will concentrate on the PostScript color model, and the quality control steps necessary to ensure good color reproduction. Some of the common mistakes made by authors attempting their own color separation will be discussed, so pay attention.

The PostScript Color Model

The PostScript page description language (Adobe Systems 1990) has become a *de facto* standard in the publishing world. For this reason I will be discussing color separation assuming the PostScript imaging model. The general principles, however, apply to any color separation, and we have typeset custom color books using Cora (Linotype 1988).

An important element of the PostScript imaging model is that all marking is opaque. That is, any mark placed on the page will completely cover (remove) any existing marks it overlaps. This applies equally for solid characters and rules, and for tinted regions. Not all imaging devices work this way, and any attempt to color separate non-PostScript output must take into account the page description languages imaging model.[4]

Adobe defined a series of commands for setting color in PostScript. Some of these commands are now built into level 2 PostScript, but others are conventions defining how to interpret other commands. If you are preparing PostScript files for color separation you should familiarize yourself with these commands and conventions. They are listed in Adobe Systems (1989), and updated (and simplified) in Adobe Systems (1990).

Custom versus process color separation. There are two methods of color separation used in book production. These will be referred to here as *custom* color separation and *process* color separation, although the terminology used by other typesetters and by printers may vary. Custom color separation[5] is the process described in the introductory example. Each element of the book design is set in a specified color. Each of these colors is printed on a separate piece of film. The colors themselves are assigned based on standard color references similar to those used to select house paints. One common reference for custom colors is the Pantone system (Pantone 1991, Pantone 1991). Pantone sells standard approved color charts, and inks. A book design, for example, may ask that Pantone 231 (a light red) be used for all section heads. The typesetter's task in this case is to provide two negatives for each page, the printers job is to prepare plates from these negatives and select an approved Pantone 231 ink for the color plate. If late in the typesetting process the the publisher changes the design to use Pantone 292 (light blue) instead of Pantone 231, the printer can supply a Pantone 292 approved ink. The typeset negatives and the prepared plates will not have to be changed. A small sample of Pantone colors can be found on color plate I.33 of Foley, van Dam, Feiner, and Hughes (1990). If you look up plate I.33 you will note that the actual names are obscured since the colors reproduced in that book are not Pantone colors.

With process color separation each color is separated into cyan, magenta, yellow, which are the subtractive color components. Black is supplied by a process called undercolor removal which removes equal amounts of black from cyan, magenta and yellow.[6] This is done to provide a better, well registered black. (Imagine how a book would look if all the text was composed of three layers of ink.) The final result is referred to as CMYK color.

Process separation is a more difficult process because of the need to have correctly calibrated colors. With custom colors the printer is responsible for supplying the correct ink. With process colors, on the other hand, the typesetter's job is more difficult because correct color balance will depend on the quality of negatives supplied to the printer, as well as the quality of ink provided by the printer. The final typeset output of a process color book is four negatives. Each negative represents one of the cyan, magenta, yellow or black components of the book. The ink will be mixed on the paper by the printing press. Process color is also called four-color, and the two terms are synonymous in this paper.

In order to provide the correct color mixture the negatives will have screened regions corresponding to page elements. A screen is an area of shading that provides a percentage of the required color. For example, an orange-red can be printed using 0% cyan, 30% yellow, 70% magenta and 0% black. The four negatives must reflect these percentages. In the case of the cyan and black negatives, the color region will be black because no cyan or black ink is required. The yellow film however, will have a screened area that is approximately 30% filled, and the magenta film will have a screened area approximately 70% filled.

[4] The imagesetter language Cora, for example, assigns priorities to overlapping tinted regions. The final marks are those of the region with the highest priority. Cora also allows the specification of different tint and pattern values for the intersection of regions.

[5] Custom colors are often called "spot" colors in desktop publishing and drawing programs.

[6] This method of undercolor removal can flatten colors and result in too much black on the final print. In practice color balance must be checked and adjusted as needed.

In order to have accurate color output with process colors it is necessary to calibrate your imagesetter to produce the correct screen percentage. Color calibration is notoriously difficult because of the limitations of color monitors, the screen patterns generated by different output devices, and the absorption properties of the paper. Each of these will be addressed below.

Knockout versus overprint. Another important consideration when separating colors is deciding what happens when one color is placed under another color. The two possibilities are that (1) the first color is removed before the second color is placed. This is called color *knockout*. The second possibility is that both colors print, which is referred to as *overprint*. When designing a book, writing macros, or working on a figure it is important to know which action a color should take. For process separation the default assumption is that the four separations (cyan, magenta, yellow and black) set overprint. This makes perfect sense since the goal is to mix the ink on paper. For custom color separation, however, the default is that custom colors knockout any element set under them. Again, this makes sense when you consider what a custom color represents. A custom color is an industry standard color selected by the designer of the book. Mixing a custom color with any other color will change its appearance from that of the standard.

There is an exception to the general rule that custom colors set knockout. A duotone is a custom color used as a process color. The effect is to mix, for example, Pantone 292 with black to create a variety of colors from pure tone Pantone 292 to pure black. Once again, the intended effect is to mix colors on the paper so the colors must set overprint. In this case, however, the custom color is being used as a process color, and in practice such books are prepared as process color books with 0% yellow and magenta components.

Recall that the PostScript model assumes knockout colors by default. Depending on how your files are separated this could work for, or against you. If the separation is done by manipulating the PostScript color-space (see Rahtz and Goossens, these proceedings) then knockout and overprint can be set via PostScript commands. This is the approach Adobe uses in their Separator program. If, on the other hand, some macro or driver manipulation is being used to remove a color during printing (for example by shifting it off the page) then a page element previously knocked out will now print. This approach effectively sets all colors to overprint. Similarly, if separation is done by simply changing the PostScript `setgray` value then all colors are effectively set to knockout.

Ink order. When preparing color separated output it is often important to consider the order in which colors will be applied to paper by the printing press. There are three rules to remember: (1) black is set last on the press, and (2) yellow is very light and tends to get lost in darker colors, and (3) black will effectively cover most other colors.[7]

Applying these rules we see that if, for example, yellow text is being set on a solid black box the yellow must knockout the black in order to be visible in the final book. If black text, however, is being printed on a solid yellow box, the black does not need to knockout the yellow. In fact, it is preferable to have the black overprint the yellow to avoid problems with trap (see below).

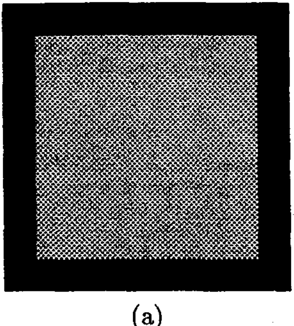
Technical Difficulties.

The task of color separation is conceptually very simple, but in practice can be fraught with difficulties. Most of these difficulties stem from the process of re-integrating the separated plates. In order to print color books the process of applying ink to paper must be understood. Many of the problems that can occur with this process are under the direct control of the typesetter.

Registration. Once the separations are made it is necessary for the printer to realign them correctly on the paper. This process is called registration, and it is a subtle point that is often missed by authors preparing their first color separated book. In order for the printer to be able to re-align separated output the output must include registration marks. These are alignment marks that are printed on every page regardless of color. By aligning the marks a correct composite should be obtained.

Registration also refers to the quality of alignment in the separated output. Obviously, the registration marks should set in the same position on each page, but less obvious is the effect that image setter capability and film quality can have on registration. Some imagesetters are rated for color by specifying the repeatability of the output. The repeatability is usually specified as the difference between negatives (in mills), and the time frame over which the repeatability holds. That is, if a negative is printed on the 1st of the month, how likely is it that the registration will be within 1/10 mill if the negative is reprinted at the end of the month. Registration is also affected by the weather. This is because the acetate used for negatives will stretch or shrink slightly as the humidity changes. Maintaining a constant work environment is therefore important for good registration,

[7] The press order is usually yellow, followed by cyan, then magenta and black. If an additional layer is being applied it may be set earlier or later depending on the design.


(a)

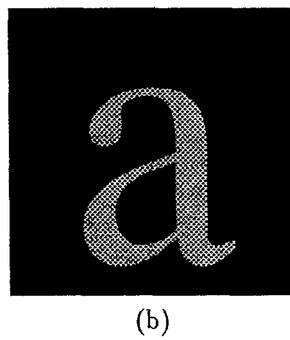

(b)

Figure 1: Examples of trap. (a) Red screen over black. (b) Red character over black requires knockout and character trapping code.

and some imagesetters include humidity and temperature control units to maintain repeatability.

Tagline. To keep track of each color of each page the printed negatives should all contain identifying information. This is called the tagline, and it can go a long way towards avoiding confusion. The tagline is less important for one-color books since most pages will have a folio (page number). Color books, however, will have one or more colors printing without a folio. Some pages, in fact, may be completely blank except for the tagline and registration.[8]

Unless you have actually tried to keep track of several hundred pages of negatives (in a shop that is printing more than one book) it is difficult to appreciate how important taglines are. The usual procedure followed by printers if they receive a page without a tagline is to return it. A tagline should include at least the folio and color. Some identifying name or title will help avoid confusion with other books.

Trap. Registration is only one aspect of realigning separated plates. Since the output of TEX is specified in scaled points, and imagesetters print at 2540 dots per inch or more, a high degree of registration is possible. But, there are cases where exact registration is not wanted. When setting elements in different colors next to each other (elements that will separate to different plates) it is necessary to provide a small region of overlap to prevent the white paper from showing through. This area of overlap is called trap. Trap is usually specified in mills, with 3 mills being a typical value. It is not a large area of overlap, but it is important.

An example requiring trap can be seen in figure 1a. The figure sets a region of Pantone 231 (red) over a region of black. Even the most exact press alignment is likely to allow a small gap of white paper to show through. To prevent this, the Pantone 231 must be trapped against the black. Note that the red square cannot just overprint the black. Instead, what is required is that the black be knocked out, and then the red set overprint overlapping the black by 3 mills. The ability to set knockout *and* overprint make trap possible. You could argue that if all colors set overprint, the black frame in figure 1a can be set with rules. Then a single red rule can be set inside the black frame. This will not work for figure 1b, which requires special support for trapping font characters.

[8] Blank pages are required to build the imposition. The printer has no idea what belongs on each page of an imposition and missing pages can cause confusion, delays and errors.

Screens. Perhaps the single most important technical difficulty with color separation is screens. A screen is a area of the paper that is not 100% filled with ink. The term tints is often used, and you can observe the effect of screening by using the PostScript `setgray` command. When setting a photograph the screen is called a halftone. Screens are used to create a region where color is not fully saturated. This is done by setting a pattern of dots which partially fill the region. The reason for using dots is that desktop printers, imagesetters and presses either place a dot, or they do not. There is no mixture of, for example, cyan and white ink to dilute the color.

While setting a screen in PostScript is simple in principle, in practice it requires special equipment and attention to detail. Desktop printers are incapable of setting correctly screened output. This is a limitation of their marking engines—300 or 600 dots per inch is just not enough dots to provide good screens. In addition, the paper and toner used by a desktop printer have too much dot-gain (see below) to present an accurate screen.

PostScript screens are a printing device dependent feature. That is, the method by which a screen is set is not defined in the PostScript language. This was most likely an intentional design decision since the best method of setting a screen will depend on the qualities of the marking engine and print media. In addition, screen generation is an active area of research, and each imagesetter company has its own brand of screening technology for sell. Screen type and quality is a very important consideration when purchasing an imagesetter—perhaps *the* most important consideration when printing separations for a process color book, because bad screens equal bad process color.

Screen attributes and types. Different imagesetters are sold with different screen types installed. Before discussing the types of screens available we need to understand how a screen works. In PostScript the `setscreen` command is used to change default screen qualities. It takes three arguments: frequency, angle and procedure. The frequency is how many lines per inch are represented in the screen.

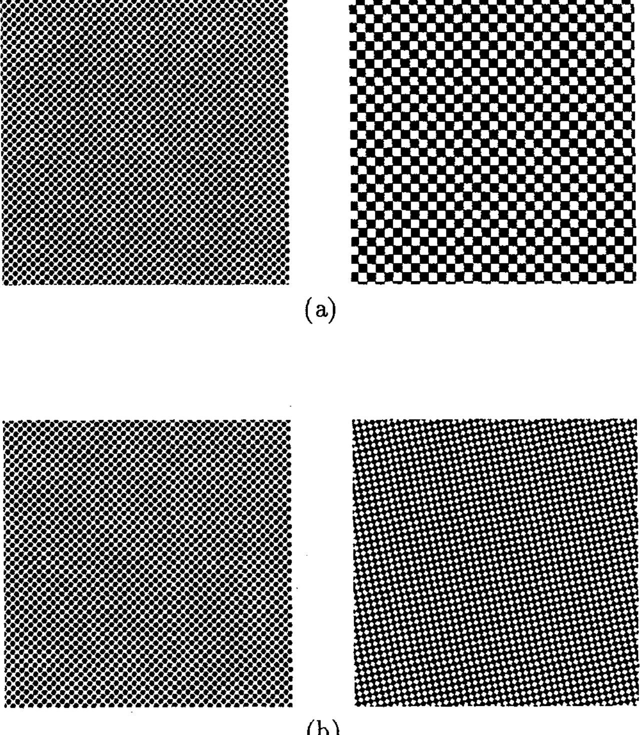


Figure 2: Examples of the effects of frequency and angle on screens. (a) Screens differ only in frequency. (b) Screens differ only in angle.

For example, in figure 2a the two screens differ only in frequency. The angle is the angle of the lines composing the screen. The screens in figure 2b differ only in angle.[9] Finally, the procedure defines a function to determine the order in which halftone cells are filled to produce the desired shade of gray.

Not all combinations of frequency and angle can be set by an imagesetter. The actual values supported vary with screen type (the procedure) and the resolution of the marking engine. The best output is usually obtained for only a subset of the possible values. In addition, the printer may request a particular range of frequency for best pickup while making the press plates since too high a frequency may not transfer, or may result in dot-gain. The best advice to offer an author preparing color separated output is to not use the `setscreen` command unless you know what type of imagesetter will be used to print the film, and only after conferring with the operator of that imagesetter.

[9] Because of PostScript device limitations the screens of figure 2b may differ in angle and frequency. The actual PostScript command requested only that the angle be changed. This is a good example of the device dependency of screens.

Most high resolution PostScript imagesetters are capable of *Rational Tangent* (RT) screens. These screens are limited to angles which have tangents that can be represented as a ratio of two integers, and frequencies that evenly divide the device resolution. On imagesetters sold for color work it is common to find *Irrational Tangent* (IRT) screens. As the name implies, IRT screens can represent angles whose tangent is a real. IRT screens are a minimum for process color books, but they suffer from limitations which result in a poor reproduction. All imagesetters sold for four-color books have some, usually proprietary, screen system included. The latest screens employ stochastic methods to eliminate repetitive patterns which result in moiré. These methods are usually built into the PostScript interpreter instead of defining a PostScript procedure with `setscreen`. This is a matter of efficiency since screen calculation can consume the majority of the CPU cycles in an interpreter.

Screens in process colors. If you are typesetting a process color book, and intend to print your own output then you will have to buy screens. There are many screens available, and most of them come with a imagesetter. This is only a slight exaggeration because screen type and quality are dependent on marking engine ability. Some companies do, however, offer screen updates to existing imagesetters. These work by changing the PostScript procedure, and can be slow as a result.

The reason that screens are so important for four-color books is that process color separation requires that screen areas overprint each other. When this is done, a variety of undesirable side-effects may occur. The most common problem is moiré. In traditional color separation the halftone screens for each of the colors are rotated so that the dots overlap to form circles called rosettes (Agfa 1990, Bruno 1986). The screen angles used are 105°, 75°, 90° and 45° for cyan, magenta, yellow and black respectively (Agfa 1990). The rosettes formed are invisible to the unaided eye, and instead the illusion of color is created. If the dots are not placed accurately, however, moiré will result. Moiré is caused by an interference pattern between the screens destroying the illusion of color. Most PostScript printers (and this includes a large number of imagesetters) are not capable of placing dots accurately enough to avoid moiré. Even with a color ready imagesetter, special screening methods are required since there are just not enough pixels to create an accurate circle at the size required. Some of the methods used are:

- Oval, or other non-circular patterns which overlap to form larger rosettes.
- Randomized noise added to the patterns to disturb the regular interference which causes moiré.

- Micro-dots (very small dot patterns) that are too small to overlap into rosettes. Micro dots can be set in a random pattern to create stochastic screens.
- Very high resolution output. It is not unusual, for example, to find advertisements for 5300 dot per inch imagesetters. These imagesetters are more capable of holding a "hard" dot (an industry term for identically shaped round dots) in screen patterns.

Each of the above has its advantages and disadvantages. The most common disadvantage is that variations on screening methods can be very slow—especially when random noise is being added. Stochastic screens usually suffer from larger scale patterns caused by poor randomization algorithms or small cell size. The result is, once again, moiré.

Other technical difficulties. Moiré is not the only problem encountered with screens. The following list of difficulties must all be overcome in some way by the imagesetter and printer. The compositors responsibility is to provide the best possible output that reduces problems for the printer.

- Dot gain.

  Dot gain is an increase in the size of a halftone dot from the time the negative is printed by the imagesetter until the final paper is printed by the press (Gretag 1993). Dot gain will affect the amount of ink transferred, from the plate to the final paper. The factors affecting dot gain are paper absorption, screen frequency (higher frequency = higher dot gain), and ink thickness. You can think of dot gain as the amount of smear that takes place on the final printed book.
- Processing Speed.

  It has already been mentioned that some screening methods take a lot of time to process. It is not unusual for a simple switch from IRT screens to a randomized oval screen to increase processing time by 800% or more. When the IRT screened page took 15 to 20 minutes to print the slowdown for oval screens can be significant.
- Screen Models and Patents.

  Many screening methods have been developed by printing and imagesetter companies and are covered by patents. These are not necessarily software patents since the original patent was granted when a "screen" meant a physical piece of acetate which is laid over a color photo when shooting negatives. The extension of the patent to software embedded in a RIP is more natural than in many other software patents. A company using patented methods without paying a license fee, or purchasing equipment and programs from the imagesetter manufacturer would be infringing.

Screens and custom colors. While screen quality is important with custom color books it is not as critical a component as with process colors. Usually RT screens provide output of sufficient quality. This is because process colors do not mix (unless doing a duotone), and both black, and custom colors are printed with 45° screens (whose tangent is 1/1). As a result, screens do not overlap and interfere with each other. It is important, however, to be able to provide screens with a dot size (frequency) that can be photographically reproduced when making press plates. It is also important that the screen density match the specifications, and that the density be even within and between the screened areas. This is usually beyond the capabilities of desktop printers, and publishers will typically avoid screens when authors are preparing camera ready copy.

Color calibration. When preparing and testing process color separated negatives it is important to maintain correct color balance. A variety of conditions can affect the final appearance of your output. The most basic is the calibration of the imagesetter. When a 20% screen is requested the output should be a screen that provides 20% fill of the area after printing. If you are proofing the color then all proofing devices from monitors to printers and photographic based proofing systems must be calibrated to provide the best color fidelity they are capable of. Final decisions about color should not be made on the basis of desktop color printer output. Finally, the conditions under which colors are checked must be constant to avoid metameric color matches (Bruno 1986), or other light dependent color changes. This may require building a color proofing room with controlled lighting.

CMYK color space is based on the absorption properties of particular inks, but ink batches can vary from printer to printer. For this reason CIE color-space is often suggested for internal calibration. Then standard printer samples (available from film suppliers) can be prepared showing a match between the local color calibration and final product. If the final color is incorrect, it can then be shown to be a problem with the plates or printing press, and not a problem with the negatives.

Color proofs. It is important to check color in house before printing negatives, and to test those negatives before printing the book. When labor is considered, a single page of negative film can cost $10 (US) or more, and a page of imposed film will cost even more. If the final printing begins without a less expensive color check (or with no color check), schedules and budgets may slip, and unlucky compositors could find themselves financially responsible for a

bad press run. Color proofs can also serve a contractual purpose.[10] For example, a page may be approved for final film based on the output of a desktop color printer, or some other color proofing method. If this is the case the client and typesetter must agree on the method to be used, and they must be aware of the limitations of the chosen method.

The only real test of color fidelity is the final printed book. This is because the actual colors will depend on the ink batch, paper, and press calibration. It can be expensive, however, to setup a press run for color testing. A variety of methods are available, therefore, to check color before the press. In general, there are two broad classes of color proofing systems—those that test the PostScript files, and those that test the negatives. Each has its place in that it is less expensive to catch errors before printing film, and before using that film to print a book. PostScript files can be checked using a color monitor or color desktop printer. Depending on the type of color being used, negatives can be tested using blues (ultraviolet sensitive paper exposed through negatives), cromalin (dry power colors), photographic paper exposures, or other systems. Each of these color proofing methods has its own tradeoff in cost, time to prepare proofs, and the quality of the proofs.

There are two reasons for color proofing. The first is for color breaks. That is, testing that each element is in the correct color. For custom colors this is usually all that is required, and desktop color printer output is often accepted as proof that the file's colors are correct. Likewise, blues or visual inspection can be used to check that custom colors were separated correctly to the final negatives. Grayscale desktop printer output can also be used if each color is printed at a different tint percentage, and separated grayscale output can be used to check the separations. It usually requires some negotiation, however, for clients to accept grayscale output as a color proof.

Process colors must be tested for both color breaks, color fidelity, and moiré. This usually requires that the negatives be tested using cromalin or photographic processes. There are, however, desktop color proofing systems that send separated files to a color desktop printer. These systems allow for moiré to be checked before printing negatives, and provide near photographic color output for checking fidelity. All of these systems, however well calibrated, do not provide perfect proofs. Once again, the only true test of final color is a press run using the same quality materials that will be used in the book.

[10] Thank you to an anonymous reviewer for pointing out this fine point of client typographer interaction.

Using TEX

After considering the above the obvious question is: "can TEX be used for professional color book production?" The answer is yes, but it requires dvi driver support, and the typographer should be aware of the procedures involved in plate making and book printing. First, be aware that most imagesetter manufacturers, and consumable suppliers will not know about TEX so you can expect little technical support when printing. On the other hand, most of the actual problems with color calibration, moiré and dot gain are common to all color separation electronic or manual, with or without TEX. If you buy an imagesetter you are also buying expertise in its use. Use your purchases to leverage help. Second, many of the issues of color fidelity and moiré apply more often to photographs than the typical TEX element. Solid color, however, is not immune to moiré and the effect looks very bad in printed books. In addition, the colors used in TEX may need to match those used in figures, and small differences in the screens and density can produce a noticeable difference in the final product. Finally, color books, especially process color books, tend to have more complex designs where text and figure elements interact with each other. A figure, for example, may be surrounded by a color box or head-element that must trap with the figure. So all of these checks and balances must work with TEX as well as with figures and photographs.

TEX provides a powerful macro language that can make the process of managing color elements very easy compared to the more common desktop systems. Adding color late in a book design, for example, can usually be accomplished with a macro change. Changes to the selected color can likewise be affected by changing macros. Low resolution "placement only" figures can be included by using a TEX conditional, or via driver switches. In short, we have found that using color with TEX is not only possible, but that TEX helps the process by virtue of being programmable.

Specials for color. Color separation in TEX requires dvi driver support. The specials for color separation must convey the following information:

- The CMYK color values;
- process versus custom color separation;
- knockout versus overprint marking;
- foreground percentage.

The first requirement is obvious. The values of cyan, magenta, yellow and black are basic to determining the final color for process separation, and for proofing both process and custom colors. Process versus custom determines the type of separation to be used for a color, and knockout versus overprint is for specifying what happens to colors set below a new item. Note that black is a color, and the specials

must keep track of when black is knockout, and when black is overprint.

Specification and interpretation of the CMYK values, however, change with the color model. When a process color is being set the CMYK values represent the actual percent values of cyan, magenta, yellow and black ink required on the paper. A change to these values will represent a change to the final color. For a custom color, however, the CMYK values represent an approximation to the final color to be used by proofing devices, and it in no way affect the final book color. Again, this is because it is the printer's job to supply the correct custom ink for the press.

The above is an important point that is central to doing color separation correctly. When specifying a color the goal is to provide the best possible quality in the final printed book. But, the values of CMYK which produce the best book color are unlikely to produce the best desktop printer or monitor colors. When printing custom colors this is of no consequence, but when printing process colors it has a number of consequences that can affect the final product. Process color separation shifts the responsibility for good color from the printer to the typesetter. A good color special, therefore, may provide different values of CMYK for final separation versus composite proofing. In fact, different values of CMYK may be required for each output device that might be used to proof the pages.

One approach to selecting the correct custom color CMYK values is to use a standard set of color names based on the Pantone color charts. The special would then specify a color name which keys the correct CMYK values. A different CMYK value would be required for each printing device. For process colors the task is slightly more difficult since there generally are no standard names. A local file which provided generic names such as "light_green", with the correct CMYK values for each device can be used instead. Alternatively, the special can store a set of CMYK values, with one value designated the "correct" value.

Some of you may think: why not use TEX to define a conditional which selects the color value. We have done this, but the problem we encounter is that the resulting dvi file is device dependent, and would have to be re-TEXed with the correct flags in order to produce correct printed output. This has resulted in a lot of lost time and material, and we avoid such device dependencies whenever possible. When they cannot be avoided, a "printertype" special is used to tell the driver which printer the dvi file is prepared for. A mismatch in printertype aborts printing. This is slightly inconvenient, but much less inconvenient than printing 100 pages of incorrect output (at 1–20 dollars per page depending on output device).

The default value of knockout and overprint differ for process and custom colors. This is because

When typesetting a color book the goal is to produce a separate printer plate for each of the colors. The process of splitting the printed output into separate plates is called color separation. There are two color separation methods commonly used. Custom color separation selects colors from standard pallets . . .

Figure 3: Back text set on a red 30% screen. Since black prints last, and is dark, it can overprint the screen without trap.

custom colors are defined based on standard color charts and inks so it is undesirable to mix them on paper with any other color. Process separations, however, are supposed to mix on the press. This does not mean, however, that custom colors should always knockout and process colors should always overprint. When trapping with a custom color, for example, it is necessary to overprint. Likewise, it may be more convenient to knockout a section with a process color (which must then be separated into overprinting CMYK values), than to typeset around the knockout (see figure 1a). A way is needed, therefore, to change the default behavior of a color.

Finally, the foreground percentage determines if a color is set at 100% of its stated value, or at some lesser percentage. A custom color may be set at 100% value which means that the negative will be clear where the Custom color should set. Just as often, however, the custom color may be set at some percentage of its value. For example, in figure 3 the black text is set over a 30% screen. The screen is a custom color (and the black will overprint so that trap is not necessary). When setting a custom color at a reduced percentage a new color cannot be used since it would then separate onto a different negative. That is, if we are setting Pantone 231 at 30% we want both the solid 100% Pantone 231 and the 30% Pantone 231 to separate to the same negative. We need a way, therefore, to change the foreground value at which any rule, screen or character will set.

Process colors can also be set at a reduced percentage. This does not affect the separations since all process colors will be split into CMYK components. The changed foreground, however, will change the resulting CMYK values, and the method used should provide good output. Often a direct percent reduction will suffice, but special settings may be required depending on how the proofs look. A red, for example, may appear too pink at a reduced percentage and a new color balance may be chosen.

The specials. In the dvi driver `dvips82` used at Publication Services we make use of the following specials to define and use colors. All of the specials follow a simple syntax of `<name> <arguments>`, where the `<arguments>` are `<key>=<value>` pairs. Standard commands which accept any TEX unit of measurement are used to read dimensions from specials. These commands convert the values into scaled points for internal processing.

DefineColor. The `DefineColor` special is used to provide an internal symbolic name for a color and it establishes the CMYK values, model and overprint value. It can also define a tint value to be used when printing on a gray-scale desktop printer. The format of the command is:

```
DefineColor
     <name>=[color(<c>,<m>,<y>,<k>)
            |pms(<pantone-name>)]
            |alias(<color-name>)
     [separation=[process|custom]]
     [overprint=[true|false]]
     [tintpercent=<%>]
```

`<name>` is the symbolic name which is used to refer to the color from this point on. The name is set to one of three definition types. The first provides the cyan, magenta, yellow and black values for the color, while the second provides a standard Pantone name which is looked up by printer type. The last definition defines the color as an alias of a previously defined color. All aliases of a color will separate together with that color, but they can have different separation, overprint and tintpercent values.

The value of `separation` can be either `process` or `custom`, with the latter being the default. The `overprint` argument defines the color as either overprint (`overprint=true`), or knockout (`false`). If the separation is process, then overprint defaults to `true`, otherwise it defaults to `false`.

The value of `tintpercent` is used for checking color breaks on a gray-scale desktop printer. It is not possible to see colors on a gray-scale printer, but worse the colors may be metameric in gray. That is, even though they are distinct colors, they appear the same in black & white. In order to aid proofing a `tintpercent` value can be specified. When printing a composite to a gray-scale printer the tintpercent will be used for all page elements in that color. With the `alias` color definitions different values of `tintpercent` can be used with, for example, knockout versus overprint versions of the same color.

SetColor. Once a color is defined it can be used to change the state of the current color. This is done with the `SetColor` special, which is defined as:

```
SetColor color=<name>
```

where `<name>` is the symbolic name of a previously defined color. All rules, characters, screens and figures from this point on will be set using the attributes defined for `<name>`.

KnockOut, OverPrint and DefKnockOut. The use of aliased colors allows the definition of knockout and overprint versions of the same color. The knockout and overprint values, however, can also be changed using specials designed for that purpose. The special `KnockOut` sets the global color state to knockout. All colors, regardless of their definition, will now set knockout. Similarly `OverPrint` sets the global color state to overprint. The special `DefKnockOut` restores the color state to that specified in `DefineColor`. There are also versions of these specials for setting knockout or overprint for the next single rule, character, screen or box encountered.

SetForground. The current foreground percent can be changed with the special:

```
SetForground fg=<n>
```

where `<n>` is the desired tint value.[11] The default value is 100%, and any value between 0% (printing white) and 100% (printing full color) is allowed. Changing the foreground percent does not change the current color. Instead, all rules and characters are set as a `<n>`% screen.

Trapped and Abutted. Trapping control is supplied by the macros:

```
Trapped trap=<m.n>
Abutted
```

where `<m.n>` is some dimension. When the value of `trap` is non-zero, all rules, characters and screens are set trapped by the trap amount. This is accomplished by first setting the element in knockout, and then setting it a second time overprint. The overprint is stroked by twice the trap amount. The effect is a region of overlap between the trapped element and anything it prints over. `Abutted` is the same as "`Trapped trap=0pt`", and there are also `TrapNext` and `TrapBox` specials to trap only the next element or box.

ScreenR, TintRule, TintChar. In addition to commands to define and change color, overprint and trapping a color book can benefit from commands to set screened elements. This can be accomplished via PostScript, but it is more efficient and easier to have built-in commands. The `dvips82` driver has a variety of commands to set screened areas (with and without rounded corners), circles and characters. Each of these commands allows a trapping specification as well as a background percentage. `dvips82` places all screens defined with these specials under all characters and rules. The specials are based on the tints and patterns commands available under Cora on an L300 imagesetter.

[11] If I were to design the system today I would spell foreground correctly.

Macro considerations. The macros that use these specials are for the most part simple and straightforward. But, there are special considerations around page boundaries. Earlier versions of dvips82 did not keep track of the color state in effect at the beginning of each page. If a macro restores color after, for example, a box then the restoration of color could appear on the next page. This caused problems when printing selected pages, or when printing to a face up device. The macro fix was to have the output routine save and restore all order dependent states with each page break. Later versions of dvips82 perform a preprocessing pass on the dvi file. This allowed optimization of font memory, and the recording of color, foreground, overprint, and so on with each page. Pages selected or rearranged (for example, for imposition printing) still cause problems, however, so that the macro approach has been maintained.

This leaves only the problem of color changes within paragraphs. If the color changes in a paragraph, and changes back before the end, and if a page break occurs in the color section, then the macro based color state will be incorrect at the top of the page. The only TEX mechanism for handling this would be marks. This type of design, however, is so infrequent that we have rarely had problems of this nature.

The separation process. There are two separation methods supported by dvips82. The first method creates an *separator compatible* PostScript file. This is a file that follows the Adobe conventions for color PostScript files, and which can be color separated in a manner similar to that used by Adobe Separator. Adobe Separator color separates a file by including PostScript commands that redefine setcmykcolor. A special PostScript command is used for custom colors so that they can be distinguished from process colors. It also allows a custom color to be converted into a process color, and separated into its CMYK components. At Publication Services we have written a UNIX version of separator that works with Adobe Illustrator and dvips82 files.

The advantage of this method is that color art can be integrated into the PostScript file before separation, allowing art and text to be separated together. For this to work correctly the symbolic *custom* color name defined with DefineColor must match the Custom color names defined in figures. If the names do not match then, as far as separator is concerned, they represent different colors. This can be a problem when using art prepared out of house by the author or another supplier. Hence, UNIX separator (and Adobe Separator) allow different colors to be combined onto the same negative.

The second method uses the driver to do all color separation. This was a very easy addition to dvips82 since all pages were handled in two passes. The first pass stores all rules, characters and other page elements in a table, which the second pass prints. This was originally done in order to place all screens below text (mimicking Cora), but it also made electronic color separation easy. If an item is in the current color it is placed in the page table. Otherwise, it is left out. Each element stored in the table has an associate set of attributes including its current color. Process color elements are printed tinted according the the value of cyan, magenta, yellow or black (depending on the requested separation). Knockout is handled by setting knockout colors with a foreground of 0

dvips82 also contains a color proofing mode which places all elements on the page table, but tints those not in the selected color. This is very helpful for checking color breaks (confirming that each element is in the correct color). The tint value is the tintpercent defined with the color, or 75% if no tintpercent was specified.

The disadvantage of the driver based method is that only color separated figures can be integrated. As a result the figures must be pre-separated. Future modifications include having dvips82 run the separation program on figures. For this to work all figures must use consistent color names, which is, once again, a problem for art prepared off-site.

Summary and Conclusions.

Professional custom and process color separation can be done with TEX and the right set of specials. The specials listed above are what we use with dvips82. They encompass some qualities that I have not seen in other color specials such as color aliasing, and support for gray-scale proofing. The use of knockout and overprint colors is needed in order to be able to trap correctly.

There are a variety of desktop systems that support color, so one may ask why we do not use them. The answer is that we do when they are the right tool for the job. Very often TEX is the right tool for the job, and TEX can easily be extended through specials to equal and exceed the color separation abilities of any desktop system.

Finally, I am very encouraged by the work being done with dvips and color. By adding support for professional color separation to dvips the task of converting an author's LATEX files into professional quality negatives will be made much easier.

References

Adobe Systems Inc. *Proposal for color separation conventions for PostScript language programs.* Technical Report 5044, December 1989.

Adobe Systems Inc. *PostScript Language Reference Manual.* Addison-Wesley, Reading, MA, second edition, 1990.

Agfa Compugraphic Division. *An Introduction to Digital Color Prepress.* Agfa Corporation, 200 Ballardvale Street, Wilmington, MA 01887, 1990. Descriptions of color models, trapping, halftone and screens.

Bruno, Michael H. *Principles of Color Proofing: A manual on the measurement and control of tone and color reproduction.* Gama Communications, P.O. Box 170, Salem, NH 03079, 1986.

Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, New York, 1990.

Goossens, Michel, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion.* Addison-Wesley, New York, 1994.

Gretag Color Control. *Applied Densitometry,* second edition, 1993.

Hafner, James L. FoilTeX, a LaTeX-like system for typesetting foils. *TUGboat,* 13(3):347–356, October 1992.

Linotype. *Linotronic 300/500 Imagesetter Interface Manual,* 1988.

Pantone, Inc. *PANTONE™: Color Specifier 1000/ Coated.* Pantone, Inc., 55 Knickerbocker Road, Moonachie, NJ 07074-9988, 1991.

Pantone, Inc. *PANTONE™: Color Specifier 1000/ Uncoated.* Pantone, Inc., 55 Knickerbocker Road, Moonachie, NJ 07074-9988, 1991.

Inside PSTricks

Timothy Van Zandt
Department of Economics, Princeton University, Princeton, New Jersey USA
tvz@Princeton.EDU

Denis Girou
Institut du Développement et des Ressources en Informatique Scientifique
Centre National de la Recherche Scientifique, Orsay, France
Denis.Girou@idris.fr

Abstract

The macro-commands of the PSTricks package offer impressive additional capabilities to (LA)TEX users, by giving them direct access to much of the power of PostScript, including full support for color. The purpose of this article is to outline the *implementation* of a few of the features of PSTricks (version 0.94).

Introduction

When a PostScript output device and a dvi-to-ps driver are used to print or display TEX files, TEX and PostScript work together, as a preprocessor and a postprocessor, respectively. The role of PostScript may simply be to render TEX's dvi typesetting instructions. However, the full power of PostScript can be accessed through `\special`'s and through features, such as font handling, built into the dvi-to-ps driver.

One can divide the PostScript enhancements to TEX into roughly four categories:

1. The use of PostScript fonts.
2. The inclusion of PostScript graphics files.
3. The coloring of text and rules.
4. Everything else.

Most TEX-PS users are familiar with the first three categories. The PSTricks macro package, by Timothy Van Zandt, attempts to cover the fourth category.[1]

The PSTricks package started as an implementation of some special features in the Seminar document style/class, which is for making slides with LATEX 2_{ε}. However, it has grown into much more. Below are some of its current features:

1. Graphics objects (analogous to LATEX picture commands such as `\line` and `\frame`), including lines, polygons, circles, ellipses, curves, springs and zigzags.
2. Other drawing tools, such as a picture environment, various commands for positioning text, and macros for grids and axes.
3. Commands for rotating, scaling and tilting text, and 3-D projections.
4. Text framing and clipping commands.
5. Nodes and node connection and label commands, which are useful for trees, graphs, and commutative diagrams, among other applications.
6. Overlays, for making slides.
7. Commands for typesetting text along a path.
8. Commands for stroking and filling character outlines.
9. Plotting macros.

For information on PSTricks from the user's point of view, consult the PSTricks *User's Guide* (Van Zandt 1994) and the article by Denis Girou (Girou 1994) in *Cahiers GUTenberg*, the review of the French TEX users' group. The latter article is useful even to those who do not read French, because it consists predominantly of examples. Several of these examples appear in this paper, courtesy of *Cahiers GUTenberg*.

Who can use PSTricks?

A goal of PSTricks is to be compatible with any TEX format and any dvi-to-ps driver. Compatibility with the various TEX formats is not difficult to achieve, because PSTricks does not deal with page layout, floats or sectioning commands.

However, compatibility with all dvi-to-ps drivers is an unattainable goal because some drivers do not provide the basic `\special` facilities required by PSTricks. The requirements are discussed in subsequent sections. All of PSTricks' features work with the most popular driver, Rokicki's `dvips`, and most features work with most other drivers.

Two dvi-to-ps drivers that support the same `\special` facility may have different methods for invoking the facility. Therefore, PSTricks reads a configuration file that tells PSTricks how to use the driver's `\special`'s.

[1] PSTricks is available by anonymous ftp from `Princeton.EDU:/pub/tvz` and the CTAN archives.

Header files

A PostScript header (prologue) file is analogous to a TeX macro file. It comes towards the beginning of the PostScript output, and contains definitions of PostScript procedures that can be subsequently used in the document.

It is always possible to add a header file to a PostScript file with a text editor, but this is very tedious. Most drivers support a `\special` or a command-line option for giving the name of a header file to be included in the PostScript output. For example, the `\special`

```
\special{header=pstricks.pro}
```

tells `dvips` to include `pstricks.pro`.

However, a few drivers, such as Textures (up through v1.6.2, but this may change) do not have this feature. Therefore, PSTricks can also be used without header files. From a single source file, one can generate a header file, an input file for use with headers, and an input file for use without headers.

For example, the main PSTricks source file, `pstricks.doc`, contains the line:

```
\pst@def{Atan}<%
  /atan load stopped{pop pop 0}if>
```

When generating the header file `pstricks.pro`, the line

```
/Atan {/atan load stopped{pop pop 0}if}def
```

is written to `pstricks.pro`. When generating the input file `pstricks.tex` for use with `pstricks.pro`, the line

```
\def\tx@Atan{Atan }
```

is written to the input file. The input file for use without `pstricks.pro` contains instead the line

```
\def\tx@Atan{%
  /atan load stopped{pop pop 0}if }
```

Other macros can use `\tx@Atan` in the PostScript code, without having to know whether it expands to a name of a procedure (defined in a header file) or to the code for the procedure (when there is no header file).

One can also use the source file directly, in which case no header is used. This is convenient when developing the macros, because TeX and PostScript macros can be written together, in the same file, and it is not necessary to make stripped input and header files each time one is testing new code.

The use of header files in PostScript documents reduces the size of the documents and makes the code more readable. However, the real benefit of using header files with PSTricks is that it substantially improves TeX's performance. It reduces memory requirements because, for example, the definition of `\tx@Atan` takes up less memory and, more importantly, `\tx@Atan` takes up less string space each time it is used in a `\special`. It reduces run time because the writing of `\special` strings to dvi output is very slow. A file that makes intensive use of PSTricks can run 3 to 4 times slower without header files!

Parameters and Lengths

To give the user flexible control over the macros, without having cumbersome optional arguments whose syntax is difficult to remember, PSTricks uses a key=value system for setting parameters.[2] For example,

```
\pscoil[coilarm=0.5,linewidth=1mm,
  coilwidth=0.5]{|->}(5,-1)
```

The `coilarm` parameter in this example is the length of the segments at the ends of the coil. Note that `coilarm` was set to `0.5`, without units. Whenever a length is given as a parameter value or argument of a PSTricks macro, the unit is optional. If omitted, the value of `\psunit` is used. In the previous example, the value of `\psunit` was 1cm. Therefore, `coilarm=0.5cm` would have given the same result. Omitting the unit saves key strokes and makes graphics scalable by resetting the value of `\psunit`. This is why the arguments to LaTeX's `picture` environment macros do not have units. However, unlike LaTeX's `picture` macros, with PSTricks the unit can be given explicitly when convenient, such as `linewidth=1mm` in the previous example.

The implementation of this feature is simple. `\pssetlength` is analogous to `\LaTeX`'s `\setlength` command, but the unit is optional:

```
\def\pssetlength#1#2{%
  \let\@psunit\psunit
  \afterassignment\pstunit@off
  #1=#2\@psunit}
\def\pstunit@off{%
  \let\@psunit\ignorespaces\ignorespaces}
```

One advantage of the key=value system is that PSTricks has control over the internal storage of values. For example, PSTricks stores most dimensions as strings in ordinary command sequences, rather than in dimension registers. It uses only 13 of the scarce dimension registers, whereas, for example, PiCTeX uses over 120. When PSTricks processes the parameter setting `coilarm=0.5`, it executes:

```
\pssetlength\pst@dimg{0.5}
\edef\psk@coilarm{\pst@number\pst@dimg}
```

`\pst@dimg` is a register. `\pst@number\pst@dimg` expands to the value of `\pst@dimg`, in `pt` units, but

[2] PSTricks has recently adopted David Carlisle's improved implementation of the parsing, contained in the `keyval` package.

without the pt. Hence, \psk@coilarm is ready to be inserted as PostScript code.

Color

To declare a new color, the user can type:

```
\newrgbcolor{royalblue}{0.25 0.41 0.88}
```

The color can then be used to color text and can be used to color PSTricks graphics. For example:

```
\psframebox[linewidth=2pt,framearc=.2,
  linecolor=royalblue,framesep=7pt]{%
  \LARGE\bf It's {\royalblue here} now!!}
```

It's here now!!

The \newrgbcolor command defines \royalblue to switch the text color, and it saves the color specification under the identifier royalblue so that the PostScript code for setting the color can be retrieved by color graphics parameters.

This support for color has been part of PSTricks since its inception. However, a problem that has arisen is that there are now many packages available for coloring text, and the user is likely to end up using some other color package in conjunction with PSTricks. But then the color names used for text cannot be used with PSTricks graphics parameters.

It is therefore important that a dominant set of color macros emerge in the TEX community, and that the macros allow the PostScript code for the declared colors to be accessible, in a standard way, by packages such as PSTricks. Version 0.94 of PSTricks is distributed with an independent set of color macros that may be a prototype for such a standard color package.

Arithmetic

One of the limitations of TEX is its lack of fast, floating-point arithmetic. It is possible to write routines for calculating, for examples, sines and cosines using TEX's integer arithmetic, but these are notoriously slow. Therefore, PSTricks offloads such arithmetic to PostScript, whenever possible.

Such offloading is not always possible because PostScript cannot send information to TEX. If TEX needs to know the result of some calculation, it must do the calculation itself. For example, suppose that one wants a macro that puts a triangle around a TEX box, analogous to LATEX's \fbox command. The macro can measure the TEX box, and pass these dimensions to a PostScript procedure via a \special. PostScript can then use its trigonometric functions to calculate the coordinates of the vertices of the triangle, and then draw the triangle. However, it may be important for TEX to know the bounding box of the triangle that is drawn, so that the triangle does not overlap surrounding text. In this case, TEX must do (slowly) the trigonometric calculations itself.

Pure graphics

A large chunk of PSTricks consists of graphics macros, which you can think of as a fancy replacement for LATEX's picture environment. The qualifier "pure" means that the graphics do not interact with TEX. For example, a rectangle is "pure", whereas a framed box is not.

A pure graphics object scans arguments and puts together the PostScript code *ps-code* for the graphics. When the code is ready, the object concludes with:

\leavemode\hbox{\pstverb{*ps-code*}}

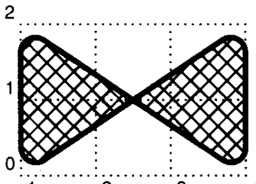
\pstverb should be defined in the configuration file to insert the code in a \special that reproduces *ps-code* verbatim in the PostScript file, grouped by PostScript's save and restore. The graphics state should have PostScript's standard coordinate system (bp units), but with the origin at TEX's current point. For dvips, the definition of \pstverb is:

```
\def\pstverb#1{\special{" #1}}
```

This \special is the only output generated. Thus, within TEX, the object produces a box with zero height, depth and width. Within PostScript, the graphics object is grouped by save and restore, and hence has no effect on the surrounding output.

For example, here is a polygon:

```
\pspolygon[linewidth=2pt,
  linearc=.2,fillstyle=crosshatch]
  (1,0)(1,2)(4,0)(4,2)
```



\pspolygon first invokes \pst@object, which collects (but does not processes) optional parameter changes, and subsequently invokes \pspolygon@i:

```
1  \def\pspolygon{\pst@object{pspolygon}}
2  \def\pspolygon@i{%
3    \begin@ClosedObj
4      \def\pst@cp{}%
5      \pst@getcoors[\pspolygon@ii}
```

\begin@ClosedObj (line 3) performs various operations that are common to the beginning of closed graphics objects (as opposed to open curves), such as processing the parameter changes and initializing

the command \pst@code that is used for accumulating the PostScript code. \pst@getcoors (line 5) processes the coordinates one at a time (\pspolygon can have arbitrarily many coordinates), converting each one to a PostScript coordinate and adding it to the PostScript code in \pst@code.

Then \pst@getcoors invokes \pspolygon@ii:

```
6  \def\pspolygon@ii{%
7      \addto@pscode{\psline@iii \tx@Polygon}%
8      \def\pst@linetype{1}%
9    \end@ClosedObj}
```

Line 7 adds the PostScript code that takes the coordinates from the stack and constructs the path of the polygon. \pst@linetype (line 8) is used by the dashed and dotted linestyles to determine how to adjust the dash or dot spacing to fit evenly along the path (the method is different for open curves and open curves with arrows). Then \end@ClosedObj (line 9) performs various operations common to the ending of closed graphics objects, such as adding the PostScript code for filling and stroking the path and invoking \pstverb.

Here is the resulting PostScript code for this example:

```
1  tx@Dict begin STP newpath 2 SLW 0 setgray
2  [ 113.81097 56.90549 113.81097 0.0
3  28.45274 56.90549 28.45274 0.0
4  /r 5.69046 def
5  /Lineto{Arcto}def
6  false Polygon
7  gsave
8    45. rotate 0.8 SLW 0. setgray
9    gsave 90 rotate 4.0 LineFill grestore
10   4.0 LineFill
11 grestore
12 gsave 2 SLW 0 setgray 0 setlinecap stroke
13 end
```

Line 1 is added by \begin@ClosedObj. STP scales the coordinate system from PostScript's bp units to pt units, which are easier for TeX to work with (e.g., \the\pslinewidth might expand to 5.4pt, and the pt can be stripped).

Lines 2 and 3 are the coordinates, which are added by \pst@getcoors.

Line 4 sets the radius for the rounded corners and line 5 defines Lineto, a procedure used by Polygon, so that it makes rounded corners. If the linearc parameter had been 0pt instead, then, instead of lines 4 and 5, \psline@iii would have added /Lineto{lineto}def.

Lines 7 to 11 are added by the fillstyle, and line 12 is added by the linestyle, both of which are invoked by \end@ClosedObj.

The code for the graphics objects is highly modular. For example, nearly all graphics objects invoke the fill style to add the PostScript code for filling the object. To define a new fill style foo for use with all such objects, one simply has to define \psfs@foo to add the PostScript code for filling a path.

The graphics objects can be used anywhere, and can be part of composite macros such as for framing text. However, they are most commonly used by the end-user to draw a picture by combining several such objects with a common origin. For this purpose, PSTricks provide the pspicture environment, which is very similar to LaTeX's picture environment. In particular, it is up to the user to specify the size of the picture. This is an unfortunate inconvenience, but one that is insurmountable. The PSTricks graphics objects include curves and other complex objects of which TeX could not calculate the bounding box, at least not without doubling the size of PSTricks and slowing it to a crawl. This is the main way in which TeX's lack of graphics and floating point capabilities hinders PSTricks.

Nodes

Drawing a line between two TeX objects requires knowledge of the relative position of the two objects on the page, which can be difficult to calculate. For example, suppose one wants to draw a line connecting "his" to "dog" in the following sentence:

The dog has eaten his bone.

One could calculate the relative position of these two words, as long as their is not stretchable glue in the sentence, but the procedure would not be applicable to connecting other objects on a page.

With PostScript as a postprocessor, there is a straightforward solution. By comparing the transformation matrices and current points in effect at two points in the PostScript output, one can determine their relative positions. This is the basic idea that lies behind PSTricks node and node connection macros, and is one that PSTricks adapted from Emma Pease's tree-dvips.sty.

Here is how PSTricks connects the words:

```
\large
The \rnode{A}{dog} has eaten
    \rnode{B}{his} bone.
\ncbar[angle=-90,nodesep=3pt,arm=.3]{->}{B}{A}
```

The dog has eaten his bone.

\rnode{A}{dog} first measures the size of "dog". Then it attaches to "dog" some PostScript code that creates a dictionary, TheNodeA, with the following variables and procedures:

NodeMtrx	The current transformation matrix.
X	The x-coordinate of the center.
Y	The y-coordinate of the center.
NodePos	See below.

Here is the code that appears in the PostScript output for this example:

```
1  tx@Dict begin
2    gsave
3      STV CP T
4      8.33331 2.33331 18.27759
5      9.1388 3.0
6      tx@NodeDict begin
7        /TheNodeA 16 NewNode
8          InitRnode
9        end
10     end
11   grestore
12 end
```

This codes gets inserted with \pstVerb, which should be defined in the configuration file to include *ps-code* verbatim in the PostScript output, *not* grouped by (g)save and (g)restore. PostScript's current point should be at TEX's current point, but the coordinate system can be arbitrary. For dvips, the definition of \pstVerb is:

```
\def\pstVerb#1{\special{ps: #1}}
```

\pstVerb is used instead of \pstverb because the latter groups the code in save and restore, which would remove the node dictionary from PostScript's memory. However, PSTricks still wants to work in pt units, and so STV scales the coordinate system.

Line 4 contains the height, depth and width of the dog. The next line (9.1388 3.0) gives the x and y displacement from where the code is inserted (on the left side of dog, at the baseline) to the center of dog. Actually, by "center" we mean where node connections should point to. This is the center by default, but can be some other position. For example, there is a variant \Rnode that sets this point to be a fixed distance above the baseline, so that a horizontal line connecting two nodes that are aligned by their baselines will actually be horizontal, even if the heights or depths of the two nodes are not equal.

NewNode, in line 7, performs various operations common to all nodes, such as creating a dictionary and saving the current transformation matrix. Then InitRnode takes the dimensions (lines 4 and 5) off the stack and defines X, Y and NodePos.

A node connection that wants to draw a line between a node named A and a node named B can go anywhere after the nodes, as long as it ends up in the dvi file after the nodes, and on the same page. The node connection queries the node dictionaries for the information needed to draw the line. In the example above, \ncbar needs to know the coordinate of the point that lies on the boundary of "his", at a $-90°$ angle from the center of node. After setting Sin and Cos to the sine and cosine of 90° and setting NodeSep to 0, the procedure NodePos in the TheNodeA dictionary returns the coordinates of this point, relative to the center of the node. The connection macro can then convert this to coordinates in the coordinate system in effect when the node connection is drawn, by retrieving and using NodeMtrx, X and Y from TheNodeA.

A node connection macro, after drawing the connection, should also save a procedure for finding the position and slope of a point on the line, so that labels can be attached to node connections. This task is similar to that of a node; it should save the coordinates of the node connection and the current transformation matrix and a procedure for extracting from this information a position on the node connection. Example 1 makes extensive use of labels.

There are many ways to position nodes, depending on the application. To create a diagram with arrows from one object to another, one can position the objects in a pspicture environment. For applications with more structure, one may want a more automated way to position nodes. PSTricks does not come with any high-level macros explicitly for commutative diagrams, but it does have a psmatrix environment for aligning nodes in an array, and this can be used for commutative diagrams. Example 1 shows psmatrix beings used for a graph. PSTricks also contains very sophisticated tree macros.

Overlays

To make overlays with SLITEX, for example, you have to use invisible fonts, and TEX has to typeset the slide once for each overlay. This makes it impossible to make overlays if a slide uses fonts other than the few for which invisible versions are available, or if the slide contains non-text material.

PSTricks uses a simple idea for creating overlays. Its operation is illustrated in Example 2. A box from which a main slide and overlays are to be created is saved, using the overlaybox environment. The \psoverlay{2} command in this box simply inserts the code

```
(2) BeginOL
```

and similar code at the end of the current TEX group to revert to the main overlay. BeginOL compares the string on the top of the stack to the PostScript variable TheOL. If it does not match, the output is made invisible. Otherwise, it is made visible. To print out overlay 2,

```
\putoverlaybox{2}
```

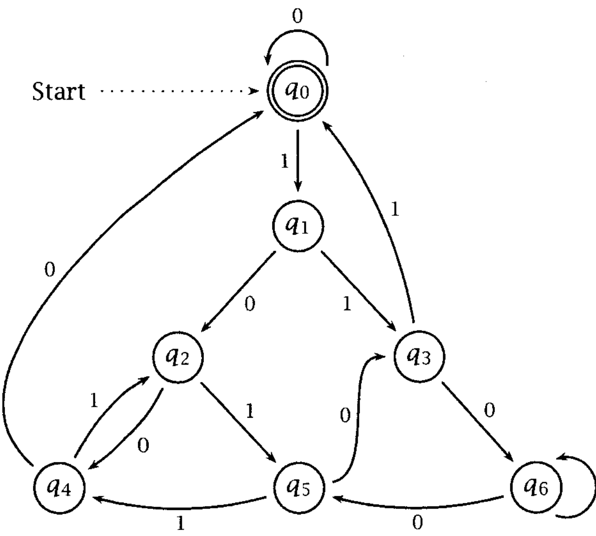
simply has to insert

```
/TheOL (2) def
```

before a copy of the box.

Because we can insert PostScript procedures in the box that can be redefined before each copy of the box, TEX only has to typeset the box once, which saves processing time and saves us from having to

```
\psmatrix
  [mnode=circle,colsep=.85cm,rowsep=1cm]
  % States:
  [mnode=R]{\mbox{Start}}
    & & [doubleline=true,name=0]$q_0$ \\
  & & [name=1]$q_1$ \\
  & [name=2]$q_2$ & & [name=3]$q_3$ \\[0pt]
  [name=4]$q_4$ & & [name=5]$q_5$ & &
       [name=6]$q_6$
\endpsmatrix
% Transitions:
\psset{nodesep=3pt,arrows=->,arcangle=15,
  labelsep=2pt,shortput=nab}
\footnotesize
\ncline[linestyle=dotted]{1,1}{0}
\nccircle{0}{.4cm}_{0}
\ncline{0}{1}_{1}
\ncline{1}{2}^{0}
\ncline{1}{3}_{1}
\ncarc{2}{4}^{0}
\ncarc{4}{2}^{1}
\ncline{2}{5}^{1}
\ncline{3}{6}^{0}
\ncarc{<-}{0}{3}^{1}
\nccurve[angleA=140,angleB=210]{4}{0}^{0}
\nccurve[angleA=10,angleB=180]{5}{3}^{0}
\ncarc{5}{4}^{1}
\ncarc{6}{5}^{0}
\nccircle[angleA=270]{6}{.4cm}_{1}
```



Example 1: An example of nodes and node connections and labels, used with the `psmatrix` environment. (Courtesy of Mark Livingston.)

```
\large
\begin{overlaybox}
  $\frac{n-2}{n-3}
    + \psframebox{\psoverlay{2}
   \frac{n-1}{n}}
    = \frac{2(n-2)(n-1)}{n(n-3)}$
\end{overlaybox}
\psset{boxsep=6pt,framearc=.15,
  linewidth=1.5pt}
\psframebox{\putoverlaybox{main}}
\psframebox{\putoverlaybox{2}}
```

$$\frac{n-2}{n-3} + \boxed{} = \frac{2(n-2)(n-1)}{n(n-3)}$$

$$\frac{n-1}{n}$$

Example 2: Overlays.

come up with a way to read the TEX input for the box several times.

There are several ways to make output invisible with PostScript, none of which is entirely satisfactory. PSTricks' default method is to translate everything far away (e.g., over by the coffee pot) so that, except in very unusual circumstances, all the "visible" output ends up off the page. This is easy to undo, by translating back.

The only problem with translation is that the node connections and labels, which use absolute coordinates, end up on the same overlay as the nodes that are connected. Therefore, users can select an alternate method for making material invisible: setting a small clipping path off the page. The problem with this method is that it can only be undone with `initclip`, which can mess up other macros that set the clipping path.

PSTricks does not use PostScript's `nulldevice` operator, because this cannot be undone except by using `grestore`. It would thus be impossible to have nested overlays. The PSTricks overlay macros are used to implement overlays in the Seminar package.

Typesetting text along a path

One facility that TEX users have long desired but have been unable to obtain is to typeset text along a path. This is a task that also stretches the limits of PostScript `\special`'s, but PSTricks contains an implementation that works for several dvi-to-ps drivers. It is illustrated in Color Example 13.

The main difficulty is that the text that goes along the path should be typeset by TEX, not by PSTricks, and then converted to PostScript output by the dvi driver. For PSTricks to get this text along the path, it has to redefine the operators that the dvi driver uses to print the text. This requires knowledge of the PostScript code the dvi driver uses to print text.

In the best case, the dvi driver simply uses PostScript's `show` operator, unloaded and unbound. PSTricks simply has to redefine `show` so that it takes each character in the string and prints it along the path. The redefined `show` checks the current point and compares it with the current point at the beginning of the box that is being typeset to find out the x and y positions of the beginning of the character. The x position is increased by half the width of the character to get the position of the middle of the character. This is the distance along the path that the middle of the character should fall. It is straightforward, albeit tedious, to find the coordinates and slope of any point on a path. We translate the coordinate system to this point on the path, and then rotate the coordinate system so that the path is locally horizontal. Then we set the current point to where the beginning of the character should be, which means to the left by half the character width and up or down by the relative position of the base of the character in the box. Then we are ready to show the character.

This method works with Rokicki's `dvips`. For other drivers, one of two problems arises:

1. `show` is "loaded" or "bound" in procedures defined by the driver for displaying text. This means that the procedures do not invoke the command name `show`, which can be redefined by PSTricks, but instead invoke the primitive operation `show`, which cannot be altered. The workaround for this is to remove the appropriate `load`'s and `bind`'s from the driver's header file.
2. The driver uses PostScript Level 2's large family of primitives for showing text. The only workaround is to redefine all these operators, which has not been attempted. The usual dvi drivers do not use Level 2 constructs. However, NeXTTEX's TEXView, which is a dvi driver based on the NeXT's Display PostScript windowing environment, does use Level 2 operators. The workaround for NeXT users is to use `dvips` to generate a PostScript file and then preview it with Preview.

Stroking and filling character paths

It is also possible to stroke and fill character paths, as illustrated in Color Example 13. The methodology is the same as typesetting text along a path, but it is easier because `show` just has to be changed to `charpath`. Nevertheless, the two problems that can trip up PSTricks' `\pstextpath` macro can also trip up `\pscharpath`. Furthermore, `\pscharpath` only works with PostScript outline fonts, since bitmap fonts cannot be converted to character paths.

Charts

PSTricks has many primitives for a wide variety of applications, but sophisticated graphics can involve tedious programming. In such cases, a preprocessor can be constructed to automatically generate the PSTricks commands. The preprocessor can generate standardized representations using only a minimum amount of information, but the user does not lose flexibility because the PSTricks code can subsequently be tweaked as desired.

For instance, we can think of preprocessors for automatic coloration of maps, generation of graphs or trees, etc. For his own needs, Denis Girou has written (in Shell and AWK) a preprocessor (`pstchart.sh`) for automatic generation of pie charts, which he extended to generate other forms of business graphics (line and bar graphs, 2D or 3D, stacked and unstacked).

Example 3 shows a data file, the unix command line for generating the PSTricks code from the data file, and the output. Color Example 14 shows the output from another example, generated with the unix command line:

```
pstchart.sh vbar dimx=9 3d boxit center\
  figure print-percentages < file2.data
```

Conclusion

There is much talk about the future of TEX and about the need to create a replacement for TEX because TEX is, by design, just a typesetting program for positioning characters and rules. We believe that when today's TEX is supplemented by PostScript, through the use of `\special`'s and good dvi-to-ps drivers, many of the special effects that users clamor for can be achieved today. PSTricks provides an example of this.

When PSTricks is combined with the Seminar LATEX2ε document class for making slides, plus PostScript fonts and macros for including graphics files, one has a complete presentation software package, that is quite far from the usual use of TEX for typesetting technical papers.

However, there are still some limitations that can only be solved by changes to TEX. The most obvious one is TEX's lack of fast, floating-point arithmetic. Although TEX can pass information to PostScript through `\special`'s, it is not possible for PostScript to pass information to TEX. This slows down many calculations and makes it impossible to calculate the bounding box of some graphics.

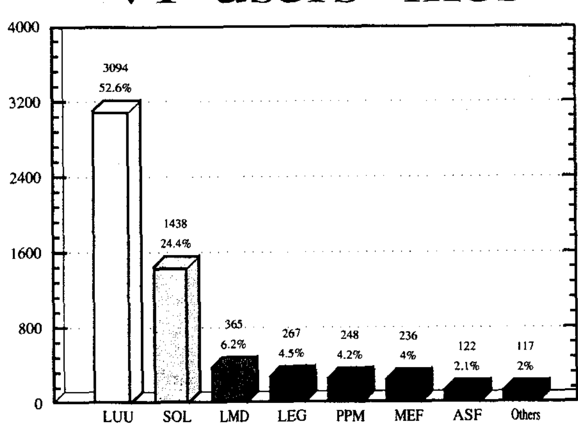
Data file:

```
3094 | LUU
1438 | SOL
 365 | LMD
 267 | LEG
 248 | PPM
 236 | MEF
 122 | ASF
  57 | DRT
  33 | AMB
  18 | TPR
   9 | RRS
```

Command line:

```
tchart.sh vbar dimx=7 3d nb-values=8 \
print-percentages print-values \
grayscale=white-black data-change-colors \
title="VP users' files" center <users.data
```

Output:



Example 3: Using the preprocessor pstchart to generate PSTricks graphs.

References

Girou, Denis. "Présentation de PSTricks," *Cahiers GUTenberg*, No. 16, pp. 21–70, Février 1994.

Van Zandt, Timothy. "PSTricks: Documented Code." 1994

Van Zandt, Timothy. "PSTricks: PostScript Macros for Generic TEX — User's Guide." 1994.

A LaTeX style file generator and editor

Jon Stenerson
TCI Software Research, Las Cruces, New Mexico
Jon_Stenerson@tcisoft.com

Abstract

This article presents a program that facilitates the creation of customized LaTeX style files. The user provides a style specification and the style editor writes all the macros. Editing takes place in a graphical user interface composed of windows, menus, and dialog boxes. While the editor may be used in any LaTeX environment, it is intended primarily for use with TCI Software Research's word processor Scientific Word.

The current style editor runs under any Windows 3.1 system. The performance is acceptable on a 386-based machine and naturally improves on 486's and Pentiums. As Scientific Word is ported to other systems so will the style editor be ported.

Introduction

The style editor is a program that facilitates the creation and modification of *styles*. It represents a style as a list of *generic markup tags*, and thinks of a tag as a list of *parameters* which determine its typesetting properties. It performs the basic operations of creating a new tag, modifying a tag's parameters, and deleting a tag. A tag's formatting instructions are not explicitly displayed. That is to say you do not see any TeX on the screen. Instead you see dialog boxes containing icons, menus, radio buttons, check boxes, and so forth. These prompt you to specify the style by filling in parameters and selecting options. There are some screen shots at the end of this article to give an idea of the style editor's general appearance.

Styles, generic markup tags, and Scientific Word

A *generic markup tag* is a device by which an author specifies a document's logical structure without specifying its visual format. For instance, the LaTeX tag `\section` conveys the information that a new section is beginning and that the tagged text is its title. By itself this has no implications for the *appearance* of the section heading. It does not tell us the heading's font, justification, or vertical spacing. A *style file*, external to the document, contains associations between the tag names and specific typesetting instructions. The style file says what tags exist and how text marked with those tags should be typeset. We see that the use of generic markup tags provides a certain division of labor. I write the article, someone else writes the style, and TeX and LaTeX do the typesetting. The only style information I need as an author is a list of tag names and instructions for their use.

These days most word processors do not make use of generic markup tags. The reason is that they want to be WYSIWYG (what you see is what you get). This means that they display on the video monitor exactly what you will get when you print the final copy. Files produced by WYSIWYG word processors are filled with explicit typesetting instructions like "put a 14pt Helvetica A at coordinates (100, 112)." Compare this approach with the generic markup approach. First, the division of labor mentioned above is lost and the author is now responsible for all typesetting decisions. Of course this is also the main attraction of such systems. Second, stylistic information is now duplicated throughout the document. If subsection headings have to be left justified rather than centered the author will have to track them all down and change them one by one.

At TCI Software Research we are trying the generic markup approach to word processing. Our word processor, Scientific Word, is not a WYSIWYG word processor in the usual sense. Instead it displays a document's text plus markup. The markup is graphical, rather than textual, in nature. Whereas in LaTeX you will see `\section{Introduction}`, in Scientific Word you will see the word Introduction in large blue letters on the video monitor. Ideally the document's text plus markup tags represents the entire content of the document. In practice there are some important exceptions where visual formatting carries a lot of information. For example, in mathematical equations and in tables the precise positioning of text contributes enormously to its meaning. Scientific Word is WYSIWYG to the extent that if the appearance of an object carries meaning, as in the case of an equation or table, then that object is displayed in an approximation to its printed form. When Scientific Word saves a document on a disk it

is saved in LaTeX form, and from there it can be typeset and printed.

Being "LaTeX-oriented" leaves Scientific Word open to some of the same criticisms leveled at LaTeX. In particular, authors do not always appreciate the division of labor I mentioned above. Some of them need or want more control over the style and cannot accept that someone else just hands them a style. At TCI we receive hundreds of requests for style modifications each year. Most of them are quite straightforward but many are not. It frustrates our customers, used to WYSIWYG systems, that some apparently trivial operations are not trivial for the casual LaTeX user. This suggests the need for another division of labor: style designer versus style writer. Our authors do not actually want to write styles, they want to specify styles. I was assigned the task of developing tools to alleviate this problem. The style editor represents the current state of that research.

For further discussion of markup tags and LaTeX see the first couple chapters of Goossens, Mittelbach, and Samarin 1994. For a discussion of generic markup in a non-LaTeX environment read about SGML (Standard Generalized Markup Language) (Bryan 1988).

The development process

Before continuing with the style editor itself I'd like to talk a little about the process of designing and implementing the editor. I was trained as an algebraic geometer in graduate school, had previously worked as a math professor, and this was my first professional programming experience. The process of programming is still novel enough to me that I feel like writing about it.

The first part of my research was to work with our customers in the capacity of style writer. I did this for four months to learn TeX, to learn how to think about style issues, and to find out what our customers wanted in the way of style modification. When I had enough experience to contemplate writing a program I e-mailed 500 customers and asked if anyone was interested in the design of a style editor. About 45 people responded and provided numerous comments and suggestions.

Still not knowing what a style editor should look like I decided to make a prototype, learn from my mistakes, and then build a release version.The prototype was implemented in three months between December 1993 and February 1994. It was complete enough to handle some realistic design issues even though it did not have a nice user interface. I wrote several styles with it including a style for one chapter of the new Scientific Word User's Guide.

In retrospect, I think that I spent the wrong amount of time on the prototype. The last few weeks of work on the prototype were spent getting it ready for testers - adding minor features, fixing bugs and writing documentation. As it turned out the testers paid little attention to the prototype editor. It was too primitive and too scary and I didn't get the feedback I'd hoped for. I either should have either gone ahead and made a nicer and more polished interface for the prototype, or I should have quit earlier and started on the release version editor sooner.

I learned many things from the prototype:

- Most importantly I learned that it is possible to develop a useful style editor. This was not obvious to me at first, but much of what I did worked better than I thought it would. I am now confident that TCI can and will develop a style editor that allows the casual user with no LaTeX knowledge to make basic style changes, and allows the advanced user to create any style at all.
- I learned that a lot more attention had to be paid to the user interface. I did not spend much time on the prototype's user interface because I had to first concentrate on getting the right model for the styles and getting the right basic functionality. For the style editor release version we added another programmer, Chris Gorman, to concentrate on getting the user interface in shape. He is responsible for much of the slick look and feel of the final program.
- Using the completed prototype to write some actual styles uncovered a number of flaws in the model I was using to represent styles.
- Writing the code for the prototype uncovered a number of flaws in my programming technique. Actually, many of these flaws were uncovered by John Mackendrick, one of our inhouse testers. I am a better programmer than I was six months ago. While the prototype always seemed a little flaky and buggy, the new program seems much more robust just by virtue of being better written.

Overall design

The style editor consists of the following components:

1. A GUI (Graphical User Interface). This manages interaction with the user and with the platform. The only platform Chris and I have worked on so far is Windows 3.1. We used Microsoft's Visual C++ and their MFC (Microsoft Foundation Classes) application framework. My understanding is that MFC code is supposed to eventually be portable to other platforms (Apple's Macintosh and Unix). So when Microsoft finishes MFC on those platforms we should be able to port the style editor.
2. A data structure called the Style. The program actually represents the style in two different

forms: internal and external. The internal form is a set of C++ classes suitable for editing. The nature of these classes lies outside the scope of this article. The external form is textual, suitable for interpretation as a LaTeX style and for human perusal. I frequently lump the internal and external data structures together into one abstract concept that I call the style *model*.

3. A set of functions called the "core". These functions do the following;
   (a) convert between the two style representations. In other words they read and write style files.
   (b) perform error checking. For example, before saving a style it makes sure that every item referred to in the style has been defined in the style.
   (c) constructs other files needed by Scientific Word. Besides the style file there is also a *shell* file and a *screen appearance* file. Each style file has a shell file that is used as a template whenever Scientific Word creates a new document of that style. The screen appearance file tells Scientific Word what tags are in the style and determines how they will appear on the video display.
4. A set of *macro writers*. These are TeX macros that interpret the style editor output as an actual style. They accomplish this by reading the style file and writing macros to implement the tags described in the style. This is all done on the fly. You will not normally see the macros written by the macro writers. They are constructed in the computer's memory and do not assume any printed appearance without inserting a `\show` command.

The key to the style editor is the last item so I'll talk about it some more. The *macro writers* are contained in a file called `sebase.cls`. This file is used as the document class for any style editor style. This is a misuse of the `.cls` extension because `sebase.cls` does not define any document class. Nor does it define any macros that may be used to markup a document. Rather it is a toolbox. The tools in `sebase.cls` are used to automatically write the macros that will be used in document markup. Eventually I will make a format file out of `sebase` but for now it depends on using the LaTeX format. Style files generated by the editor are read in with a `\usepackage` command.

Here is an example. In my scheme the definition of a section tag would look something like this:

```
\Division{
  \Name{section}
  \Level{1}
  \Heading{SectionHeading}
  \EnterTOC{true}
  \StartsOn{NextPage}
  \SetRightMark{true}
}
```

This is somewhat simplified but it gives the basic idea of what the style editor output might look like. In the file `sebase` there is a macro writer called `\Division` that writes a document division[1] macro on the basis of its parameters. In this case it writes a macro named `\section`. You see parameters describing the division's behavior with regard to the table of contents and running header and whether it must start on a new page, but you do not see any formatting instructions for a heading. This is because I distinguish between the division and its heading. There is just a reference to a heading. The heading itself is defined like this:

```
\DisplayElement{
  \Name{SectionHeading}
  \SkipBefore{20pt plus 4pt minus 2pt}
  \SkipAfter{12pt plus 2 pt minus 1pt}
  \ParagraphType{HeadingParagraph}
  \Font{MajorHeadingFont}
  \Components{
     Section \sectionCount .\Space{2mm}
     \CurrentHeading}
}
```

I have around 20 macro writers. Each of these is responsible for writing a certain *category* of macro. Thus I have a Division category, a Display Element category, a List category, a Font category, and so forth. These are discussed in more detail in the next section.

To get a feeling for how an editing session proceeds look at the screen shots at the end of this article. The first shows the start-up screen. You can see various controls for adjusting margins and page sizes. At the top of the screen is a menu labeled Category. The second screen shot shows the category menu pulled down and the division category about to be selected. You can see all of the categories. The third screen shot shows the screen after selecting the division category. Look at the split screen window. The left part of the window lists all the instances of the category that have been defined so far. In this case it lists all of the style's divisions: chapter, section, subsection and appendix. This list may be added to or deleted from. The figure also shows that "section" has been selected from the list of all divisions. The information for the section division is displayed in a dialog box contained in the right pane of the split window. This dialog changes radically depending on the category. One uses the controls found in that pane to inspect or alter the displayed

[1] I started using the term "division" because I found it awkward to continually refer to sections, subsections, and chapters as "sections".

parameters. When the style is saved the information is written in a form similar to that shown above.

Parametrized macro writing is not a new idea. For example, in the code for the LaTeX format there is a macro called `\@startsection`. This macro is used to define sectioning macros. It has a number of parameters and by specifying various values for these parameters one defines a wide variety of sectioning commands. Here is a typical definition of the `\section` tag from a human authored style file:

```
\def\section{
  \@startsection {section}{1}{\z@}
       {3.5ex plus 1ex minus .2ex}
       {2.3ex plus .2ex}{\large\bf}}
```

Only a dedicated person can remember what those parameters do, or that if one is negative it has a different meaning than if it is non-negative. On the other hand I have noticed that many styles override `\@startsection` itself, suggesting that it may not have enough parameters! In addition to borrowing ideas from LaTeX I have found that Bechtolsheim's *TeX in Practice* (Bechtolsheim 1993) is an excellent source of ideas for parametrized macros.

The idea of macro-writing macros is also not new. A trivial example is the `\title` macro found in LaTeX styles. It is defined like this:

```
\def\title#1{
  \def\@title{#1}
}
```

It takes a parameter and uses it to write another macro.

Victor Eijkhout's Lollipop format (Eijkhout 1992) is an example of a complete system of macro writing tools. I have not had an opportunity to use Lollipop but from the article I suspect that it would be possible to put a user interface on it similar to the one used with sebase. I thank the anonymous reviewer of this article for pointing out the existence of Eijkhout's work. I am a relative newcomer to TeX and was not aware of Lollipop but it is clearly related to what I am doing. Since I don't know Lollipop I will quote verbatim an example from the reference showing how a subsection heading might be created in that system:

```
\DefineHeading:SubSection counter:i
    whitebefore:18pt  whiteafter:15pt
    Pointsize:14  Style:bold
    block:start  SectionCounter literal:,
      SubsectionCounter literal:.
      fillupto:levelindent title
    external:Contents title external:stop
    Stop
```

You can see that this uses the idea of defining macros by specifying parameters in the form of keyword plus value.

A model for styles

Preliminaries. I think that a good piece of software must be based on a clean and straightforward model. In the case of the style editor this means finding an abstract representation of a style. My initial reaction after learning TeX and thinking about styles for a while was that it was not possible to write a style editor. There seemed to be so much disorganized "stuff" that I had no idea where to start. Had I started programming at this point I probably would have picked for my model a particular style file, say `article.sty`, and my program would have been an expert at editing all of the parameters and options found in this file. Instead I had a few "modelling" talks with Roger Hunter (TCI's president) and Andy Canham (development team leader). The model that came out of those meetings was implemented in the prototype and was subsequently modified for the release version based on that experience.

I said before that the model has two concrete representations: one as a C++ class, the other as a style file. The latter is probably more familiar to the reader so we will identify the style file with the style model. The remainder of this section talks about style files written by the style editor. The main idea behind style editor style files is that they contain no algorithmic information. There are no sequences of instructions, no branches, and no loops. They consist only of a long list of declarative information. Style editor style files use a very uniform syntax for this declarative data and therefore look different from other style files.

The style file consists of a list of declarations. The syntax for a declaration is always the same:

```
\CategoryName{
   \Parameter1{value 1}
   \Parameter2{value 2}
   ...etc...
}
```

Every category requires a fixed number and type of parameters. Parameters are discussed in the following subsection, and categories in the subsection after that.

The samples shown below are simplified. Actual style editor files contain information related to the operation of the style editor program. They also contain multiple versions of style data related to features described in the section on the user interface. I will suppress these kinds of data in the following discussion.

There is nothing proprietary about style editor style files. Anyone can go in with an ASCII editor and make changes to them without the style editor. For that matter anyone can write an entire style editor style file without using the style editor.

Parameters. Every parameter expects a value of a particular type. I've found the following types of parameters to be adequate:

1. A word parameter requires a string of letters (A–Z, a–z). These are usually references to macros.
2. A text parameter requires a string of characters. Any characters that are not among TeX's special characters (like braces and dollar signs) are allowed.
3. A boolean parameter requires one of the two words "true" or "false".
4. A numeric parameter requires a signed decimal number.
5. A dimension parameter requires a dimension in the TeX sense of a number plus a unit. The style editor knows all of the TeX units and can convert between them.
6. A glue parameter requires a glue value in the TeX sense of a natural dimension with a stretch dimension and a shrink dimension.
7. A component list parameter requires a list of *components*. Each component is either text in the sense given above, or a control sequence which is called a reference component in the style editor.

Some of these were demonstrated in the previous section's example of a `\Division`: `\Heading` is a word parameter, `\EnterInTOC` is a boolean parameter, and `\Level` is a numeric parameter. Next look at the `\DisplayElement` example also in the previous section. `\SkipBefore` and `\SkipAfter` are glue parameters and `\Components` is a component list parameter. The value of `\Components` in the example consists of five components: two text components "Section " and ".", and three reference components.

Categories. Now we'll take a look at some of the other categories that the style editor knows about. There are more categories than I can describe even briefly so I'm just going to try get across a few ideas about how it all fits together. In particular we will not see categories that define Lists, Table of Contents, Index, Bibliography, or Math. These perform fairly specialized functions and after reading what follows you may be able to imagine their nature.

Document Variables. These are macros that the document uses to pass information back to the style. A typical example is a macro to handle the document's title:

```
\DocumentVariable{
  \Name{Title}
  ...
}
```

A document variable's most important parameter is its name. It actually has a couple more parameters that have to do with Scientific Word's handling of the variable. The macro writer, `\DocumentVariable`, writes a macro called `\SetTitle`. The `\SetTitle` macro is used in the document like this:

```
\SetTitle{My TUG paper}
```

This in turn defines a macro `\Title` whose replacement text is `My TUG paper`. Thus `\SetTitle` and `\Title` have the same relation to each other as `\title` and `\@title` have in LaTeX.

The style editor also knows about several built-in macros that get information from the document. These include `\PageNum`, and `\CurrentHeading`. These keep track of the current page number and the title of the most recently encountered division.

Fonts. The font category provides an interface to NFSS. Here is a sample style file entry:

```
\FontNFSS{
  \Name{BodyTextFont}
  \Family{Serif}
  \Shape{Upright}
  \Series{Medium}
  \Size{normalsize}
}
```

`\FontNFSS` will write a macro, `\BodyTextFont`, which performs the indicated font switch. The precise nature of the various families, shapes, series, and sizes are determined by selecting a "Font Scheme" elsewhere in the style.

Paragraphs and Environments. The paragraph category provides an interface to a number of TeX parameters related to paragraph typesetting: font, baseline-to-baseline distance, indentations and so forth. By setting these properly you can create tags like the `\quote` and `\center` found in LaTeX. Here is an example:

```
\Paragraph{
  \Name{Center}
  \Font{BodyTextFont}
  \ParIndent{0pt}
  \LeftIndent{0pt plus 1fil}
  \RightIndent{0pt plus 1fil}
  \ParFillSkip{0pt}
  \ParSkip{0pt}
  \PageBreakPenalty{100}
  \HyphenationPenalty{100}
}
```

When used in conjunction with an environment category item this will make available in the document an environment `\begin{Center} ... \end{Center}` that typesets a prefix, such as a vertical skip, then switches to the centering paragraph, and then has a suffix.

In-line and display elements. An in-line element is just a component list plus a font. It is intended to typeset text which is part of a surrounding paragraph. Here is an example:

```
\InlineElement{
```

```
  \Name{AbstractLeadin}
  \Font{SmallCapFont}
  \Components{Abstract.\Space{1pc}}
}
```

This creates a macro, \AbstractLeadin, which typesets the word "Abstract" followed by a period and a space. It uses a font called \SmallCapFont which must be defined via the Font category. Component lists may use the names of in-line elements so this \AbstractLeadin item may be reused throughout the style.

A display element is intended to be typeset in its own paragraph and set off from the surrounding text. We have already seen an example of this earlier in the section.

In-line and display elements are frequently used in conjunction with a document variable. For example, consider generating a macro to typeset the title of the document. We would first declare a document variable to hold the title

```
\DocumentVariable{
  \Name{Title}
}
```

and then declare a display element that uses the document variable

```
\DisplayElement{
  \Name{TITLE}
  \SkipBefore{0pt}
  \SkipAfter{0pt}
  \Paragraph{CenterHeading}
  \Font{MajorHeadingFont}
  \Components{\Title}
}
```

This produces a macro called \TITLE that typesets the value of the variable \Title with the given paragraph and font settings. The \TITLE macro may be used in the document but will probably be used in a title page macro (see below).

Page Setup. This category provides an interface to many TEX parameters involved in page style: page size, trim size, margins, headers and footers, footnotes and margin notes. Most styles will need to create only one item in the page setup category.

Exceptional Pages. An exceptional page is one that deviates from the surrounding pages in that it has some special formatting requirements. A typical example is a title page. A title page has some specially typeset material and usually has special headers and footers. Here is an example:

```
\Exception{
  \Name{TitlePage}
  \VerticalMaterial{
     \Space{2cm}
     \TITLE
     \Space{1cm}
     \AUTHOR
     \DATE
     \Space{1cm}
  }
  \ContinueTextOn{ThisPage}
  \SpecialLeftHead{}
  \SpecialMiddleHead{}
  \SpecialRightHead{}
  ...etc...
}
```

This writes a macro called \TitlePage which in turn causes a new page to begin, typesets the vertical material, and then allows text to continue on this page. The vertical material consists of built-in macros such as \Space or names of elements defined elsewhere in the style such as \TITLE, \AUTHOR, and \DATE.

The user interface

The prototype editor had a simple interface. In essence there were dialog boxes in one-to-one correspondence with the macro writers and in each dialog box there were controls in one-to-one correspondence with the macro writer's parameters. To a TEX programmer this interface would probably seem pretty friendly. If you saw an edit control labeled "Par. Skip" you'd probably have a good idea of the sort of thing you might enter. Editing with the prototype was not that far removed from editing the style file with an ASCII editor. The major step forward was the ease with which you could move around the style. I'm sure that all TEX programmers have had the experience of searching style files for a macro definition. The prototype style editor could find any piece of data instantly.

Most of our customers however do not want to fill in parameters. They do not want to know what glue is. They do not even want to see the word "skip" on the screen. They want to use the mouse to click on a picture of what they want, check a few boxes or radio buttons, and have the program do the right thing. On the other hand I liked the prototype's powerful interface and was not willing to give it up. So I opted for a hybrid scheme. A category item can now have two different interfaces: a "quick screen" in which a few simple options are presented, and a "custom screen" which presents all the category's parameters. The quick screen for the Paragraph category has several sets of icons. By selecting an icon from each set you determine certain characteristics of the paragraph. For example, one set is labeled "Paragraph spacing" and it contains two icons. One icon suggests tight spacing, the other suggests loose spacing. The custom screen by contrast has several places where actual dimension and glue values must be given. To prevent casual users from stumbling into dialogs they don't understand the program has two modes. In the first mode many features including all the custom screens are disabled.

As described so far the quick screen seems limited. It has two icons for paragraph spacing, but what glue values should correspond to these two icons? It clearly depends on the style. I therefore decided to let the icons themselves be programmable. By selecting an icon and pressing the F2 key you get a dialog box where a specific glue can be given. This value is saved in the style file. Finally, if you can't get the effect you want from the quick screen, the quick screen F2 modifications, or the custom screen, you can tell the style editor that you want to write this macro yourself. You will then have to do so in another macro file.

Conclusion

The style editor as it now stands is a useful program but there is still a lot of work to be done before it is a complete program. What I anticipate in the near future is that a style writer will prepare a style using the style editor together with a little straight TeX to fill in the gaps. The resulting style, at least those parts that do not rely on the plain TeX additions, can be customized by the author without any TeX knowhow. As time goes by I will manage to get more and more TeX into the editor's quick screens and there will be fewer and fewer gaps.

I have more basic functionality planned. For instance, I want to include a fancy "cut and paste" feature that will facilitate moving tag definitions from one style to another. The editor will resolve internal naming conflicts and make sure that auxiliary definitions needed for the tags being moved are moved at the same time. Having an abstract style representation should make it possible to move features from style to style. This in turn will make it possible to "change styles". A frequent customer request is to change a document from one style to another. If the two styles have the same set of markup tags this is pretty easy. If they do not this is pretty hard. If the style editor can reliably move tags from one style to another then this problem will be solved.

Shortly before the TUG meeting I received preprints of two other papers, Baxter 1994 and Ogawa 1994, that are found elsewhere in these proceedings. These talk are about using the object-oriented paradigm in TeX programming and in document markup. In some ways the style editor is also part of this discussion on the object-oriented approach. In fact the style editor directly represents the style as a C++ class in which each generic markup tag acts as a "style object" that can be acted upon by an object-oriented interface. I think that combining a style editor of the sort I've described here with a markup scheme such as described in the above references would lead to quite a powerful typesetting system.

References

Baxter, William Erik. "An Object-Oriented Programming System in TeX." These proceedings.

Bechtolsheim, Stephan von. *TeX in Practice*. Springer-Verlag, New York, NY, USA, 1993.

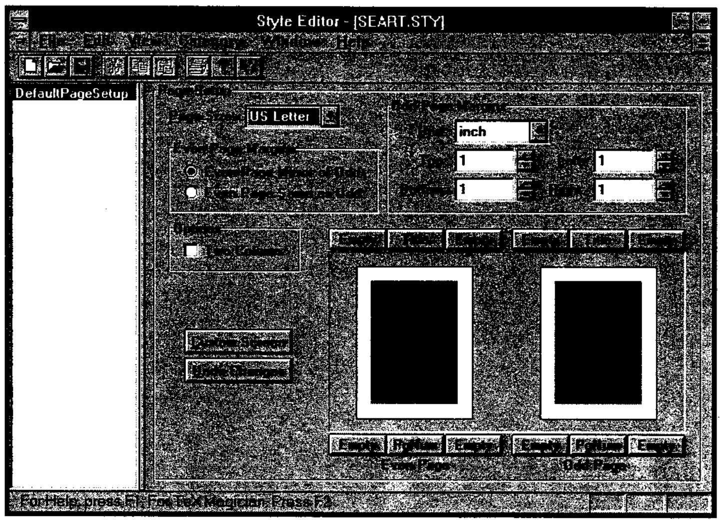
Eijkhout, Victor "Just give me a lollipop (it makes my heart go giddy-up)." *TUGboat* **13** (3), pages 341–346, 1992.

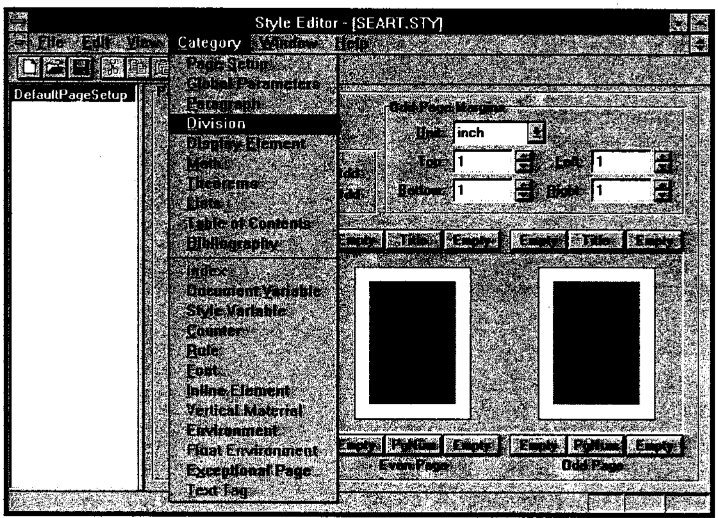
Goossens, Michel, Frank Mittelbach and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, MA, USA, 1994.

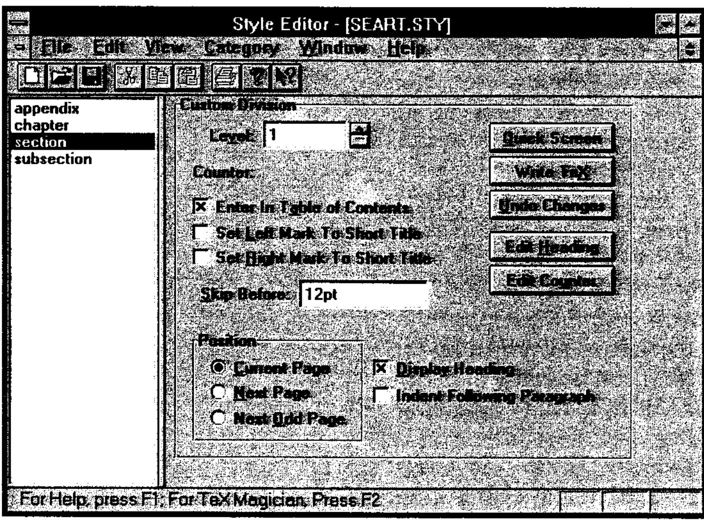
Mittelbach, Frank. "An extension of the LaTeX theorem environment." *TUGboat* **10** (3), pages 416–426, 1989.

Ogawa, Arthur. "Object-Oriented Programming, Descriptive Markup, and TeX." These proceedings.

Bryan, Martin. *SGML: an Author's Guide*. Addison-Wesley, Reading, MA, USA, 1988.







Examples of user interface screens.

Document Classes and Packages for LaTeX2ε

Johannes Braams
PTT Research, P.O. box 421, 2260 AK Leidschendam, The Netherlands
J.L.Braams@research.ptt.nl

Abstract

The first section of this article describes what document classes and packages are and how they relate to LaTeX 2.09's style files. Then the process of upgrading existing style files for use with LaTeX2ε is described. Finally there is an overview of standard packages and document classes that come with the LaTeX2ε distribution.

Introduction

This article is written for people who have written document styles for LaTeX 2.09 and want to upgrade them for LaTeX2ε. For a description of the new features of the user level commands, see *LaTeX2ε for authors* (in the file `usrguide.tex` in the LaTeX2ε distribution). The details about the interface for class and package writers can be found in *LaTeX2ε for class and package writers* (in the file `clsguide.tex`). The way LaTeX now deals with fonts is described in *LaTeX2ε font selection* (in the file `fntguide.tex`).

What are document classes and packages?

LaTeX is a document preparation system that enables the document writer to concentrate on the contents of his text, without bothering too much about the formatting of it. For instance, whenever he starts a new chapter the formatting of the chapter is defined outside of his document. The file that contains these formatting rules used to be called a 'document style'. Such a document style can have options to influence its formatting decisions. Some of these options are stored in separate files, 'document style option' files. An example of such option files is `fleqn.sty` which was part of the LaTeX 2.09 distribution. This option changes one aspect of the formatting of a document —it makes displayed equations come out flush left instead of centered.

There are also extensions to LaTeX that implement constructs that are not available in the default system, such as `array.sty`. These extensions are also known as 'document style option' files, although they can often be used with many kinds of documents.

To make a better distinction possible between these two kinds of 'options' new names have been introduced for them. What used to be called a 'document style' is now called a 'document class'[1]. Extensions to the functionality of LaTeX are now called 'packages'.

Options, options, options... Like the document styles of LaTeX 2.09 document classes can have options that influence their behaviour—to select the type size for instance. But with LaTeX2ε it is now also possible for packages to have options. As a consequence there are now two kinds of options, 'local options'—which are only valid for the package or document class they are specified for—and 'global' options which can influence the behaviour of both the document class and one or more packages. As an example of this let's consider a document written in German. The author chooses to use the `babel` package. He also wants to be able to refer to a figure 'on the following page' so he uses the `varioref` package. The preamble of his document might then look like:

```
\documentclass{article}
\usepackage[german]{babel}
\usepackage[german]{varioref}
...
```

As you see the option 'german' was specified twice. Using a 'global option' this preamble could be changed to read:

```
\documentclass[german]{article}
\usepackage{babel}
\usepackage{varioref}
...
```

This way it is known to the document class as well as *all* packages used in the document that the option 'german' is specified.

Command names. This new version of LaTeX comes with a new set of commands. Those LaTeX users who have written their own extensions to LaTeX in the past know that in version 2.09 basically two types of commands existed, namely "internal" commands—with '@'-signs in their name—and "user level" commands—without '@'-signs in their name.

LaTeX2ε has also commands that have both upper- and lowercase letters in their name. Those commands are part of the interface for package and

[1] This also gives a possibility to distinguish between documents written for LaTeX 2.09 and documents written for LaTeX2ε.

cls	A file containing a document class
clo	A file containing an external option to a document class
sty	A file that contains (part of) a package
cfg	An optional file that is looked for at runtime and which can contain customization code
def	A file containing definitions that will be read in at runtime.
ltx	A file used when building the LaTeX2ε format
dtx	Documented source code for .cls, .clo, .sty, .cfg, .def, and .ltx files
fd	A font definition file
fdd	Documented source code for .fd files
ins	DOCSTRIP instructions to unpack .dtx and .fdd files

Table 1: Extensions for LaTeX2ε files

class writers. They are not intended for use in documents, but they are meant to provide an 'easy' interface to some of the internals of LaTeX2ε.

Filenames. The new version of LaTeX introduces a number of new file extensions. This makes it easy to distinguish between files that contain a Document Class, files that contain an external option to a Document Class and files that contain Packages. In table 1 you can find an overview of the extensions that have been introduced. I would suggest that you would stick to the same set of extensions when you upgrade your old .sty files.

Upgrading existing 'styles' — general remarks

Is it a class or a package? The first thing to do when you upgrade an existing style file for LaTeX2ε, is to decide whether it should become a document class or a package. Here are a few points which might help you to decide what to do with your .sty file.

- Was the original .sty file a documentstyle? Then turn it into a document class.
- Was the original .sty file meant to be used for a certain type of document? In that case you should consider turning it into a document class, possibly by building on top of an existing class. An example of this is proc.sty which is now proc.cls.
- Was it just changing some aspects of the way LaTeX does things? In that case you would probably want to turn your .sty file into a package.
- Was it adding completely new functionality to LaTeX? Examples of this kind of .sty file are files such as fancyheadings.sty and XYpic.sty. This you most certainly will want to turn into a package for LaTeX2ε.

Style options → packages

Trying it out unchanged. After you've decided to produce a package file, you should first try to run a document that uses your .sty file through LaTeX2ε unmodified. This assumes that you have a suitable test set that tests all functionality provided by the .sty file. (If you haven't, now is the time to make one!) The experience of the last months has shown that most of the available .sty files will run with LaTeX2ε without any modification. Yet if it does run, please enter a note into the file that you have checked that it runs and resubmit it to the archives if it was a distributed file.

Bits that might have failed. Some .sty files will need modification before they can be used successfully with LaTeX2ε. Such a modification is needed for instance when you used an internal macro from the old font selection scheme. An example is \fivrm which is used by some packages to get a small dot for plotting. The obvious solution for this seems be to include a definition such as:

```
\newcommand{\fivrm}
      {\normalfont
       \fontsize{5}{6.5pt}\selectfont}
```

But that involves a lot of internal processing and may result in long processing times for your documents that use this. For this purpose the command \DeclareFixedFont is available. It bypasses a lot of the overhead of the font selection scheme. Using this command the solution becomes:

```
\DeclareFixedFont{\fivrm}
                 {OT1}{cmr}{m}{n}{5}
```

This tells LaTeX that the command \fivrm should select a font with OT1 encoding, cmr family, medium weight, normal shape and size 5 point.

Pieces of code that might need checking. If your .sty file uses commands that used to be part of the way LaTeX used to deal with fonts than your file will almost certainly *not* work. You will have to look in *LaTeX2ε font selection* or *The LaTeX Companion* (Goossens et al. 1994) to find out the details about what needs to be done.

Commands such as \tenrm or \twlsf have to be replaced:

```
\tenrm  →  \fontsize{10}{12pt}\rmfamily
\twlsf  →  \fontsize{12}{14.5pt}\sffamily
```

Another possibility is to use the rawfonts package, described in *LaTeX2ε for Authors.*

Also commands such as \xipt do not exist any longer. They also have to be replaced:

```
\vpt   →  \fontsize{5}{6.5pt}\selectfont
\xipt  →  \fontsize{11}{13.6pt}\selectfont
```

LaTeX 2.09 used commands with names beginning with \p for 'protected' commands. For example, \LaTeX was defined to be \protect\pLaTeX, and \pLaTeX produced the LaTeX logo. This made \LaTeX robust, even though \pLaTeX was not. These commands have now been reimplemented using \DeclareRobustCommand (described in *LaTeX2ε for class and package writers*). If your package redefined one of the \p-commands, you should replace the redefinition by one using \DeclareRobustCommand.

When you use internal commands from NFSS version 1 you will have to be very careful to check if everything still works as it was once intended.

Note that macros such as \rm are now defined in class files, so their behaviour may differ for each class. Instead you should use the lower level commands such as \rmfamily in packages. When you want to make sure that you get a certain font, independent of the environment in which your macro is activated, you can first call \normalfont and then switch the various parameters of the font selection scheme as necessary.

In some cases you may need to use the user level commands such as \textrm. This is necessary for instance when you define a command that may also be used in mathmode.

Document styles → Classes

Minimal updates are necessary. When you are upgrading a document style to a document class there are a few things that you really *have* to change, or your class will not work.

One of the things that must be done, is making sure that your class doesn't define \@normalsize but \normalsize. Make sure that \renewcommand is used to redefine \normalsize as it is already defined in the kernel of LaTeX, but to produce a warning that it needs to be given a real definition.

Another aspect that needs to be dealt with, is that the parameters \@maxsep, \@dblmaxsep and \footheight no longer exist. The first two were part of the float placement algorithm, but a change in that algorithm made them superfluous. The parameter \footheight was reserved in LaTeX 2.09, but it was never used.

The declarative font changing commands (\rm, \sf etc.) are no longer defined by default. Their definitions have been moved to the class files. Make sure that you define them or that they are not used by the users of your class. The standard document classes all contain definitions such as the following:

```
\DeclareOldFontCommand{\rm}
        {\normalfont\rmfamily}{\mathrm}
```

This tells LaTeX that when \rm is used in the text it should switch to \normalfont and then select the roman family. When \rm is used in mathmode LaTeX will select the font that would be selected by \mathrm[2].

[2] See *LaTeX2ε font selection* for more details.

Build on standard classes. When upgrading your own document style you should consider to reimplement it by building on an existing Document Class. With the new features of LaTeX2ε this has become very easy. The advantage of this approach is that you don't have to maintain a whole lot of code that is probably basically a copy of the code in one of the standard document classes. (See below for a few examples of how to build your own document class on an existing class.) Some documentstyles written for LaTeX 2.09, such as ltugboat, contain a command such as \input{article.sty}. This was the only solution in LaTeX 2.09—to build a new documentstyle upon an existing style. But, there was no way of ensuring that the file article.sty which was found by LaTeX wasn't out of date. As you see in the examples below, it is now possible to ensure that you use a version of article.cls that was released after a certain date.

Suggested updates. Apart from the essential changes to your document class, there are also a few changes that you are encouraged to make. Most of these changes have to do with the new possibilities the package and class writers interface gives you.

In a LaTeX 2.09 document style an option was declared by defining a command that starts with \ds@ followed by the name of the option. Later on in the documentstyle the command \@options was called to execute the code for the options that were supplied by the user. For example, the document style article contained the following lines of code:

```
...
\def\ds@twoside{\@twosidetrue
          \@mparswitchtrue}
\def\ds@draft{\overfullrule 5\p@}
...
\@options
...
```

This code fragment defined two options, twoside and draft.

The same effect can be achieved by using LaTeX2ε syntax, as is shown by the following code fragment from the document class article:

```
...
\DeclareOption{oneside}
       {\@twosidefalse \@mparswitchfalse}
\DeclareOption{twoside}
         {\@twosidetrue  \@mparswitchtrue}
\DeclareOption{draft}
            {\setlength\overfullrule{5pt}}
\DeclareOption{final}
            {\setlength\overfullrule{0pt}}
...
\ProcessOptions
```

As you can see, the intention of this code is easier to understand.

I consider it good practice, when writing packages and classes, to use the higher level LaTeX commands as much as possible. So instead of using `\def...` I recommend using one of `\newcommand`, `\renewcommand` or `\providecommand`. This makes it less likely that you inadvertently redefine a command, giving unexpected results.

When you define an environment use the commands `\newenvironment` or `\renewenvironment` instead of `\def\foo{...}` and `\def\endfoo{...}`.

If you need to set or change the value of a ⟨*dimen*⟩ or ⟨*skip*⟩ register, use `\setlength`.

The advantage of this practice is that your code is more readable and that it is less likely to break when future versions of LaTeX are made available.

Some packages and document styles had to redefine the `\begin{document}` or `\end{document}` commands to achieve their goal. This is no longer necessary. The "hooks" `\AtBeginDocument` and `\AtEndDocument` are now available. They make it more likely that your package will work together with someone else's.

When a document class needs to pass information to the user, you can use one of the commands `\ClassInfo`, `\ClassWarning`, `\ClassWarningNoLine` or `\ClassError`. A similar set of commands exists for packages.

Be colour safe. One of the new features of LaTeX2ε is the support for coloured documents. To create a document that contains colour you need:

- the `color` package, which is part of the LaTeX2ε distribution;
- a driver which supports colour—dvips by Tomas Rokicki is an example of such a driver;
- colour safe macros.

The first two points are probably obvious, the third point needs some explanation. TeX has no knowledge of colour, therefore the macros need to keep track of the colour. To achieve that, various changes have been made to the kernel of LaTeX. This has been done in such a way that the changes are 'dormant' when the `color` package isn't used. As an example, here is the current definition[3] of the LaTeX command `\sbox`:

```
\def\sbox#1#2{\setbox#1\hbox{%
  \color@@setgroup#2\color@@endgroup}}
```

The extra level of grouping is activated by the `color` package and is needed to keep colour changes local. For more information about being 'color safe' you should read the documentation that comes with the `color` package.

If you use the LaTeX commands for boxing sunch as `\mbox`, `\sbox`, `\fbox`, etc. instead of the low level commands `\hbox`, `\vbox` and `\setbox`, your code will be automatically 'colour safe'.

[3] Shown here only as an illustration; the actual implementation may change.

Upgrading existing 'styles'—an example tour

A minimal class. Most of the work of a class or package is in defining new commands, or changing the appearance of documents. This is done in the body of the class or package, using commands such as `\newcommand`, `\setlength` and `\sbox` (or `\savebox`).

However, there are some new commands for helping class and package writers. These are described in detail in *LaTeX2ε for class and package writers.*

There are three definitions that every class *must* provide. These are `\normalsize`, `\textwidth` and `\textheight`. So a minimal document class file is:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{minimal}
              [1994/06/01 Minimal class]
\renewcommand{\normalsize}{%
  \fontsize{10}{12}\selectfont}
\setlength{\textwidth}{6.5in}
\setlength{\textheight}{8in}
```

However, most classes will provide more than this!

Extending a class with new commands. The first example shows how you can extend an existing class with a few extra commands. Suppose you call your new class `extart`. It could start off with the following code:

```
%------------ Identification ------------%
\NeedsTeXFormat{LaTeX2e}[1994/06/01]
\ProvidesClass{extart}
    [1994/08/01 v2.0j
     Article like class with new commands]
```

This first line tells LaTeX that your code was written for LaTeX2ε, released after june first, 1994. The second line informs LaTeX that this file provides the document class `extart`, dated august 1, 1994, and with version 2.0j.

```
%------------ Option handling -----------%
\DeclareOption*{%
  \PassOptionsToClass{\CurrentOption}
                     {article}}
```

The code above instructs LaTeX to pass on every option the user asked for to the document class `article`.

```
\ProcessOptions
%------------ Load other class ----------%
\LoadClass[a4paper]{article}[1994/06/01]
```

The command `\ProcessOptions` executes the code associated with each option the user specified. The `\LoadClass` command subsequently loads the class file. The first optional argument to `\LoadClass` passes the option a4paper to the class; the second optional argument to `\LoadClass` asks for `article.cls` dated june first, 1994, or later.

Note that if you change your mind and load `report` instead you also have to change the second argument of `\PassOptionsToClass`.

```
%----------- Extra command -------------%
\newcommand\foo{\typeout{Hello world!}}
...
```

The rest of the file contains the extra code you need such as the definition of the command `\foo`.

Changing the layout produced by another class. The first few lines of a class that modifies the layout of an existing class would look much the same as in the example above.

```
%----------- Identification ------------%
\NeedsTeXFormat{LaTeX2e}[1994/06/01]
\ProvidesClass{review}
  [1994/08/01 v1.0
   Article like class with changed layout]
%----------- Option handling -----------%
\DeclareOption*{%
  \PassOptionsToClass{\CurrentOption}
                     {article}}
\ProcessOptions
%----------- Load other class ----------%
\LoadClass{article}[1994/06/01]
```

Suppose we have to print on paper 7 inch wide and 9.875 inch tall. The text should measure 5.5 inch by 8.25 inch

```
%------------ Layout of text ----------%
\setlength{\paperwidth}{7in}
\setlength{\paperheight}{9.875in}
\setlength{\textwidth}{5.5in}
\setlength{\textheight}{8.25in}
```

What we have to do now is position the body of the text in a proper place on the paper.

```
\setlength{\topmargin}{-.5625in}
\setlength{\oddsidemargin}{-.25in}
\setlength{\evensidemargin}{-.25in}
\setlength{\marginparwidth}{.25in}
\setlength{\headsep}{.1875in}
```

We could go on and modify other aspects of the design of the text, but that is beyond the scope of this article.

Extending a class with new options. As before, we start the document class with some identification.

```
%----------- Identification ------------%
\NeedsTeXFormat{LaTeX2e}[1994/06/01]
\ProvidesClass{optart}
  [1994/08/01 v1.0
   Article like class with extra options]
```

Suppose you want to be able to print a document in 9pt type, or when you want to be loud, print it in 14pt type. You know that the standard LaTeX classes contain the command
`\input{size1\@ptsize.clo}`
just after the execution of `\ProcessOptions`. Supposing you don't want to print an article in 19pt type, you can use the file name `size9.clo` to implement your design for a layout that assumes the type size is 9pt. To implement a design for 14pt type you create the file `size14.clo`.

Adding the options to your extended document class is done by the following two lines of code:

```
\DeclareOption{9pt}%
              {\renewcommand\@ptsize{9}}
\DeclareOption{14pt}%
              {\renewcommand\@ptsize{4}}
```

All other options have to be passed on to the `article` class.

```
%----------- Option handling -----------%
\DeclareOption*{%
  \PassOptionsToClass{\CurrentOption}
                     {article}}
\ProcessOptions
%----------- Load other class ----------%
\LoadClass{article}[1994/06/01]
```

A real life example. Apart from adding options to an existing document class it is also possible to *disable* options that are allowed by the document class you are building upon. An example of this is the document class `ltxdoc`, used by the LaTeX3 project team for the documented source code of LaTeX. It contains the following lines of code:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{ltxdoc}
     [1994/05/27 v2.0n
      Standard LaTeX documentation class]
\DeclareOption{a5paper}%
    {\@latexerr{Option not supported}%
    {}}
\DeclareOption*{%
    \PassOptionsToClass  {\CurrentOption}%
                         {article}}
...
```

The interesting bit is the line that associates the option `a5paper` with an error message. When someone specifies the `a5paper` option to the class `ltxdoc` he will be warned that this document class does not support printing on A5 paper.

This document class allows customization by checking if a file `ltxdoc.cfg` exists. If a file with that name is found the user is told that the file is read in.

```
\InputIfFileExists{ltxdoc.cfg}
{\typeout{%
 *************************************^^J%
 * Local config file ltxdoc.cfg used^^J%
 *************************************}}
{}
```

Such a configuration file might contain the instruction to use A4 paper for printing:
`\PassOptionsToClass{a4paper}{article}`

When the configuration file is read, the options are processed and the `article` class is loaded.

```
\ProcessOptions
\LoadClass{article}
```

Then the package doc is required. This package is needed to print documented TeX source code, which the document class `ltxdoc` is made for.

```
\RequirePackage{doc}
```

The last line from this document class that is interesting is the following:

```
\AtBeginDocument{\MakeShortVerb{\|}}
```

This instructs LaTeX to store the command `\MakeShortVerb` together with its argument (`\|`) to be executed when `\begin{document}` is encountered.

Informing the user

Error handling. LaTeX2ε contains a set of commands that provide an interface for error handling. There are commands to signal an error (and prompt for corrective user input); commands to issue a warning about something and commands to just provide some information. In figure 1 you can see an example of the use of the command `\PackageWarningNoLine`. The result of executing the command is also shown.

Compatibility with LaTeX 2.09. Upwards compatibility is provided by the compatibility mode of LaTeX2ε. This mode was introduced to be able to run old LaTeX 2.09 documents through LaTeX2ε, yielding (almost) the same result. If this is what you need to achieve, than you may be pleased to know that the `\if@compatibility` switch can be used to test for compatibility mode. Using this switch, you can develop a full blown LaTeX2ε Package or Document Class out of a LaTeX 2.09 style file and yet still be able to print your old documents without changing them.

Possible Pitfalls while upgrading. Some mistakes that might be easily made and that can lead to unexpected results:

- You declare options in your package using `\DeclareOption` but forget to call `\ProcessOptions`. LaTeX will give an error, 'unprocessed options' unless sometimes other errors in the class file intervened and prevent the system detecting this mistake.
- The usage of either `\footheight`, `\@maxsep` or `\@dblmaxsep` outside of compatibility mode will lead to a complaint from TeX about an unknown command sequence.
- With LaTeX 2.09 the order in which options to a documentstyle were specified was *very* significant. A document would fail if the options were given in the wrong order. By default LaTeX2ε does *not* process the options in the order that they were specified in the document. It rather processes them in the order that they are declared in the class or package file. When the order of processing the options is relevant to your code you can use the command `\ProcessOptions*`. This will make LaTeX2ε evaluate the options in the order that they were specified in by the user.

  For the babel package for instance, the order of processing the options is significant. The last language specified in the option list will be the one the document starts off with.

`article`	successor of the `article` document style
`report`	successor of the `report` document style
`book`	successor of the book document style
`letter`	successor of the `letter` document style
`slides`	successor of the `slides` document style *and* SLITeX
`proc`	Successor of the proc style option
`ltxdoc`	to typeset the documented sources of LaTeX2ε
`ltxguide`	to typeset the LaTeX2ε guides
`ltnews`	to typeset the news letter that comes with each release of LaTeX

Table 2: Document classes that are part of LaTeX2ε

Document Classes and Packages in the LaTeX2ε distribution

Standard Document Classes. In table 2 an overview is given of the document classes that are available when you get the standard distribution of LaTeX2ε.

Most of these will be familiar to you, they are the successors of their LaTeX 2.09 counterparts. Basically these document classes behave like the old document styles. But there are a few changes:

- The options openbib and twocolumn are now internal options, the files `openbib.sty` and `twocolumn.sty` do not exist any more.
- A number of new options are implemented; supporting a range of paper sizes. Currently implemented are a4paper, a5paper, b5paper, letterpaper, legalpaper and executivepaper. These options are mutually exclusive.

  Another new option is the landscape option. It switches the dimensions set by one of the ..paper options. Note that this does not necessarily mean that when you combine a4paper and landscape the whole width of the paper will be used for the text. The algorithm which computes the

```
\PackageWarningNoLine{babel}
   {The language 'Dutch' doesn't have hyphenation patterns\MessageBreak
    I will use the patterns loaded for \string\language=0 instead.}
```

produces:

```
Package babel Warning: The language 'Dutch' doesn't have hyphenation patterns
(babel)                I will use the patterns loaded for \language=0 instead.
```

Figure 1: An example of the use of the command \PackageWarning

\textwidth from the \paperwidth has an upper bound in order to make lines of text not too long.

- The document class letter now also supports the option twoside. It does not support the option landscape.
- The document class slide can now be used with LaTeX, SLiTeX does not exist as a separate format any longer.

  Two column (using the option twocolumn) slides are not supported.

  While processing the document class slides LaTeX tries to load the optional file sfonts.cfg. This file can be used to customize the fonts used for making slides.
- The former *option* proc.sty has now been turned into a separate document class, which is implemented by building on article using the \LoadClass command. This class does not allow the options a5paper, b5paper and onecolumn.

A few new document classes have been added to the distribution of LaTeX. These are mainly meant to be used for documents produced by the LaTeX3 project team, but they can be used as an example of how to build a new class on top of an existing class. These classes are not yet finished and will probably change in the future.

- The document class ltxdoc is used in the documentation of all the LaTeX2ε source code. The document class is built upon the article class and also loads the doc package.

  It defines the command \DocInclude which works like the \include command from LaTeX, but sets things up for formatting documented source code.

  The formatting of the source code can be customized by creating the file ltxdoc.cfg. Such a file could for instance select your favorite paper size. This can be done by entering the following command in ltxdoc.cfg:

  ```
  \PassOptionsToClass{a4paper}{article}
  ```

  Selecting a5paper is not allowed; the source listings wouldn't fit.

ifthen	successor of the ifthen option
makeidx	successor of the makeidx option
showidx	successor of the showidx option
doc	successor of the doc option
shortvrb	implements \MakeShortVerb and \DeleteShortVerb
newlfont	successor of the newlfont option
oldlfont	successor of the oldlfont option
latexsym	makes the LaTeX symbol fonts available
exscale	implements scaling of the math extension font 'cmex'
fontenc	supports switching of *output* encoding
syntonly	successor of the syntonly option
tracefnt	successor of the tracefnt option

Table 3: Packages that are part of LaTeX2ε

- The document class ltxguide is used for the user guides that are included in the distribution.
- The document class ltnews is used for the short newsletter that accompanies the LaTeX distribution.

Packages. The packages that are contained in the LaTeX2ε distribution are listed in table 3. Most of these packages are described in *The LaTeX Companion.*

The package ifthen (which used to be the option ifthen) has been enhanced and now also defines \newboolean, \setboolean and \boolean{...} to provide a LaTeX interface to TeX's switches. Other new commands are \lengthtest and \ifodd.

The package shortvrb has only recently been introduced. It contains the definitions of the commands \MakeShortVerb and \DeleteShortVerb from the doc package. By providing this package those commands can also be used in other documents besides LaTeX source code documentation.

Related software bundles. Table 4 lists some related software bundles that are distributed separately.

The packages in these bundles come with documentation and each of them is also described in

amslatex	Advanced mathematical typesetting from the American Mathematical Society
babel	Supports typesetting in over twenty different languages
color	Provides support for colour
graphics	Inclusion of graphics files
mfnfss	Typesetting with bit-map (Metafont) fonts
psnfss	Typesetting with Type 1 (PostScript) fonts
tools	Miscellaneous packages written by the LaTeX3 project team

Table 4: Software bundles *not* part of LaTeX2ε

at least one of the books *The LaTeX Companion* (Goossens et al. 1994) and *LaTeX: A document preparation system* (Lamport 1994).

References

Goossens, Michel, Frank Mittelbach and Alexander Samarin. *The LaTeX Companion.* Addison-Wesley Publishing Company, 1994.

Lamport, Leslie. *LaTeX: A Document Preparation System.* Addison-Wesley Publishing Company, second edition, 1994.

The LaTeX3 Project team. *LaTeX2ε for Authors.* A document provided in the LaTeX2ε distribution in file usrguide.tex

The LaTeX3 Project team. *LaTeX2ε for class and package writers.* A document provided in the LaTeX2ε distribution in file clsguide.tex

The LaTeX3 Project team. *LaTeX2ε font selection.* A document provided in the LaTeX2ε distribution in file fntguide.tex

PostScript Fonts in LaTeX2ε

Alan Jeffrey
School of Cognitive and Computing Sciences, University of Sussex, Falmer, Brighton, BN1 9QH, UK
alanje@cogs.susx.ac.uk

Abstract

This paper describes the LaTeX2ε PostScript fonts package psnfss, and the Adobe Times math fonts package mathptm. This paper describes some of the design decisions made in psnfss, and gives an overview of how other fonts can be used in a similar fashion.

Introduction

This paper describes Sebastian Rahtz's psnfss package for using PostScript fonts in LaTeX. The psnfss software has been the standard way of using PostScript fonts in LaTeX for a number of years, and has recently been updated for LaTeX2ε and the fonts generated by the author's (1994) fontinst package.

The psnfss package aims to make using PostScript fonts as simple as possible. Once psnfss has been installed, users can select a LaTeX2ε package such as times and their document will be set in Adobe Times. The psnfss package comes with a standard set of TeX font metric (tfm) files, so psnfss documents should be portable between different TeX implementations.

One of the new features of psnfss is the mathptm package, which allows Adobe Times to be used as a math font as well as a text font. This only uses standard fonts (Adobe Times, Adobe Symbol and Computer Modern) so mathptm documents are portable, and the resulting PostScript files can be distributed without worrying about proprietary fonts.

Together with David Carlisle and Sebastian Rahtz's (1994) graphics and color packages, psnfss will help to free LaTeX from its popular image as only setting academic texts in Computer Modern with picture mode graphics.

Using psnfss

Once psnfss has been installed, it is very simple for users to use. They just select an appropriate package, for example:

```
\documentclass{article}
\usepackage{times}
```

Some document classes will be designed for use with PostScript fonts, and will automatically select PostScript fonts without the user selecting a package. For example, a publishing house producing *The Journal of Dull Results* may have their own jdullres document class, with options for pre-prints or final copy. An author would type:

```
\documentclass[preprint]{jdullres}
```

and would get a pre-print document set in Adobe Times, whereas the production staff would type:

```
\documentclass[crc]{jdullres}
```

to get camera-ready copy set in Autologic Times.

Documents written using psnfss can be printed with an appropriate dvi driver such as dvips, or OzTeX. Some previewers, such as xdvi, cannot preview PostScript fonts without turning the fonts into bitmaps (using up valuable disk space). But the PostScript can be previewed, using ghostview or pageview.

PostScript math fonts

One of the common complaints about using LaTeX with PostScript fonts is that the mathematics is still set in Computer Modern, for example as in Figure 2. This is unfortunate, since Computer Modern is a much lighter, wider and more cursive font than suits Adobe Times.

Until recently, the only thing that could be done about this was to use the MathTime fonts, available from Y&Y. These are fine fonts, and can produce excellent math setting. Unfortunately, they are proprietary software, and so cannot be distributed as freely as Computer Modern.

A less beautiful, but cheaper, solution is to use the mathptm package. This provides drop-in replacements for Computer Modern using virtual fonts built from Adobe Times, Symbol, Zapf Chancery, and Computer Modern. The results can be seen in Figure 3.

The mathptm fonts are distributed free, and the resulting PostScript documents can be made available for anonymous ftp without having to worry about unscrupulous readers stealing the fonts from the PostScript documents.

Roadmap

The rest of this paper describes some technical details about the implementation of psnfss, for the TeXnically minded.

Suppose $f \in \mathcal{S}_n$ and $g(x) = (-1)^{|\alpha|}x^\alpha f(x)$. Then $g \in \mathcal{S}_n$; now (c) implies that $\hat{g} = D_\alpha \hat{f}$ and $P \cdot D_\alpha \hat{f} = P \cdot \hat{g} = (P(D)g)\hat{}$, which is a bounded function, since $P(D)g \in L^1(R^n)$. This proves that $\hat{f} \in \mathcal{S}_n$. If $f_i \to f$ in $\mathcal{S}_n$, then $f_i \to f$ in $L^1(R^n)$. Therefore $\hat{f}_i(t) \to \hat{f}(t)$ for all $t \in R^n$. That $f \to \hat{f}$ is a *continuous* mapping of $\mathcal{S}_n$ into $\mathcal{S}_n$ follows now from the closed graph theorem. *Functional Analysis*, W. Rudin, McGraw–Hill, 1973.

Figure 1: Computer Modern text with matching math

Suppose $f \in \mathcal{S}_n$ and $g(x) = (-1)^{|\alpha|}x^\alpha f(x)$. Then $g \in \mathcal{S}_n$; now (c) implies that $\hat{g} = D_\alpha \hat{f}$ and $P \cdot D_\alpha \hat{f} = P \cdot \hat{g} = (P(D)g)\hat{}$, which is a bounded function, since $P(D)g \in L^1(R^n)$. This proves that $\hat{f} \in \mathcal{S}_n$. If $f_i \to f$ in $\mathcal{S}_n$, then $f_i \to f$ in $L^1(R^n)$. Therefore $\hat{f}_i(t) \to \hat{f}(t)$ for all $t \in R^n$. That $f \to \hat{f}$ is a *continuous* mapping of $\mathcal{S}_n$ into $\mathcal{S}_n$ follows now from the closed graph theorem. *Functional Analysis*, W. Rudin, McGraw–Hill, 1973.

Figure 2: Adobe Times text with Computer Modern math

Suppose $f \in \mathcal{S}_n$ and $g(x) = (-1)^{|\alpha|}x^\alpha f(x)$. Then $g \in \mathcal{S}_n$; now (c) implies that $\hat{g} = D_\alpha \hat{f}$ and $P \cdot D_\alpha \hat{f} = P \cdot \hat{g} = (P(D)g)\hat{}$, which is a bounded function, since $P(D)g \in L^1(R^n)$. This proves that $\hat{f} \in \mathcal{S}_n$. If $f_i \to f$ in $\mathcal{S}_n$, then $f_i \to f$ in $L^1(R^n)$. Therefore $\hat{f}_i(t) \to \hat{f}(t)$ for all $t \in R^n$. That $f \to \hat{f}$ is a *continuous* mapping of $\mathcal{S}_n$ into $\mathcal{S}_n$ follows now from the closed graph theorem. *Functional Analysis*, W. Rudin, McGraw–Hill, 1973.

Figure 3: Adobe Times text with matching math

The unpacked `psnfss` package comes as a number of files:

- Files ending with `sty` are LaTeX2ε packages. For example, `times.sty` contains the `times` package.
- Files ending with `fd` are LaTeX2ε font definition files. For example, `T1ptm.fd` contains the font definitions for Adobe Times. This tells LaTeX2ε that, for example, Adobe Times bold italic is called `ptmbiq`.
- Files ending with `tfm` are TeX font metric files. For example, `ptmbiq.tfm` contains the font information which TeX needs for Adobe Times bold italic.
- Files ending with `vf` are virtual fonts. For example, `ptmbiq.vf` contains the font information which some printers and previewers need for Adobe Times bold italic. This tells the printer that, for example, the character 'Ć' is made from an 'acute' and an 'C'.

The `sty` and `fd` files are used by LaTeX, the `tfm` files are used by TeX, and the `vf` files are used by printers and previewers.

Document portability

The `psnfss` package is intended to make documents as portable as possible. To achieve this, the `sty`, `fd` and `tfm` files should be the same at all sites. This means that documents using the `times` package will print identically on different sites.

Although the `psnfss` package distributes all of the files that are used by LaTeX and TeX, it does *not* include the files which are used by particular printer drivers, since these change from site to site. The `psnfss` package makes no requirements on the printer driver, except that it can print with PostScript fonts.

Virtual fonts

Although `psnfss` comes with virtual fonts, these are an optional part of the package. Some printer drivers (such as Y&Y's `dvipsone`) use PostScript font re-encoding rather than virtual fonts. The advantages of virtual fonts include:

- More than one font can be combined together. For example, many PostScript fonts contain 'ff' ligatures in the Expert fonts, but TeX requires ligatures to be in the same font.
- Composite letters such as 'Ă' and 'Ć' can be produced. TeX requires such letters to be in a font for the hyphenation algorithm, but most PostScript fonts do not contain them.
- Most printer drivers and previewers can use virtual fonts, so they are portable between systems.

The advantages of PostScript font re-encoding are:

- PostScript font re-encoding is faster, for example Textures previewing with virtual fonts can be twice as slow as with raw fonts.
- PostScript font re-encoding is a standard technology supported by many other applications.

- In order to access characters like 'Á', the printer driver has to use PostScript font re-encoding anyway, so virtual fonts need two levels of re-encoding rather than one.

Since there is a trade-off between virtual fonts and PostScript font re-encoding, the psnfss package makes no assumptions about using virtual fonts. The vf files are available for those who want to use them, but not all sites will want to use them.

Existing PostScript fonts

In the past, there have been problems with installing psnfss on systems which have already got PostScript fonts generated using Tom Rokicki's afm2tfm converter.

The main difference between the fontinst fonts and the afm2tfm fonts is that the fontinst fonts are designed to be drop-in replacements for Computer Modern, and can be used with no new macros. The afm2tfm fonts contain some new characters (such as '©') and some old characters (such as the accent on 'ă') in different slots, so need special macros. See Tables 2 and 3.

One of the most important features of LaTeX2ε is that different fonts can be used without new macros, which is one of the reasons for using the fontinst fonts rather than the afm2tfm fonts.

In the past, there were problems using psnfss on systems which had already got the afm2tfm fonts, because they used the same font names. This has now been changed, and fontinst uses the letters 7t to indicate a '7-bit TeX text' font, and 0 to indicate an 'Adobe Standard' font. For example the filenames for Adobe Times are:

encoding	*fontinst name*	*afm2tfm name*
Adobe Standard	ptmr0	rptmr
7-bit TeX text	ptmr7t	ptmr
8-bit TeX text	ptmrq	none

There should now be no clashes between the fonts generated by afm2tfm and those generated by fontinst.

Font naming

The psnfss fonts are named using Karl Berry's naming scheme. This tries to fit as many font names as possible into the eight letters provided by some operating systems. For most fonts here, the names are given by:

- One letter for the font foundry, e.g. p for Adobe.
- Two letters for the font family, e.g. tm for Times Roman, hv for Helvetica or cr for Courier.
- One letter for the weight, e.g. r for regular or b for bold.
- An optional letter for the shape, e.g. i for italic, o for oblique, or c for small caps. No letter means 'upright'.
- One or two letters for the encoding, e.g. q for 'Cork' encoding, 0 for 'Adobe Standard' encoding or 7t for Knuth's 'TeX text' encoding.

For example:

- ptmr7t is Adobe, Times Roman, regular weight, upright shape, TeX text encoding.
- phvbcq is Adobe, Helvetica, bold weight, small caps shape, Cork encoding.
- pcrro0 is Adobe, Courier, regular weight, oblique shape, Adobe Standard encoding. This is the font Adobe call 'Courier-Oblique'.

The naming scheme is described in more detail by Berry (1994). It is far from ideal, but does allow most fonts to be named in a consistent fashion. Most systems require a translation from the TeX font name to whatever the local font name is. For example, dvips can be told that pcrr0 is Adobe Courier with a line in the psfonts.map file:

```
pcrr0 Courier
```

OzTeX requires a line in the Default configuration file:

```
= pcrr0 Courier Courier Mac.enc
```

Textures can add new font names using the EdMetrics application.

The mathptm fonts

The mathptm fonts are an interesting exercise in using the fontinst package to generate quite complex virtual fonts.

The mathptm fonts are shown in Tables 4-7:

- zptmcmr is the 'operators' font, used for operators such as \log and \sin. It is built from Adobe Times roman (for most letters), Adobe Symbol (for upper-case Greek), and Computer Modern roman (for some symbols such as =).
- zptmcmrm is the 'letters' font, used for math italic. It is built from Adobe Times italic (for most letters), Computer Modern math italic (for most symbols) and Adobe Symbol (for Greek and \wp). The Adobe Times letters have their sidebearings changed for mathematics.
- zpzccmry is the 'symbols' font, used for math symbols. It is built from Computer Modern symbols (for geometric symbols such as \oplus), Adobe Symbol (for humanist symbols such as \nabla), Adobe Times (for text symbols such as \P) and Adobe Zapf Chancery (for calligraphic upper-case).
- zpsycmrv is the 'big symbols' font, used for large symbols such as '∑'. It is built from Computer Modern extensions (for most symbols) and Adobe Symbol (for \sum and \prod). The big operators such as \bigcup are set in 9pt rather than 10pt since these blend with the smaller operators from Adobe Symbol.

Γ Δ Θ Λ Ξ Π Σ Υ Φ Ψ Ω ff fi fl ffi ffl ı ȷ ` ´ ˇ ˘ ¯ ˚ ¸ ß æ œ ø Æ Œ Ø
- ! ” # $ % & ’ ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; ¡ = ¿ ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [“] ˆ ˙
‘ a b c d e f g h i j k l m n o p q r s t u v w x y z – — ˝ ˜ ¨

Table 1: Computer Modern Roman

■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ff fi fl ffi ffl ı ▮ ` ´ ˇ ˘ ¯ ˚ ¸ ß æ œ ø Æ Œ Ø
■ ! ” # $ % & ’ ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; ¡ = ¿ ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [“] ˆ ˙
‘ a b c d e f g h i j k l m n o p q r s t u v w x y z – — ˝ ˜ ¨
Ł
ł

Table 2: The fontinst Adobe Times roman

' ¡ ¿ ı ` ´ ˇ ˘ ¯ ˚ ¸ ß æ œ ø Æ Œ Ø
! " # $ % & ’ ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\ ] ˆ _
‘ a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ˜ ¨
^ ~ Á Â Ä À Å Ã Ç É Ê Ë È Í Î Ï Ì Ñ Ó Ô Ö Ò Õ Š Ú Û Ü Ù Ÿ Ž á â
ä à ¢ £ / ¥ ƒ § ¤ å “ « ‹ › fi fl ã – † ‡ · ç ¶ • ‚ „ ” » … ‰ é ê
ë è í î ï ì ñ ˙ ó ô ö ò õ ˝ ˛ š — ú û ü ù ÿ ž
ª Ł º ł

Table 3: The afm2tfm Adobe Times roman

Since the virtual fonts are built from commonly available fonts, rather than specifically designed fonts such as MathTime, there are some oddities. The parentheses are from Computer Modern rather than Adobe Times, since they have to blend with the extensible parentheses from zpsycmrv; this means the math parentheses are different from the text parentheses. Adobe Symbol has no large '$\sum$' and '$\prod$', so these have to be faked by magnifying the text characters, which does not look great. Some glyphs, such as \coprod are missing. The 'letters' Greek is upright rather than italic.

But despite these features, the fonts are acceptable, and make TeXnical typesetting with generally available scalable fonts possible for the first time.

Installing your own PostScript fonts

The psnfss fonts were all built using the fontinst font installation package. This package is written entirely in TeX, so it can be run on any TeX installation with enough memory. It is quite slow (about 20 minutes for a font family on a Macintosh with a 33MHz 68030) but this is acceptable since it is not run often.

Version 0.19 was described by the author (1993), but the user interface has changed considerably since then.

To install a new font using fontinst, you need the Adobe Font Metrics (afm) files for the fonts. These should be named with Karl Berry's naming scheme, for example ptmr0.afm for Adobe Times Roman. They should be in a directory which can be read by TeX.

Installing Latin fonts is quite simple. For example, to install the Adobe Times fonts, you run TeX on fontinst.sty and say:

```
\latinfamily{ptm}{}
```

If you have bought the Expert fonts, you can install Adobe Times Expert and Adobe Times Old Style by saying:

```
\latinfamily{ptmx}{}
\latinfamily{ptm9}{}
```

Once TeX has finished, it produces:

- pl files, which are text representations of tfm files. They can be converted to tfm files with pltotf (part of EdMetrics in Textures and OzTools in OzTeX).
- vpl files, which are text representations of vf files. They can be converted to vf files with vpltovf (part of EdMetrics in Textures and OzTools in OzTeX).
- fd files, which are used by LaTeX2ε.

The Adobe Times fonts can then be used in LaTeX by saying:

```
\renewcommand{\rmdefault}{ptm}
\rmfamily
```

Γ Δ Θ Λ Ξ Π Σ Υ Φ Ψ Ω ff fi fl ffi ffl ı ▮ ` ´ ˇ ˘ ¯ ˚ ¸ ß æ œ ø Æ Œ Ø
■ ! ” # $ % & ’ ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; ¡ = ¿ ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [“] ˆ ˙
‘ a b c d e f g h i j k l m n o p q r s t u v w x y z – — ˝ ˜ ¨
Ł
ł

Table 4: The mathptm 'operators' font

Γ Δ Θ Λ Ξ Π Σ Υ Φ Ψ Ω α β γ δ ε ζ η θ ι κ λ μ ν ξ π ρ σ τ υ φ χ
ψ ω ε ϑ ϖ ρ ς φ ↼ ↽ ⇀ ⇁ ‘ ’ ▷ ◁ 0 1 2 3 4 5 6 7 8 9 . , < / > ⋆
∂ *A B C D E F G H I J K L M N O P Q R S T U V W X Y Z* ♭ ♮ ♯ ⌣ ⌢
ℓ *a b c d e f g h i j k l m n o p q r s t u v w x y z ı* ■ ℘ ⃗ ⁀

Table 5: The mathptm 'letters' font

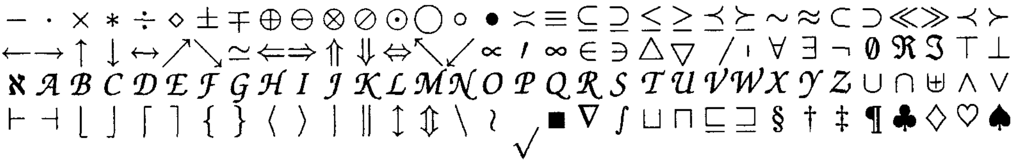


Table 6: The mathptm 'symbols' font

Table 7: The mathptm 'big symbols' font

If you want to write your own package similar to times, you just create a sty file containing these lines.

For example, an Adobe Times Old Style package would contain:

```
\ProvidesPackage{timesold}
   [1995/04/01 Times old style digits]
\renewcommand{\rmdefault}{ptm9}
\rmfamily
```

It is also possible to create customized fonts (such as the mathptm math fonts) using fontinst, but this is somewhat trickier, and is described in the fontinst documentation.

If you use fontinst to install some fonts, and you would like to distribute the results, please mail them to me, and I'll include them in the fontinst distribution.

Ongoing work

There are a number of areas of ongoing work with fonts and TeX.

There is a TUG working group on TeX directory structures, which will recommend how fonts should be installed on any TeX system. This will make it easier to distribute TeX software, because there will be a standard directory structure to refer to. The CTAN fonts/metrics directory will reflect the TDS structure.

The fonts produced by fontinst do not include 'eth' or 'thorn' because these characters are not available in Adobe Standard encoding. We are working on developing a suitable replacement for Adobe Standard encoding which will allow access to 'eth', 'thorn' and the other missing characters.

There is a TUG/LaTeX working group on math font encodings, which will develop symbol encodings which will be supported by fontinst (this working group has been rather slowed down by the production of LaTeX2ε).

Bibliography

Berry, Karl. *Filenames for fonts.* Available as fontname.texi from CTAN, 1994.

Carlisle, David and Rahtz, Sebastian. *The color and graphics LaTeX2ε packages.* Available from CTAN, 1994.

Goossens, Michel, Frank Mittelbach and Alexander Samarin. *The LaTeX Companion.* Addison-Wesley, 1994.

Jeffrey, Alan. "A PostScript font installation package written in TeX." *TUGboat* **14** (3), pages 285–292, 1993.

Jeffrey, Alan. *The fontinst package.* Available from CTAN, 1994.

Rahtz, Sebastian. *Notes on setup of PostScript fonts for LaTeX2ε*. Part of the psnfss package, available from CTAN, 1994.

BibTeX 1.0

Oren Patashnik
Center for Communications Research, 4320 Westerra Ct., San Diego, CA 92121, U.S.A.
opbibtex@cs.stanford.edu

Abstract

This paper discusses the history of BibTeX, its current status, and the future goals and plans for it. BibTeX 1.0 will be the frozen version of BibTeX, just as TeX 3 (but not as TeX 2!) is the frozen version of TeX. Among the goals for BibTeX 1.0 are: easier creation of nonstandard bibliography styles; easier sharing of database files; and better support for non-English users. Among the new features will be: a program that lets users create their own bibliography styles; support for 8-bit input; support for multiple bibliographies within a single document; and the capability to indicate in a bibliography entry where in the text the entry was cited.

Introduction

BibTeX is the bibliography program designed originally to accompany Leslie Lamport's LaTeX (it now works with other versions of TeX, too). The first publicly released version of BibTeX, 0.98, came out in 1985. The second main release, version 0.99, came out in 1988. The long overdue final version, 1.0, is still under preparation. This paper discusses BibTeX and the plans for version 1.0. The remaining sections: explain BibTeX for those who haven't used it; give a brief history of BibTeX; describe the general goals for BibTeX 1.0; describe some specific new features for achieving those goals; and make some requests of the TeX community that will facilitate the release of BibTeX 1.0.

Using BibTeX

To use BibTeX you put into your (LA)TeX source file citations like

```
... in the \TeX{}book~\cite{knuth:tex} ...
```

along with two other commands:

```
\bibliography{mybib}
\bibliographystyle{plain}
```

(LA)TeX will typeset the citation as

... in the TeXbook [23] ...

or

... in the TeXbook[23] ...

or

... in the TeXbook (Knuth, 1984) ...

depending on which citation style you specify; (LA)TeX's default citation style is a number in brackets. (In some citation styles — for example in the author-date style that produces 'Knuth, 1984' — BibTeX helps (LA)TeX produce the citation.)

The `\bibliography` command does two things: it tells (LA)TeX to put the bibliography at that spot in your document, and it tells BibTeX which file(s) to use for the bibliographic database, here just the single file `mybib.bib`. The `\bibliographystyle` command tells BibTeX which bibliography style to use, here the standard style `plain`.

The `\cite` command's argument `knuth:tex`, called a cite-key, must have a matching database-key for some entry in the bibliographic database. That entry (in `mybib.bib`) will look like

```
@BOOK{knuth:tex,
   author = "Donald E. Knuth",
   title = "The {{\TeX}}book",
   publisher = "Addison-Wesley",
   year = 1984,
}
```

The `@BOOK` tells BibTeX that this is a book entry. The `knuth:tex` is the database key. And the rest of the entry comprises four ⟨field⟩ = ⟨field-value⟩ pairs appropriate for this entry type. (LA)TeX and BibTeX might (depending on the bibliography style) typeset this as

23. Donald E. Knuth. *The TeXbook.* Addison-Wesley, 1984.

(LA)TeX determines a few things about how the reference list is formatted — things like whether the label 23 is followed by a period or is enclosed in brackets, and the vertical spacing between entries. But the BibTeX bibliography style determines everything else — things like how the entries are sorted, whether to use a slanted or italic type style for a book's title, whether the author's surname comes first or last, and whether to use the author's full given name or just initials.

To actually produce the typeset document, you run (LA)TeX, BibTeX, (LA)TeX, (LA)TeX. The first (LA)TeX run writes, to an `.aux` file, information for BibTeX:

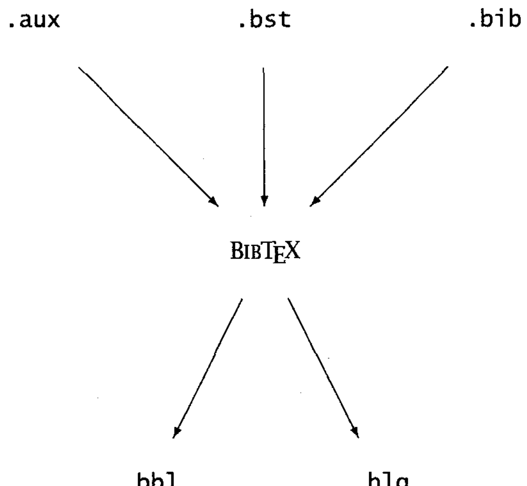


Figure 1: BibTeX's input and output files.

which bibliography style to use, which database file(s) to use, and which database entries to include. The BibTeX run reads all that information from the .aux file, reads the specified database (.bib) files, formats the reference list according to the instructions in bibliography style (.bst) file, and writes its output onto a .bbl file. The next (LA)TeX run reads the .bbl file and incorporates the reference list into the document. The final (LA)TeX run gets the forward references due to the \cite commands correct. Figure 1 shows the files that BibTeX uses (the file with extension .blg is BibTeX's log file).

That's a quick overview. The following sources provide details about using BibTeX. Leslie Lamport's LaTeX book (1994) explains how to use BibTeX with LaTeX. The file btxmac.tex (1992) documents its use with plain TeX, with or without Karl Berry's eplain.tex package (for which the btxmac macros were originally written). The "BibTeXing" document (1988a), which is distributed along with BibTeX itself, contains further hints for BibTeX users. The "Designing BibTeX Styles" document (1988b), also distributed with BibTeX, explains the postfix stack-based language used to write BibTeX bibliography styles. Michel Goossens, Frank Mittelbach, and Alexander Samarin's *LaTeX Companion* (1994) summarizes much of the information contained in the sources above, and it describes some of the tools available for helping with BibTeX bibliographies. Norman Walsh's *Making TeX Work* (1994) also describes such tools.

History

Brian Reid, in the late 1970's at Carnegie-Mellon University, wrote a document production system called Scribe (Unilogic 1984). One of its basic tenets was that, to the extent possible with a computer program, writers should be allowed to concentrate on content rather than on formatting details. Or, as Reid so amusingly put it[1]:

Not everyone should be a typesetter.

(I think of LaTeX as a fairly successful Scribification of TeX — LaTeX is almost as easy to use as Scribe yet almost as powerful as TeX.)

In any case, Scribe had become popular in certain academic circles, and Leslie Lamport decided that, to make it easy for Scribe users to convert to LaTeX, he would adopt Scribe's bibliography scheme in LaTeX. But TeX macros alone were insufficient in practice to do all the things, like alphabetizing, a bibliography processor needs to do; he decided instead to have a separate bibliography program. That program would manipulate the bibliographic information in Scribe-like database files according to the instructions programmed in a special-purpose style language. The postfix stack-based language he had in mind was to be powerful enough to program many different bibliography styles.

My own work on BibTeX started in February 1983 as a "three-week project" (not unlike the "three-hour tour" of the 1960's American television series *Gilligan's Island*, in which an afternoon's harbor cruise became a shipwreck adventure lasting years). Over the course of the next year and a half I implemented Lamport's basic design, with a few enhancements.

The first working version of BibTeX (0.41) trudged forth in the summer of 1984. Lamport wrote, and Howard Trickey modified, a bibliography style based on Mary-Claire van Leunen's suggestions in her *Handbook for Scholars* (1979). Trickey's modified version was to become btxbst.doc, which is the template from which BibTeX's four standard styles (plain, abbrv, alpha, and unsrt) are generated.

The first public release of BibTeX, in March 1985, was version 0.98, for LaTeX version 2.08. Several upgrades, including one for LaTeX 2.09, followed later that year. Version 0.99, which added many new features, was released in January 1988; two minor upgrades followed the next month, but BibTeX itself has remained unchanged since then. The standard styles have been unchanged since March 1988.

In 1990 Karl Berry wrote some macros, for use in his eplain.tex package, that made BibTeX usable with plain (and other versions of) TeX. He and I modified the macros and released them as btxmac.tex in August 1990, usable with or without the eplain package. Several upgrades followed, the most recent in March 1992.

The current versions are: 0.99c for BibTeX itself; 0.99b for btxbst.doc (the standard styles' template file — but version 0.99a for each of the four standard styles); and 0.99j for btxmac.tex.

[1] — while barefoot, with a pregnant pause, to a Bell Labs Computer Science Colloquium audience that included some troff true believers

Goals

BIBTEX has been very stable for several years now. Software stability is nice; it helps others build tools that augment the software. Indeed many tools have grown up around BIBTEX. But the popularity of (LA)TEX has taken BIBTEX into unanticipated places, necessitating some changes. I have five main, general goals for BIBTEX 1.0:

1. Easier nonstandard styles: The most frequent requests I see are for new bibliography styles. Creating a new bibliography style generally entails programming in the `.bst` language, which is difficult. For BIBTEX 1.0, ordinary users, too, must be able to create new bibliography styles reasonably easily.
2. More international: BIBTEX has spread to the non-English-speaking world. BIBTEX 1.0 must address associated issues.
3. Enhanced sharing capabilities: There now exist huge `.bib`-format bibliographic databases, some available to users world wide. BIBTEX 1.0 needs to make the sharing of those databases easier.
4. Better documentation: BIBTEX 1.0 documentation needs to be more extensive and easier to find.
5. FroZEN: To enhance stability of the program (and its author — that is, for both practical and personal reasons) BIBTEX needs to be frozen. As with TEX 3.0, BIBTEX 1.0 will be upgraded for bug fixes only.

Some of the features planned for implementing those goals appear in the next section.

New Features

Over the last six years I have accumulated a list of new features and probable changes for BIBTEX 1.0. The list below is certainly not exhaustive, but it contains the most important items. Each one listed has a high probability of existing in BIBTEX 1.0.

- A Bibsty program: There will be a new scheme for generating bibliography style (`.bst`) files. A program called Bibsty will create a customized `.bst` file from (i) a BIBTEX template (`.btp`) file — which will be similar in spirit to (but contain lots more options than) the current file `btxbst.doc` — together with (ii) options that the user specifies. BIBTEX 1.0's standard template file, to be called `btxstd.btp`, will, among other options, have an easily changed symbolic name for each string that a bibliography style outputs directly (such as 'editor' or 'volume'). This will make it much easier to, for example, convert bibliography styles from one language to another. Figure 2 shows how the new Bibsty program will fit into the scheme.

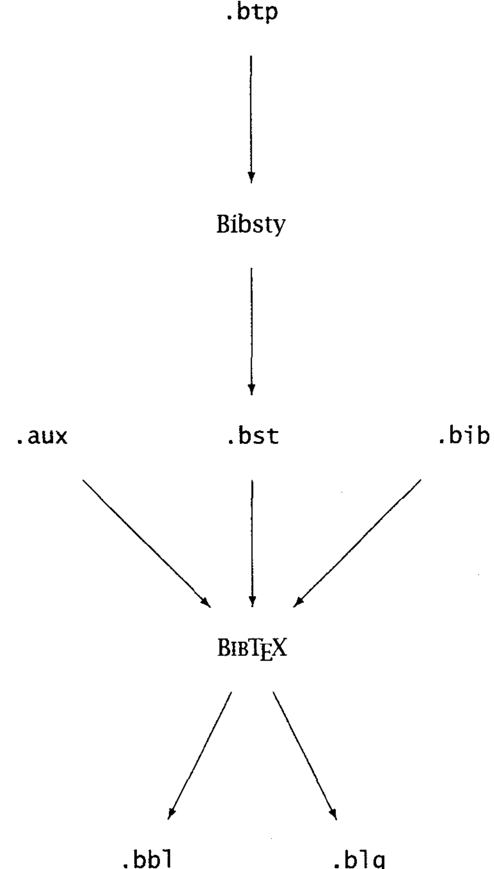


Figure 2: BIBTEX 1.0 input and output files.

- Reference-list back pointers: BIBTEX 1.0 will provide the capability to indicate in a reference-list entry where in the text that entry was cited. This is a very useful feature that I suspect will become widespread now that our new typesetting technology makes it painless.
- Eight-bit input: BIBTEX 1.0 will support this by adhering as closely as possible to the character-set conventions of TEX 3.
- Support for multiple bibliographies within in a single document: Many large documents contain several bibliographies — a book might have one bibliography per chapter, or a conference proceedings might have one per paper. Several solutions have arisen for handling such situations, but BIBTEX 1.0 will support multiple bibliographies directly, hence those solutions won't be necessary.
- An `@ALIAS` command: Suppose you have a database file that uses a different database-key from the cite-key you prefer. For example the database file might have the database-key `knuth:tex` for an entry for which you've used `texbook` as a cite-key. With BIBTEX 1.0 you will

be able to keep the cite-key and database-key as is, as long as you put a command like

```
@ALIAS{texbook = knuth:tex}
```

in your database.

- A `@MODIFY` command: With BibTeX 1.0 you will be able to make changes to an entry in a public database file without having to repeat in your own personal database file all the information in that entry. For example, to create a second-edition update for an entry whose first edition is in a public database, you can put something like

  ```
  @MODIFY{latexbook,
     edition = "second",
     year = 1994,
  }
  ```

  in your own database file, as long as the database-key of the `@MODIFY` command matches the database-key from the public database.
- Distinguishing among identical database-keys: If you are using two different database files that happen to use the same database-key for different entries, you will be able to specify which entry you want by using a citation of the form

  ```
  \cite{filename::database-key}
  ```
- A `.bib`-file extraction mode: BibTeX 1.0 will have a mode that will let you extract just the `.bib`-file information you need into a much smaller database file. For example if you are submitting a paper to a journal that wants a `.bib`-file in addition to a `.tex`-file, but the bibliographic database you are using for the paper is huge, you can use the extraction mode to put just the entries you need for the paper into a separate `.bib` file that you can then send to the journal.
- A `\bibtexoptions` (LA)TeX command: This command will improve communication between (LA)TeX and BibTeX 1.0. For example, currently BibTeX has a compile-time constant `min_crossrefs`; a `\bibtexoptions` command might let a user set this from within the (LA)TeX file.
- Extensions to the (LA)TeX `\cite` command: Many citation styles aren't handled very gracefully by (LA)TeX's current `\cite` command. BibTeX 1.0 and (LA)TeX will support more flexible `\cite` commands.
- Standard-style changes: The standard styles for BibTeX 1.0 will have a few minor changes, such as the addition of `day`, `isbn`, and `issn` fields, and a new `@PERIODICAL` entry type.
- `.bst`-language changes: There will be a few minor changes to the `.bst` language.
- `btxmac.tex` update: These macros will be updated so that the user interfaces to BibTeX 1.0 from LaTeX and plain TeX are identical.
- Documentation: The "BibTeXing" (1988a) and "Designing BibTeX Styles" (1988b) documents currently distributed with BibTeX are unfortunately not as widely known as they should be. To improve the situation for BibTeX 1.0, the documentation will be in a book. It will be much more thorough than the current documentation. For example it will give a `.bib`-file grammar, so that those who are writing tools to manipulate the database files can make their software more robust.

Requests

To facilitate the release of BibTeX 1.0, I have some requests of the TeX community.

- Please don't ask me routine BibTeX questions via e-mail. Instead, post them to the newsgroup `comp.text.tex`; in fact your request will get a wider distribution if you send it to the mailing list `INFO-TeX@SHSU.edu`, as it will also be posted automatically to the `comp.text.tex` newsgroup. State clearly in your message exactly what it is you want to know. Ask to have responses sent to you directly (assuming you aren't on that mailing list and don't read that newsgroup). Usually you get useful responses.
- Until BibTeX 1.0 is finished, I will skim the `comp.text.tex` newsgroup for BibTeX-related postings, so it suffices to post there anything you think I should see.
- Please send directly to me any suggestions for BibTeX 1.0 that are probably not of interest to the rest of the TeX community, such as:
  - Primitives that you think belong in the `.bst` language based on your experience programming it.
  - Things you've had to put in BibTeX's `WEB` change file for your system that you think belong in `bibtex.web` itself.
  - Non-English language pitfalls that you think BibTeX 1.0 should avoid.

To conclude the paper: I can't say for sure when BibTeX 1.0 will actually appear; a beta-test version might exist by the end of 1995. But as soon as it's available it will be announced on `comp.text.tex`.

Acknowledgments

I thank Nelson Beebe for his BibTeX suggestions and for reading an earlier draft of this paper.

References

Berry, Karl and Oren Patashnik. "`btxmac.tex`." Macros to make BibTeX work with plain TeX; current version 0.99j, 14 March 1992.

Goossens, Michel, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion.* Addison-Wesley, 1994.

Lamport, Leslie. *LaTeX: A Document Preparation System.* Addison-Wesley, second edition, 1994.

Patashnik, Oren. "BibTeXing." General documentation for BibTeX users, contained in the file `btxdoc.tex`, 8 February 1988(a).

Patashnik, Oren. "Designing BibTeX Styles." Documentation for BibTeX style designers, contained in the file `btxhak.tex`, 8 February 1988(b).

Unilogic, Ltd., Pittsburgh. *Scribe Document Production System User Manual*, fourth edition, 1984. Chapter 12 and appendices E8 through E10 deal with bibliographies.

van Leunen, Mary-Claire. *A Handbook for Scholars.* Knopf, 1979.

Walsh, Norman. *Making TeX Work.* O'Reilly & Associates, 103 Morris Street, Suite A, Sebastopol, CA 95472, 1994.

A Typesetter's Toolkit

Pierre A. MacKay
Department of Classics DH-10, Department of Near Eastern Languages and Civilization (DH-20)
University of Washington, Seattle, WA 98195 U.S.A.
Internet: mackay@cs.washington.edu

Abstract

Over the past ten years, a variety of publications in the humanities has required the development of special capabilities, particularly in fonts for non-English text. A number of these are of general interest, since they involve strategies for remapping fonts and for generating composite glyphs from an existing repertory of simple characters. Techniques for selective compilation and remapping of METAFONTs are described here, together with a program for generating PostScript-style Adobe Font Metrics from Computer Modern Fonts.

Introduction

History. For the past ten years I have served as the technical department of a typesetting effort which produces scholarly publications for the fields of Classics and Near Eastern Languages, with occasional forays into the related social sciences. Right from the beginning these texts have presented the challenge of special characters, both simplex and composite. We have used several different romanizations for Arabic, Persian and Turkish, and at least three conventions for fully accented Ancient Greek: book text, epigraphical and numismatic. The presses we deal with will sometimes accept the default Computer Modern, which looks very handsome indeed when set in a well designed page, but they often have a house style or a series style that requires some font not available in METAFONT coding. Special characters are still needed even in those styles, but fortunately we have not yet had to create any completely new glyphs for the PostScript Type 1 fonts we rely on, only to modify existing glyphs.

To appreciate the constraints we are under, you have to remember that most of our work is in the humanities, and that the humanities are the Third World in the scholarly family of nations. It is an odd correlation, but apparently a sound one, that the most productive of humanities scholars, is probably going to be the one with the least access to useful technology. If you are ever curious about items of industrial archaeology such as the original KayPro microcomputer, search among the humanists of your local university, and you may find a dozen or so still in use. Humanist manuscripts are often the result of fifteen or twenty years' effort, and usually start out in a technology that was out of date in the first place. In the past year we have had to work with one text made up of extensive quotations from Greek inscriptions, composed in a version of RUNOFF dating from about 1965 and modified by an unknown hand to produce Greek on an unknown printer. The final version of the manuscript was submitted to the editorial board in 1991. I have no idea when the unique adaptation for epigraphical Greek was first written. At least I know what RUNOFF is. A subsequent text arrived in 1992 composed, with passages in both Greek and Latin, in a system named SAMNA, about which we could find out nothing at all at the time. (The reviewer of this paper kindly informed me that it is a mid-range PC wordprocessor, but in our case I got the impression that we were dealing with a mainframe program.)

The editor of a humanities series is not at liberty to set the kind of conditions that were set for submissions to this session of the TEX Users Group. You do not tell a scholar who has fought through twenty years to the completion of a monograph that he must now retype the entire thing in a different convention. Each monograph is a unique problem, with unique demands for special fonts, formatting and coding. So the first items in our toolkit are general tools for remapping and character selection, simple but powerful tools for any work in font development. These tools preserve the integrity of the basic character designs in a font, eliminate the proliferation of dialect versions of public source files, and make the effort of development much simpler and faster.

We have had to explore both METAFONT and PostScript rather intensely for some of this work, and to do so under considerable pressure. (It is

surprising how often a really complex typesetting task arrives with the request that it be returned as final camera-ready copy some time the previous week.) This paper describes some of the tools we have developed in and around the basic TEX and METAFONT utilities. Not all the tools are written in TEX and METAFONT coding. It is undoubtedly possible to write a serviceable awk interpreter in TEX macros, but when the pressure is on, the fact that awk and sed are already to hand in any Unix system is irresistible. Since contributors to the software distributions of the Free Software Foundation are making many of these tools available even outside the Unix realm, the mention of them is no longer to be condemned as purely Unix Cultural Imperialism.

METAFONT Tools

Selection from existing character files. One of our earliest METAFONT tools came out of the need to experiment expeditiously with numerous mode_defs for laser-printers. To do this conscientiously, you have to manipulate at least two of the usual mode_def parameters, blacker and fillin, and you have to do it at several resolutions, so as to be sure that you have reached a compromise that is reasonable over the normal range of text sizes from seven to fourteen point, and not utterly impossible above and below those sizes. Since blacker and fillin interact with one another in unexpected ways, this evaluation can become impossibly burdensome if an entire font must be generated for each attempt. I chose therefore to borrow from Knuth's test files, and produced modetest.mf, which is a straight steal, modified very slightly so that the characteristic use_it/lose_it macros will not be confused with Knuth's version. (An extract from modetest.mf is given in the Appendix.)

Because the if...fi construction is about the first test applied to either TEX or METAFONT input, discarding unwanted characters in this way wastes very little time and races through to produce a selective minifont in only a few seconds on a medium-fast workstation. The process of evaluating the results can be further accelerated with the aid of other tools which I shall return to in another paper. Here, I wish to explore some of the other uses of the use_it/lose_it macros. When I originally wrote the code for a slab-serifed uppercase 'I' to make a Computer Modern version of the font used in Leslie Lamport's SliTEX, I took a copy of the entire romanu.mf file, called it sromanu.mf and introduced the modified letter into that. This was not merely a wasteful approach, it was intrinsically a bad one. Over the past ten years, various changes have been made to the Computer Modern character files, and I am virtually certain that most of the copies of sromanu.mf across the world have not kept up with those changes. In this particular instance it could be argued that it doesn't much matter, but there are dozens of other places where literal copies of Knuth's Computer Modern have been used unnecessarily, and where the probability is very strong that they have been let go stale while small improvements were made to the official release. The present UnixTEX release of sromanu.mf is quite small, and runs no risk of missing out on changes in the basic romanu.mf.[1] (For an extract from sromanu.mf, see the Appendix.)

Silvio Levy's Greek font provides another example very much to the point. In order to achieve the maximum compatibility with Computer Modern for bilingual texts, the uppercase letters of the Greek alphabet in this font were copied from romanu.mf and greeku.mf, given different code positions, and collected into the file upper.mf in the original release. So far as I know, no change has been made in upper.mf in many years, although the Computer Modern sources have seen several small changes. A more efficient approach than making a copy of the character files is gribyupr.mf. (An extract is given in the Appendix.) This generates the entire uppercase Greek alphabet, including a lunate uppercase sigma and a digamma, without any new character design code being provided at all. The Computer Modern sources are treated as read-only files, as they ought to be, and remapped into a convenient order in the Greek font. The mapping in evidence here is actually for a font associated with modified *Thesaurus Linguae Graecae* Beta-code, not for Silvio Levy's original fonts, and extensive remapping and even character substitution is done in other parts of the font as well. But Silvio Levy's files are never altered. They are treated as a read-only resource throughout.

[1] I have attempted to adapt a similar approach to speed up the the generation of invisible fonts but the existing pattern of if...fi groupings around such characters as lowercase g is pretty intractable. The present approach, for which I am afraid I am responsible in the Unix environment at least, is to let METAFONT go through all the work of creating the picture and then erase it, a practice that is dismally slow on a Macintosh or a PC.

Remapping an existing character set in METAFONT. The remapping used in this example uses the convention suggested by the Computer Modern caps and small-caps fonts in the file csc.mf, where it is used to map the small caps into the lowercase positions in the font. In instances where the first half of a 256-character font must be remapped, `code_offset` is the obvious way to do it. Greek in a Latin-letter input convention is one of the most obvious examples, but the trick can be used to create a consistent Turkish input convention, with the dotless 'ı' associated with the principal, unmodified alphabet or to respond to a Latin-letter input convention for Cyrillic. Since it involves no redesign, it is extremely economical, and so long as the input file does not make unexpected use of the characters with special `catcodes` in TEX, this approach avoids any major alterations in the way TEX handles normal text. In the upper end of such a font, however, some rather more interesting things can be done. Here we are free of the constraint against wholesale changing of `catcodes`, and by making the entire range of characters from 128 to 255 into `active` characters, we can set up a broadly flexible remapping system, which can be tailored to each application.

This approach to remapping is best illustrated again with Silvio Levy's font. Because this font was made up before the TEX3 enhancement in ligature coding, Levy provided an entire repertory of letters following the medial form of sigma, and kept the final form as the default. The more adaptable style of ligature coding allows us to reverse this arrangement, by using the following ligature table:

```
ligtable "s": "+" =: sampi, "i" kern i#,
  boundarychar =: sigmafinal,
  "." =:| sigmafinal, "," =:| sigmafinal,
  "?" =:| sigmafinal, ":" =:| sigmafinal,
  ";" =:| sigmafinal, "(" =:| sigmafinal,
  ")" =:| sigmafinal, "*" =:| sigmafinal,
  "!" =:| sigmafinal, "%" =:| sigmafinal,
  "<" =:| sigmafinal, ">" =:| sigmafinal,
  "[" =:| sigmafinal, "]" =:| sigmafinal,
  "{" =:| sigmafinal, "}" =:| sigmafinal,
  "|" |=: null_space;
```

The last element in this table is necessary because the ligature program is otherwise so effective that it makes it nearly impossible to terminate a word with a medial sigma, though that is often required by classical texts.

With all the code positions freed up by the elimination of sigma+letter pairs, it became possible to supply the combinations of vowel with barytone (grave) accent that were missing from the original coding, but these had to be placed in fairly arbitrary places in the font. I needed a way of separating the accidents of TEX input coding from the accidents of METAFONT character coding, but I did not want two separate coding files which might get out of step with each other. One mapping file had to serve for both TEX and METAFONT.

It is not entirely easy to make a file readable in both TEX and METAFONT, but it can be done as shown in the file `ibygrk.map`. (Listing in the appendix.) One significant advantage of this approach is that it fosters the assignment of symbolic names for all characters. The consistent use of symbolic names for characters is one of the most significant virtues of the PostScript system of encoding and, dare I say it, the overuse of numeric indices such as `ASCII"A"`, and `oct"000"` is a noticeable weakness in even Knuth's METAFONT programming. It may, of course, have been impossible to allocate large enough regions of symbol storage in early versions of METAFONT.

In the comments lines of `ibygrk.mf`, mention is made of the resolution of composite characters into digraphs and trigraphs before they are given as text character input to TEX. This is perhaps on the margins of a discussion of tools, but it has been so grossly misunderstood when I have brought it up in the various electronic newsletters devoted to TEX that it needs airing here. In an environment where complex multilingual text comes in from a wide range of sources, using every well-known and several extremely obscure word-processing systems, usually after local customization, it is hardly ever possible to arrange for final copy editing with the original system and coding. If the editor is using a Macintosh as an ASCII terminal over a telephone line, it is not very helpful to know that on the wierd, unique input system used by the author you get an omega with rough breathing, perispomenon accent and an iota subscript by pressing the F1 key. There has to be a fall-back coding available that does not depend on customized special-purpose hardware and software. In our case the character just described would be produced by the sequence `w(=|` which will get through all but a few really incompetent mail interfaces as well as through a normal serial communications line.

In a special file of character sequences, the definition

```
\def\w_asprperiisub{w(=|}
```

ensures that whatever arbitrary eight-bit character happens to be equated with

```
omega_spiritusasper_perispomenon_iotasub
```

it will be decomposed into the sequence w(=¦ before it actually reaches horizontal mode processing, and the tfm ligature program will take care of reconstructing it again. For the final copy-editor, this is very important. It is sometimes impossible to introduce an arbitrary octal character into a word-processor text, and even when it is possible, constant reference to a book of code pages is dreary and time-wasting. One might insert a \char'nnn sequence, but that would require having the font layout accessible at all times, which is not much of an improvement over looking at code pages. Digraphs, trigraphs and tetragraphs provide the same economy and efficiency that an alphabet provides by comparison with a syllabary, and a syllabary by comparison with an ideograph.

The objection that is always raised to digraphs etc. is that they may do strange things to hyphenation. This belief is based on a serious misunderstanding of the way in which hyphenation patterns are coded. It takes only the minimum amount of additional coding to ensure that no character in the diacritical set may ever be split off from the preceding character, and if that leaves some undesirable hyphenations possible with a reasonable setting of \lefthyphenmin and \righthyphenmin, they can be prevented by creating hyphenation patterns with the ▪ boundary char. (I use a bullet here in place of the period which is actually coded. The period is too hard to notice.) If I want to ensure that the position before w(=¦ at word end is never considered for hyphenation, I have only to include the pattern 8w(=¦▪ in the hyphenation file, and it will be disallowed absolutely. In actual fact, we have never had to set enough continuous Greek to make automatic hyphenation necessary, so I have never created such a table.[2] I have used a hyphenation table built up from digraphs and trigraphs for Ottoman Turkish in an extended diacritical convention based on modern Turkish, and we have no trouble whatsoever with unexpected wordbreaks.

Remapping through Virtual Fonts

In place of the direct manipulations of METAFONT source suggested above, the more general approach through Virtual Fonts may be applied even to METAFONTs such as Computer Modern. Until recently, I was unable to make the fonts for Ottoman Turkish in romanized transcription available to my colleagues with PCs except as precompiled PK files. The METAFONT sources require a very large compilation that is usually not available on PC architectures. But since the release of virtual font dvi interpreters for the smaller machines it is now possible to make composite accented characters available with a very small addition to the actual font library on a small machine. The approach I use is based on Tom Rokicki's afm2tfm. A great deal of it can also be done with the fontinst package, but that was just being started when I was working on the following tools. Moreover, some of these tools are complementary to fontinst, rather than alternatives.

Rokicki's afm2tfm takes Adobe Font Metrics as its inputs, so the first requirement is to create afm files for Computer Modern.[3] According to the *Red Book* the Adobe FontMatrix is based on a 1000 unit coordinate system.[4] In METAFONT terms this translates to 1000 dots per em. We are not talking about dots per inch here, but dots per em. Knuth's designs conform to the old traditional definition of the em, or more precisely the em-square, as the square on the body of the type. The body of the type is measured from top to bottom, since that ought to be the reference dimension for any stated point size, and it includes shoulders at both top and bottom, so that with most faces, the distance from the top of ascenders to the bottom of descenders is

---

[2] I have looked with great interest at Yannis Haralambous's rationale for hyphenation in classical Greek, but I should like to compare the results with the only large computer-readable resource of Greek text in existence — the *Thesaurus Linguae Graecae.* The creation of an acceptable hyphenation scheme for a "dead" language is more problematic than it appears on the surface. It appears that the ancient preference was for word-break between a vowel and any following consonant cluster, no matter how irrational and unpronounceable that cluster might seem. But most of our early evidence is from inscriptions, and inscriptions may not be a very good guide. I strongly suspect that a systematic review of the European classical tradition, out of which almost all standard editions have been issued, would show national preferences. That is to say, an English edition, a French edition and a German edition of Plutarch might well be recognizably different in their hyphenation choices.

[3] Except for very special effects, direct editing to convert a PL file to a VPL file is not a reasonable alternative. VPL coding is a valuable adjunct to the creation of composites and special effects through virtual font mapping, but the format is diffuse and difficult to work through.

[4] PostScript Language Reference Manual, p. 226.

less than the stated point size.[5] We want, therefore, to take our measurements from a font generated so that the distance from the top of the top shoulder to the bottom of the bottom shoulder is exactly 1000 units. For a ten point font this translates to 7227dpi. For a five-point font it translates to a horrendous 14454dpi. These resolutions are easily achieved with a mode_def which I use as smode input.

```
numeric xppi;
xppi := in# / designsize ;
mode_param (pixels_per_inch,
            xppi * 1000);
mode_common_setup_;
```

The result will be an absolutely monstrous gf file, which must then be converted to an even more overwhelming mnemonic file by way of gftype. Actually, it takes two runs of gftype, one with minimal output to get the character widths from the terminal lines, and one to get the values for the character bounding box. In Unix command line conventions, the command for the second run is

```
gftype -m cmr9.8030gf
```

to get the mnemonic lines only. We throw away ninety percent of the output from gftype, using only the lines that read

```
. . . char 82: 40<=m<=752 -22<=n<=682
```

which provide values that correspond with the Adobe Font Metric Character bounding box.

Of course, this doesn't look very much like an afm bounding box specification yet; the lines still have to be converted to

```
C 82 ; WX 756 ; N R ; B 40 -22 752 682 ;
```

The conversion is achieved by an interplay of sed and awk scripts which are too detailed to be explored here. The scripts provide symbolic names for all characters in the PostScript fashion, and the names used match PostScript character names as closely as possible. The encoding schemes TeXtext, TeXtypewritertext, TeXmathitalic, TeXmathsymbols and TeXmathextension are provided for. For most coding conventions the match between PostScript Adobe Encoding and the Computer Modern afm files is complete, but the mathematics fonts have some characters that are unknown to the PostScript repertory, so their names are adapted from those in the *TeXbook*. A run of the executable script gf2afm produces the following messages, which give some idea of what is going on behind the scenes.

```
Making gf file at 1000 dots per (true) em
 this takes a while . . . \
       don't get impatient
 Running gftype on cmsy10.7227gf\
       for character widths
 running gftype on cmsy10.7227gf\
       for character bounding boxes
 Getting slant and space values\
       from cmsy10.tfm
 making cmsy10.afm
 extracting kerns from cmsy10.pl
 Ambiguous values (if any) in KernData
 Computer Modern Roman has these\
       for k<-a and v<-a
 In this instance it happens not to matter.
```

The cmsy10.afm file that results is far too long to be illustrated in its entirety but enough of it can be printed to show both the similarities to and a few differences from a regular Adobe afm file.

```
StartFontMetrics [2.0]
Comment Created by gf2afm
Comment Scripts composed by
Comment Internet address:
Comment Creation Date:
FontName cmsy10
FullName Computer Modern\
        Math Symbols 10 point
FamilyName Computer Modern
EncodingScheme TeXmathsymbols
ItalicAngle -14
Stretch 0
Shrink 0
Xheight 431
Quad 1000
Extraspace 0
Numerator1 677
Numerator2 394
Numerator3 444
Denominator1 686
Denominator2 345
Superscript1 413
Superscript2 363
Superscript3 289
Subscript1 150
Subscript2 247
Superscriptdrop 386
Subscriptdrop 50
Delimiter1 2390
Delimiter2 1010
Axisheight 250
CapHeight 682
FontBBox -29 -960 1117 774
StartCharMetrics 128
C 0 ; WX 778 ; N minus ; B 83 230 694 269 ;
       .
       .
       .
C 127 ; WX 779 ; N spade ; B 55 . . . ;
C 128 ; WX 0 ; N space ; B 180 0 180 0 ;
```

[5] In Computer Modern, the parentheses reach to the limits of the type body, but this is far from universally the case. If you depend on this assumption when working with Palatino, you can get very strange effects.

```
EndCharMetrics
StartKernData
StartKernPairs 26
KPX Amathcallig minute 194
        .
        .
        .
KPX Zmathcallig minute 139
EndKernPairs
EndKernData
StartComposites 0
EndComposites
EndFontMetrics
```

At the head of this file I have included some dimensions that are absolutely unknown to PostScript fonts. I suppose that there may be a few applications which make use of afm files and which would break on encountering these, but such applications are probably never going to be used with Computer Modern fonts. It seems better therefore not to throw any potentially useful information away. In addition there is always the hope of teaching by example. Perhaps some of these values may be taken up by the wider PostScript world if they are seen to be useful. It should also be remembered that unlike most PostScript fonts this has a designsize which is very much a part of the name. Even when the metric file is given in this PostScript derived format, cmsy9 is distinctly different from cmsy10.

At the end of the CharMetrics list is the line

```
C 128 ; WX 0 ; N space ; B 180 0 180 0 ;
```

TEX, of course has no need for or use for space as a font element, but the character occurs in all PostScript font codings, even in pi fonts, and programs such as afm2tfm make use of the character width of the space to establish some of the other dimensions in the file. Leaving the character out has unfortunate results, but there is no need to take up a useful code position with it. (I actually remove the character from the vpl file produced by afm2tfm, once it is no longer needed.) TeXtext encoding requires a more elaborate addition at the end of the CharMetrics section.

```
C 128 ; WX 333 ; N space ; B 180 0 180 0 ;
C -1 ; WX 625 ; N Lslash ; B 33 0 582 682 ;
C -1 ; WX 277 ; N lslash ; B 33 0 255 693 ;
```

and close to the end of the file

```
StartComposites 2
CC Lslash 2 ; PCC lcross 0 0 ; PCC L -41 0 ;
CC lslash 2 ; PCC lcross 0 0 ; PCC l 0 0 ;
EndComposites
```

These characters are expressed as kern pairs in a tfm file, but it seems best in the PostScript-derived environment to treat them as composites. The lcross resides in code position 32, where Adobe encoding would put the space character, and this is yet another reason for moving the space outside the normal range of Computer Modern 128-character fonts.

A full set of afm files for Computer Modern has been made available on the CTAN archive, and will be part of the final UnixTEX distribution when it is released. together with the gf2afm tool itself. What you do with them is up to you, but many of the refinements in Alan Jeffery's fontinst package expect afm input. I continue to use the awk and sed scripts that I had already developed when I first learned of fontinst, but this is not the place to go into those in detail. The history of one application, however, shows how advantageous it can be to develop accented character sets through virtual font manipulation.

In 1987, Walter Andrews and I described the Ottoman Text Editions Project at the Eighth Annual Meeting of TUG. The fonts for that stage of the project were created by a laborious effort of recoding, using Knuth's accent.mf as the basis for a file of macro definitions, and for a full font set at 300dpi, we found we had to have nearly a megabyte of additional storage. More seriously, the effort of keeping up with even the minor adjustments and alterations of Computer Modern was not likely to be made and, in fact, it was not made. The next stage was to learn from Silvio Levy's systematic use of saved pictures, and to develop a set of macros which could draw on the character definitions in existing Computer Modern files and assemble them into the necessary composite character glyphs. This worked well enough, but it required a very large version of METAFONT to store all the saved pictures. Even with the most careful pruning of unused pictures, the Turkish characters could rarely be compiled on any PC version of the program. Moreover, when they were compiled the storage requirements were the same as for the previous version. With the set of tools I now have available, I create an afm file with a full set of the composites needed for Turkish and most of the other accented composites as well. That leaves about ten characters that must still be coded directly, but in this case I am dealing with characters that have no parallel in any Computer Modern file. The result is a Turkish "expert" font, to borrow from common foundry terminology, and it is tiny. Even the magstep4 compilation is only just over 1600 bytes.

Except where characters are called out from the Turkish "expert" font, the "raw" fonts for Turkish are the regular Computer Modern fonts. The virtual font tfm and vf files map out to absolutely standard cm* fonts from Knuth's original set. The

space needed to accommodate OTEP Turkish fonts has dropped from a megabyte or so to about 100 kilobytes and, more importantly, the characters in the composite alphabet will never get out of step with their sources.

Tools and Techniques for Outline Fonts

Much as I admire the polished precision of Computer Modern — and you cannot work deeply into its details without admiring it — I am delighted by the variety that PostScript outline fonts bring to TEX. We have gone through quite an upheaval in the past generation. In the years before the Lumitype metamorphosed into the Photon, type was a pretty democratic commodity. You had to be able to afford the investment in cases of lead type, but if you could do that, the foundry had no interest in limiting your choice of sticks, frames, presses etc. Monotype and Linotype rather sewed up the volume business with their specific technologies,[6] but it was possible to get matrices for either machine from independent foundries. I know certainly of one Arab venture which specialized in the independent production of matrices for the Linotype.

All that changed with the advent of cold typesetting. Plate fonts for the Photon ran on the Photon; filmstrips for the C/A/T typesetter, whose characteristics are still deeply embedded in `troff`, ran only on that system, and the cost of a new design for any of these systems was near astronomical. Early digital systems were no better. The VideoComp had its own unique run-length encoding which, as I know to my own personal regret, was useless on Alphatype, APS or Compugraphic machines. Years ago, when I visited some of these firms in an attempt to advertise the virtues of TEX and METAFONT, I was told by one old hand in the font department of Compugraphic that the actual typesetting machine was priced with a very small margin of profit. "We sell the machines to create an appetite for fonts", he said, and the necessary assumption behind that scheme was that the company had absolute monopoly control over fonts for its machines. METAFONT and PostScript have brought this brief period of vertical integration to an end. Font foundries are once again independent of the makers of typesetting machinery, and the cost of a really well cut case of PostScript type has fallen to less than what you would have paid for a couple of isolated characters two or three decades ago.

PostScript fonts are a splendid resource for the TEX user, but not necessarily in the form in which they are supplied by the foundry. Adobe Font Metric files have their virtues, but they convey far less information than TEX Font Metric files, and the majority of documentation systems don't seem to use the information anyway. The notion of programmable ligature combinations in an `afm` file is rudimentary at best, and the integration of ligature programs with hyphenation patterns appears to be just about unique to TEX. When 'fi' and 'fl' are in one font and 'ff' 'ffi' and 'ffl' in an entirely different one, there is no easily accessible way to make TEX's word-breaking algorithms work at full efficiency other than to resort to virtual fonts. Over the past few years I have resorted to a combination of standard Unix tools together with Rokicki's `afm2tfm` to set up my virtual fonts. Alan Jeffrey offers a different method, one which does not depend on the Unix collection of tools. It is not possible to explore the wide range of possibilities in this paper, but I shall conclude with the mention of one very useful tool which was put together by a colleague at the University of Washington.

One of the first requirements for an intelligent remapping of a PostScript font is that you know what is in it. This is not as straightforward as it sounds. The upper half of a "standard" font table is sparsely populated with characters in no easily discernable order. (I am not yet certain whether this part of the encoding vector, from position 128 to 255, is part of Adobe Standard Encoding or only positions 0-127.) That still leaves out about a third of the characters supplied with a normal text font and that third has no fixed code position. In the new class of "superfonts" there are, in fact, more unencoded than encoded characters. The version of Courier released to the X Consortium by IBM is immense, and includes a great wealth of pi characters for forms makeup. You have to plug these into an encoding vector before they can be used, but since they are not there already, you often have no hint that they exist, and you have neither their names nor their appearances to work from. Dylan Mcnamee's `showfont.ps` creates a table of the regularly encoded characters as do several other known PostScript utilities, but it also extracts the names of all the unencoded characters and displays them as well. This program has been offered to the Free Software Foundation for inclusion in the appropriate package, and will be made available in

[6] Ladislas Mandel, in a recent article in *EPODD*, notes that the dominance of these two firms effectively drove all French foundries out of the business of cutting text fonts.

the UnixTeX distribution and in the ftp directory at `june.cs.washington.edu`. Further distribution is encouraged.

Bibliography

Adobe Systems Inc. PostScript Language Reference Manual, 2nd Edition. Reading, Mass.: Addison-Wesley, 1990.

Andrews, Walter and MacKay, Pierre. "The Ottoman Texts Project." *TeXniques* **5**, pages 35-52, 1987.

Haralambous, Yannis. "Hyphenation patterns for ancient Greek and Latin." *TUGboat* **13** (4), pages 457-469, 1992.

Knuth, Donald E. *The TeXbook*. Reading, Mass.: Addison-Wesley, 1984.

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

Mandel, Ladislas. "Developing an awareness of typographic letterforms." *Electronic Publishing — Origination Dissemination and Design* **6** (1), pages 3-22, Chichester: John Wiley and Sons, 1993.

Appendix

The METAFONT Input File modetest.mf (Partial Listing)

```
% Additional characters may be selected by adding
% additional elseif lines at will, but there has to come
% a point of diminishing returns.
% This file can be named on the command line, as in
% cmmf modetest input cmr10
%
string currenttitle;
def selective expr t =
  currenttitle:= t;
  if t = "The letter K" : let next_ = use_it_
  elseif t = "The letter W" : let next_ = use_it_
  elseif t = "The letter o" : let next_ = use_it_
  else: let next_ = lose_it_ fi; next_ enddef;
% Add _ to the macro names used by iff to avoid confusion.
def use_it_ = message currenttitle; enddef;
def lose_it_ = let endchar = fi; let ; = fix_ semi_
  if false: enddef;
let cmchar = selective;
```

The METAFONT Input File sromanu.mf (Partial Listing)

```
string currenttitle;
def exclude_I expr t =
  currenttitle:= t;
  if t = "The letter I" : let next_ = lose_it_
  else: let next_ = use_it_ fi; next_ enddef;
% Add _ to the macro names used by iff to avoid confusion.
def use_it_ = relax; enddef;
def lose_it_ = let endchar = fi; let ; = fix_ semi_
  if false: enddef;
let cmchar = exclude_I;
input romanu
let cmchar=relax;

cmchar "The letter I";
beginchar("I",max(6u#,4u#+cap_stem#),cap_height#,0);
         .
         .
         .
math_fit(0,.5ic#); penlabels(1,2,3,4,5,6,7,8); endchar;

endinput;
```

The METAFONT Input File gribyupr.mf (Partial Listing)

```
def selectupper expr t =
  currenttitle:= t;
  if t = "The letter C" :
code_offset := Cigmalunate - ASCII"C"; let next_ = use_it_
  elseif t = "The letter D" : let next_ = lose_it_
  elseif t = "The letter F" :
code_offset := Digamma - ASCII"F"; let next_ = use_it_
  elseif t = "The letter G" : let next_ = lose_it_
                    .
                    .
                    .
  elseif t = "The letter Y" : let next_ = lose_it_
```

```
  elseif t = "The letter P" :
code_offset := ASCII"R" - ASCII"P"; let next_ = use_it_
  else: code_offset := 0; let next_ = use_it_ fi; next_ enddef;

def recodeupper expr t =
  currenttitle:= t;
  if t = "Uppercase Greek Xi" :
    code_offset := ASCII"C" - oct"004";
  elseif t = "Uppercase Greek Delta" :
    code_offset := ASCII"D" - oct"001";
  elseif t = "Uppercase Greek Phi" :
    code_offset := ASCII"F" - oct"010";
                .
                .
                .
  elseif t = "Uppercase Greek Omega" :
    code_offset := ASCII"W" - oct"012";
  elseif t = "Uppercase Greek Psi" :
    code_offset := ASCII"Y" - oct"011";
  else: code_offset := 0; fi; next_ enddef;

let cmchar = selectupper;
input romanu
let cmchar = recodeupper;
input greeku

% Now we restore cmchar and code_offset to defaults.
let cmchar = relax;
code_offset := 0;
```

A File which can be read by both TEX and METAFONT

```
%
% These macros make it possible to read *.map files as either
% \TeX{} or METAFONT input
%
% A well-known conditional test in METAFONT;
% It creates mismatch of character tokens 'k' and 'n' in TeX
\if known cmbase: %                              Interpret as a
                  %                              METAFONT file
     let re_catcode=relax; let let_=gobble; let no_let=gobble;
  else:
     message "Must have cmbase loaded for this, or else some macros
from it" ;
%
% END OF METAFONT INTERPRETATION--TeX INTERPRETATION FOLLOWS
%
  \else %                              Interpret as a TeX file
     \catcode`\_11 % allow underscore in csnames as in METAFONT
     \def\re_catcode{\catcode`\=12 \catcode`\;12 \catcode`\_8}%
     \def\ignore_to_comment#1#2{}%
% Now activate all the characters from ^^80 to ^^ff
     \count255=`\^^80
     \loop \ifnum\count255 < `\^^ff
       \catcode\count255\active \advance\count255 by 1 \repeat
     % activate the \^^ff character separately if it is needed
     \expandafter\input\the\digraphs % Filename in a \toks register
     \catcode`\;0 % treat the first ; (required by METAFONT) as an
  % escape
     \catcode`\=14 % treat the = in the METAFONT part as a comment
   % character
     \let\let_\let \let\no_let\ignore_to_comment
  \fi
```

The code lines based on this scheme look a bit bizarre, but they do the trick.

```
%
% This is a rather specialized version of the map file,
% developed for Greek only.  There are certain restrictions
% in this case, because we do not want to alter Silvio Levy's
% source code--only the mappings.
% The upper level codes (^^80--^^ff) are based on a version of
% Greek Keys (a word-processor package for Macintosh, distributed
% through the American Philological Association), but the mapping
% is worked out by experience not from any documentation, and
% local customization often alters even this mapping.
% Consistency is provided by the ASCII digraphs
% and trigraphs to which all word-processor codes are remapped
% before they are used in TeX.  These digraphs and trigraphs
% (even tetragraphs in the case of iota subscript) are very close
% to Ibycus/TLG betacode, except for the unfortunate uppercasing
% of betacode.
%
%  a known set of word-processor       Some "hidden" characters
%  equivalents is "let_" for TeX     Only METAFONT needs to know
%  \no_let is used where there         what is in this column
%  seems to be no certain mapping
%
\let_ ^^80;oxy_tone     = ASCII"'";        endash = 0;
\let_ ^^81;bary_tone    = ASCII"`";        emdash = oct"177";
\let_ ^^82;peri_spomenon  = ASCII"=";      null_space = ASCII" "; % zero width
\let_ ^^83;sp_lenis     = ASCII")";        dieresis = oct"053"; % use plus sign
\let_ ^^84;sp_asper     = ASCII"(";        minute = ASCII"&"; % prime for numbers
\let_ ^^85;lenis_oxy    = oct"136";        asper_glyph = ASCII"<"; % location in
\let_ ^^86;lenis_bary   = oct"137";        lenis_glyph = ASCII">"; % Levy font
\let_ ^^87;lenis_peri   = oct"134";        guillemotleft = ASCII"{"; % two small
\let_ ^^88;asper_oxy    = oct"207";        guillemotright = ASCII"}"; % awks
\let_ ^^89;asper_bary   = oct"203";        iotasubscript = ASCII"|";
\let_ ^^8a;asper_peri   = oct"100";        boundarychar := oct"377"; % N.B. :=
\no_let \dmy;quoteleft  = oct"034";        quotedblleft = oct"253";
\no_let \dmy;quoteright = oct"035";        quotedblright = oct"257";
\no_let \dmy;diaeroxy   = oct"043";        bracketleftbt = oct"363";
\no_let \dmy;diaerbary  = oct"044";        bracketrightbt = oct"367";
%
% alpha with accents
%
\let_ ^^8b;a_oxy        = oct"210";        Digamma = ASCII"V";
\let_ ^^8c;a_bary       = oct"200";        digamma = ASCII"v";
\let_ ^^8d;a_peri       = oct"220";        Koppa = oct"001";
\let_ ^^8e;a_len        = oct"202";        koppa = oct"002";
\let_ ^^8f;a_aspr       = oct"201";        sampi = oct"003";
\let_ ^^90;a_lenoxy     = oct"212";        Cigmalunate=oct"004";
\let_ ^^91;a_asproxy    = oct"211";        cigmalunate=ASCII"J";
\let_ ^^92;a_lenbary    = oct"223";        % "J" is all that's available
\let_ ^^93;a_asprbary   = oct"213";        sigmafinal=ASCII"j";
\let_ ^^94;a_lenperi    = oct"222";        r_aspr = oct"373"; % GreekKeys "="!!
\let_ ^^95;a_asprperi   = oct"221";        r_len  = oct"374";
%
% alpha with accents and iota subscript
%
\let_ ^^fb;a_isub           = oct"370";  angleleft = oct"303";
\let_ ^^96;a_oxyisub        = oct"214";  angleright = oct"307";
\let_ ^^97;a_baryisub       = oct"204";   braceleft = oct"333";
\let_ ^^98;a_periisub       = oct"224";   braceright = oct"337";
\let_ ^^99;a_lenisub        = oct"206";   dagger = oct"375";
\let_ ^^9a;a_asprisub       = oct"205";   daggerdbl = oct"376";
        .
        .
        .
```

Symbolic Computation for Electronic Publishing

Michael P. Barnett
Department of Computer and Information Science,
Brooklyn College of the City University of New York, Brooklyn, NY 11210
Internet: barnett@its.brooklyn.cuny.edu

Kevin R. Perry
Interactive Computer Graphics Laboratory,
Computer and Information Technology,
Princeton University, Princeton, NJ 08540
Internet: perry@princeton.edu

Abstract

Recently, we developed and reported some novel ways to make MATHEMATICA produce derivations that contain conventionally structured narratives and formulas, using a procedure that interprets files containing expressions, which MATHEMATICA evaluates symbolically, control information and text. Now, this work is extended to give TEX typeset output. It supports the interactive crafting of publications that contain mechanically generated formulas and diagrams, for teaching and research, in subjects that use mathematics and other algorithmic methods. It goes beyond the direct capabilities of the built-in `Splice` and `TeXForm` functions.

1. Introduction

Recent dramatic gains in the accessibility and power of workstations has given symbolic computation greatly increased exposure. Several monographs describe MATHEMATICA (Wolfram 1991) and other computer algebra systems. Interactive use, with and without "notebooks", and batch runs that produce lengthy FORTRAN statements for numerical evaluation on mainframe computers are common. A relatively underutilized application is the combined use of electronic typesetting and symbolic computation to produce research journals, monographs, text books, reference compendia, problem books and other documents containing mathematical formulas that are:

- the result of lengthy proofs and derivations,
- burdensome to copy and check,
- numerous and closely related to each other,
- needed in different notations,
- internally repetitive (e.g. matrices whose elements are written in a common form).

Computerized symbolic calculation of the formulas in all these cases can save considerable time and effort that is needed, otherwise, to produce the results, type the manuscripts, and read and correct the proofs. The need to print mathematical formulas that were produced by primitive symbolic computations led to some of the early work on electronic typesetting over 30 years ago (Barnett 1965) — the formulas were too numerous and lengthy to derive by hand. The present paper describes some current activity that uses MATHEMATICA with plain TEX (Knuth 1986), LATEX (Lamport 1994) and AMS-TEX (Spivak 1986). It builds on our `autorun` procedure and `bilo` package (Barnett 1991; Barnett 1993; Barnett 1994), and the application of these to atomic and molecular structure calculations (Barnett 1991a; Barnett 1994a), robot kinematics (Chen 1991) and other topics.

To introduce some of the principles that underly this work, the box below depicts a workstation screen during a simple MATHEMATICA session. The default prompt for input has the form "`In[n]:=`". The value that the system computes for each input statement is displayed alongside an "`Out[n]=`" tag. Here, statement 1 assigns a simple expression to `s`. Statement 2 replaces `n` by 1 and expands the result.

```
In[1]:= s = (1 + x/(x+1))^(n+1)

                  x   1 + n
Out[1]= (1 + -----)
                1 + x

In[2]:= s /. n -> 1 // Expand

                   2
                  x           2 x
Out[2]= 1 + -------- + -----
                     2    1 + x
             (1 + x)
```

MATHEMATICA evaluates:

$$a\ /.\ b \text{ -> } c$$

by substituting c for b throughout a. Also, b -> c is called a "transformation rule", a "replacement rule" or just a "rule". The function f of x is written as:

$$f[x], \quad x//f, \quad f@x$$

These are synonymous. f is called their "head".

2. Interpreting a control file

Our typesetting work builds on the autorun procedure (Barnett 1991), that reads a file of MATHEMATICA statements and, in the default mode, mimics interactive operation. Thus, given the control file demo1 consisting of the two records:

```
s = (1 + x/(x+1))^(n+1)
s /. n -> 1 // Expand
```

a continuation of the MATHEMATICA session of Section 1 is shown in the next box. Only the autorun statement is typed. It simulates the earlier interactive action, by interpreting the control file and generating "In" and "Out" tags that combine the sequence numbers in the MATHEMATICA and autorun sessions. The screen display can be recorded, e.g., as a UNIX script file, and printed.

```
In[3]:= autorun[demo1]

In[3.1]:= s = (1 + x/(x+1))^(n+1)

                     x   1 + n
Out[3.1]= (1 + -----)
                   1 + x

In[3.2]:= s /. n -> 1 // Expand

                  2
                 x        2 x
Out[3.2]= 1 + -------- + -----
                     2   1 + x
               (1 + x)
```

Often, users want to see the output of the successive symbolic evaluations, without a playback of the input. autorun provides an "outputOnly" mode, and lets the control file contain formatting and related "directives" that do not get displayed, and text that does. These are labeled by # and $*$ symbols, as in the file demo2, shown in the next box. It contains two bilo formatting functions. The first reverses the reordering of the exponent — numbersLast simply moves numerical terms to the right in its target. The second treats the target as a polynomial in x.

```
# outputOnly
* Given
s = (1 + x/(x+1))^(n+1)
# format = toThe[n+1][numbersLast]
* then the expansion, when n=1, is
# format = sortToIncreasePowersOf[x]
s /. n -> 1 // Expand
```

Interpreting this file gives:

```
In[4]:= autorun[demo2]

Given

          x    n + 1
 (1 + -----)
        1 + x

then the expansion, when n=1, is

                        2
     2 x          x
1 + ----- + --------
    1 + x           2
             (1 + x)
```

3. Producing typeset output

A file demo2.tex was produced from demo2 by:

autorecord[demo2]

This invokes autorun in a mode that converts the evaluated expressions to TEX. autorecord then writes the relevant portion of the screen display to an output file, invokes LATEX or, optionally, TEX, and a previewer. Then it prints, if requested. In the TEX file that produced this paper, \input demo2 imported the TEX coded contents of the next box.

> Given
> $$\left(1+\frac{x}{1+x}\right)^{n+1}$$
> then the expansion, when n=1, is
> $$1+\frac{2x}{1+x}+\frac{x^2}{(1+x)^2}$$

The "n=1" in the second line of text is set in math mode by putting delimiters around it in the input.

A file close to demo2.tex can be produced from a slightly different control file using the built-in Splice function of MATHEMATICA, which uses the built-in TeXForm function to do the TEX encoding of individual expressions. We use the procedure toTeX, that is part of our forTeX package, to do the encoding. It provides greater flexibility and control.

4. Compound heads

Formatting the final result in demo2 involved, implicitly, the evaluation of:

```
sortToIncreasePowersOf[x][1 + ...]
```

In this, the head itself is a function of x.

```
sortToIncreasePowersOf[x]
```

We call this a "compound head". In general:

$$f[u_1, u_2, \ldots][v_1, v_2, \ldots]$$

denotes a function $g[v_1, v_2, \ldots]$, where g is the function $f[u_1, u_2, \ldots]$. Thus, the entire expression is a function of $u_1, u_2, \ldots, v_1, v_1 \ldots$ The principle is extended, e.g. in a[b][c][d][e]. Invented by Ruffini two centuries ago (see (Ruffini 1799) and (Cajori 1919, page 81)), the notation was reinvented by Curry (Curry et al. 1958; Hindley and Seldin 1986). We use it extensively, for convenience both in symbolic computation and when encoding to TeX. The ability of MATHEMATICA to handle it is very important.

5. The basic toTeX conventions

autorecord hides the TeX output of toTeX from the user. To show it here, we display a MATHEMATICA expression s, the "leads to" symbol ⤳ and the typeset value of toTeX[s] as, for example, in:

sin[theta] sin[pi+theta] ⤳ $\sin\theta\sin(\pi+\theta)$

This example, incidentally, illustrates selective parenthesization which TeXForm does not provide.

For processing by toTeX, elementary expressions are represented using standard MATHEMATICA conventions, that include == between the sides of an equation. The characters listed in (Knuth 1986, Appendix F, Tables 1-14) and many other objects are denoted by TeX-like names that omit the backslash. Thus:

alpha,	beta,	...	⤳	α,	β,	...
aleph,	hbar,	...	⤳	$\aleph$,	$\hbar$,	...
pm,	mp,	...	⤳	$\pm$,	$\mp$,	...
leq,	prec,	...	⤳	$\leq$,	$\prec$,	...

We use function notation to represent fonts, subscripts, superscripts, decorations, special bracketing and binary expressions. Thus:

bf[A + B + C]	⤳	$\mathbf{A+B+C}$
P[sub[n]][sup[m]][x]	⤳	$P_n^m(x)$
hat[x], overline[x+y]	⤳	$\hat{x}$, $\overline{x+y}$
enpr[x], encr[x, y]	⤳	(x), $\{x,y\}$
cap[cup[A, B], C]	⤳	$A\cup B\cap C$

enpr abbreviates "enclose in parentheses" and sapr provides parentheses that are sized automatically. Corresponding names with br, cr, lr and so forth provide square brackets, curly braces, < > and other enclosing symbols.

We use compound heads to represent several functionals. These include:

sum[i, j, k][f[i]]	⤳	$\sum_{i=j}^{k} f(i)$
limit[t -> 0][h[t]]	⤳	$\lim_{t\to 0} h(t)$
D$[{x, 2}, y][phi]	⤳	$\frac{\partial^3\varphi}{\partial x^2\partial y}$

bilo has many functions that operate on these representations in symbolic calculations, e.g. rightExpand performs the reduction:

$$\sum_{i=j}^{k} f(i) \rightarrow \sum_{i=j}^{k-1} f(i) + f(k)$$

Denoting MATHEMATICA evaluation by ⇒, the action of rightExpand is specified also by:

```
sum[i, j, k][f[i]] // rightExpand
⇒ sum[i, j, k-1][f[i]] + f[k]
```

Other bilo functions perform the reductions:

$$\sum_{i=j}^{k} f(i) \rightarrow \sum_{i=j}^{k+1} f(i) - f(k+1)$$

$$\sum_{i=j}^{k} (a+b+\ldots) \rightarrow \sum_{i=j}^{k} a + \sum_{i=j}^{k} b + \ldots$$

$$\sum_{i=j}^{k} f(i) + \sum_{i=k+1}^{l} f(i) \rightarrow \sum_{i=j}^{l} f(i)$$

and several related operations.

The compound head notation allows numerous variations, e.g. in symbolic calculations that involve integrals, we use expressions that include:

integral[x][y] ⤳ $\int y\,dx$

integral[{x_1, x_2}][y] ⤳ $\iint y\,dx_1dx_2$

integral[{x_1, x_2, x_3}][y]

⤳ $\iiint y\,dx_1dx_2dx_3$

integral[x, u, v][y] ⤳ $\int_u^v y\,dx$

integral[{{x_1, u_1, v_1}, {x_2, u_2, v_2}}][y]

⤳ $\int_{u_1}^{v_1}\int_{u_2}^{v_2} y\,dx_1dx_2$

integral[{{x_1, u_1, v_1}, ..., {x_n, u_n, v_n}}][y]

⤳ $\int_{u_1}^{v_1}\ldots\int_{u_n}^{v_n} y\,dx_1\ldots dx_n$

lineIntegral[dot, L, s][bf[V]]

⤳ $\int_L \mathbf{V}\cdot d\mathbf{s}$

surfaceIntegral[S][f] ⤳ $\iint_S f\,d\mathbf{S}$

6. Operating on toTeX input

Other names can be converted to TeX-like names by replacement rules that precede the statements which

invoke toTeX. The rules are input as directives (see Section 9). So are targeting functions that put function heads onto specific subexpressions as in:

```
f[a] + g[b] + h[c] //
  inSuccession[
    toThe[f][bf]
    collectivelyToTerms[
      containingAny[g, h]][sabr]]
⇒  bf[f[a]] + sabr[g[b] + h[c]]
```

$\rightsquigarrow$ **f(a)** $+ [g(b) + h(c)]$

The targeting function here is read "apply the function bf to the part of the target that has the head f, then apply the function sabr collectively to the parts that contain either g or h." bilo contains an easily extensible set of functions that address terms, factors, elements of a list or a vector or a matrix, coefficients, arguments, parts of a fraction or a relationship, patterns and explicit items.

Suppressing parentheses: toTeX converts the brackets in an arbitrary MATHEMATICA function to parentheses. unpr elides them. Thus:

```
f[x], f[unpr[x]], f[unpr[circ]][unpr[x]]
```

$\rightsquigarrow$ $f(x)$, $f\,x$, $f \circ x$

Varying the style: Our default style for square roots is controlled by the toTeX variable defaultSqrtStyle. Initially it is 1, with the effect:

```
sqrt[1 - sqrt[1 - Delta^2]]
```

$\rightsquigarrow$ $\sqrt{1-\sqrt{1-\Delta^2}}$

Setting it to 2 leads to output in the style:

```
sqrt[1 - sqrt[1 - Delta^2]]
```

$\rightsquigarrow$ $(1-(1-\Delta^2)^{\frac{1}{2}})^{\frac{1}{2}}$

The default can be changed by a simple MATHEMATICA assignment. Also, in the formation of the TEX codes by toTeX each sqrt[x] is converted to the intermediate form:

style[sqrt, m][x]

with m set to the value of defaultSqrtStyle that is current. An individual square root in a formula can be set in a different style n by changing it to style[sqrt, n][x] before applying toTeX. To do this, the transformation rule

sqrt -> style[sqrt, n]

is applied to the appropriate subexpression(s) by a bilo targeting function. Similar tactics are used to style other fractional powers and fractions.

Independent options: Several features in the styling of an integral can be altered independently. toTeX begins the encoding by converting the primary head integral to the form:

style[integral, a_1 -> b_1, a_2 -> b_2, ...]

The system initializes the option values b_i that are associated with the option names a_i. The statement:

setOptions[integral, a_j -> b_j, a_k -> b_k, ...]

is used to change options that, for integrals, include:

- differentialsInNumerator — defaults to no,
- allowRepeatedIntegral — no outputs multiple indefinite integrals with a single ∫ symbol,
- variableInLower — all, multiple, and none include the variable of integration in the lower limit(s) of, respectively, all definite integrals, only multiple definite integrals, and in none,
- variableInUpper — yes and no include and omit the integration variable in the upper limit,
- includeLimits — yes and no allow and preclude the inclusion of limits.

Options are set analogously for other functionals.

7. bilo formatting functions

Rearranging an expression: Often, the default order of an expression imposed by MATHEMATICA does not give the preferred form. Non-atomic objects are represented internally by nested function expressions, for example, (a+b)(c+d) by:

```
Times[Plus[a, b], Plus[c, d]]
```

Formatting often involves rearranging the argument lists of one or more such functions, and insulating the result from automatic reordering. bilo includes a suite of sorting functions based on the action of:

$\{x_1, \ldots, x_n\}$ // partByCriteria[$c_1, \ldots, c_m$]

This evaluates to $\{s_1, \ldots, s_{m+1}\}$ where:

1. for $\nu = 1$ to m, s_ν is the list of x_i for which $c_1(x)$, ..., $c_{\nu-1}(x)$ are false and $c_\nu(x)$ is true,
2. s_{m+1} is the list of x_i for which $c_1(x)$, ..., $c_m(x)$ are false.

The order of the xs in each list s_ν is the same as in the original list $\{x_1, \ldots, x_n\}$.

The higher level sortByPresence[$v_1, v_2, \ldots$] and sortByAbsence[$v_1, v_2, \ldots$] are very useful. The v's can be explicit terminal subexpressions or intermediate level heads or patterns for either. Related functions include those used in Section 2.

The bilo sort functions are applied to the relevant subexpressions by targeting functions. For this kind of work, expressions containing bilo sort functions are much shorter, usually, than is possible with functions that require pairwise ordering criteria.

Further rearrangement requirements include moving the denominator of a fraction to the beginning, as in $\frac{1}{2}(x+y)$ and preventing, say, $-(a+b)$ from opening up to $-a-b$. bilo handles these.

Skeletalizing: The reduction:

$(1+x)^{10}$ // Expand // showTerms[{1, 6, -1}]

$\Rightarrow$ $1 +$ «4 terms» $+ 252\,x^5 +$ «4 terms» $+ x^{10}$

typifies the action of another suite of bilo functions. If p stand for a pointer list, then showTerms[p]

and showFactors[p] act on Plus and Times expressions, showElements[p] acts on lists, vectors, matrices and tensors. showArguments[p] acts on any function. Each show*Objects* function skeletalizes the target. By default, each shows that items are omitted by an expression of the form:

«*n object(s)*»

This can be overridden to display ellipses.

Insertion of codes: If p is a MATHEMATICA pattern that matches one of the terms in a Plus, then applying splitAfter to this Plus breaks the line after the + or − sign that follows the term. splitBefore[p] breaks the line before the sign.

Applying a split function to any non-atomic expression puts the subexpression which is matched onto a new line. insertBefore[p][c] is wrapped by splitBefore[p], where c is a string of TeX codes. splitAfter is handled correspondingly. Further toTeX wrappers will allow convenient control of indention and vertical spacing. Nested targeting expressions can be used to focus on the part of the target into which codes must be inserted.

8. A simple forTeX application

This relates to teaching special functions of mathematical physics. The statement:

autorecord[legendreAuto]

reads the file legendreAuto, shown in the next box, and constructs the file legendreAuto.tex.

```
* {\bf Demonstrating orthogonality.\ } We
* integrate the product of two Legendre
* polynomials of different degree. Consider
s1 = integral[x, -1, 1][P[2, x] P[4, x]]
* Substituting explicit polynomials for
* the $P_n(x)$ gives
# beginLeftAlignedGroup["="]
s2 = s1 // evaluateAndHoldSpecialFunctions
s3 = s2 // releaseAndExpandAll
# endLeftAlignedGroup
* Then term by term integration gives
# beginRunonGroup["="]
s4 = s3 // integrateTermByTermAndHold
s4 // allowEvaluation
# endRunonGroup
```

The statement \input legendreAuto in the file that set this paper produced the contents of the box in the right hand column. By default, the outputOnly mode is in effect. So is the sandwiching of individual TeX coded output expressions between centering delimiters. The first three records in the control file begin the text. The next record is the MATHEMATICA statement that assigns, to the variable s1, the integral to be evaluated. The next two records provide the next piece of text. As regards the rest of the file:

1. beginLeftAlignedGroup puts the codes to begin a left-aligned multi-formula display before the next coded expression, and an = symbol at the right. Also, it sets a switch that includes codes to continue the display, when subsequent input expressions are processed.
2. evaluateAndHoldSpecialFunctions is a short MATHEMATICA procedure, written for this application, that converts Legendre, Laguerre and other special polynomials, written as P[n, x], L[n, x], ..., to explicit polynomials that are kept separate. Here, this separation stops MATHEMATICA multiplying the denominators.
3. endLeftAlignedGroup writes the codes to end a multi-formula display and resets switches.
4. releaseAndExpandAll removes the insulation around individual parts of the target expression and applies the built-in ExpandAll function.

Demonstrating orthogonality. We integrate the product of two Legendre polynomials of different degree. Consider

$$\int_{-1}^{1} P_2(x)P_4(x)\,dx$$

Substituting explicit polynomials for the $P_n(x)$ gives

$$\int_{-1}^{1}\left(\frac{-1+3x^2}{2}\right)\left(\frac{3-30x^2+35x^4}{8}\right)dx =$$
$$\int_{-1}^{1}\left(-\frac{3}{16}+\frac{39x^2}{16}-\frac{125x^4}{16}+\frac{105x^6}{16}\right)dx$$

Then term by term integration gives

$$-\frac{3}{8}+\frac{13}{8}-\frac{25}{8}+\frac{15}{8} = 0$$

5. beginRunonGroup, beginLeftAlignedGroup, and endRunonGroup, endLeftAlignedGroup produce analogous effects.
6. integrateTermByTermAndHold distributes integration over the Plus in the integrand, and stops the result coalescing.
7. allowEvaluation removes the Hold around each term.

A large variety of worked examples to help teach orthogonality of special functions can be generated by applying autorecord to input files that follow the general style of this prototype, use the same bilo functions, and cycle through sets of parameters.

9. Directives

The control file directives are simply statements that use raw MATHEMATICA or invoke procedures

in our `bilo`, `forTeX` and related packages. Thus, in Section2, `bilo` sorting functions are assigned to `format` — a surrogate head that is applied to each output expression before it is encoded. The directives in Section 8 are the names of TEX functions that put the delimiters for display math into the output, and control several program switches.

Statements are written as directives that:

1. substitute TEX-like names for other identifiers,
2. convert function expressions that carry indexes as ordinary arguments into representations that lead to sub- and superscripting,
3. change font and impose bracketing,
4. fine tune the style as described in Section 5,
5. perform substantive steps that are not typeset.

There is further flexibility. `autorecord` is used in the `displayBoth` mode to document the symbolic algebra part of a mechanized derivation. It plays back the input in typewriter font and the evaluated expressions as displayed mathematics. Options allow typewriter font for both. Each input expression can be run-on or left-aligned above the "⇝" symbol and conventionally displayed result. Further styles of alignment are provided for the `outputOnly` and `outputBoth` modes. Provision can be made to number the equations.

Options can be changed dynamically. Also, the directives can be made conditional on flags that are set at execution time, to vary the style of output from different runs using the same control file. The content can be varied by conditional directives that make evaluation occur silently until balancing directives are reached, or bypass text and/or evaluation completely. This can be used, e.g. to produce terse and detailed derivations from a single file, or problem sets with and without solutions.

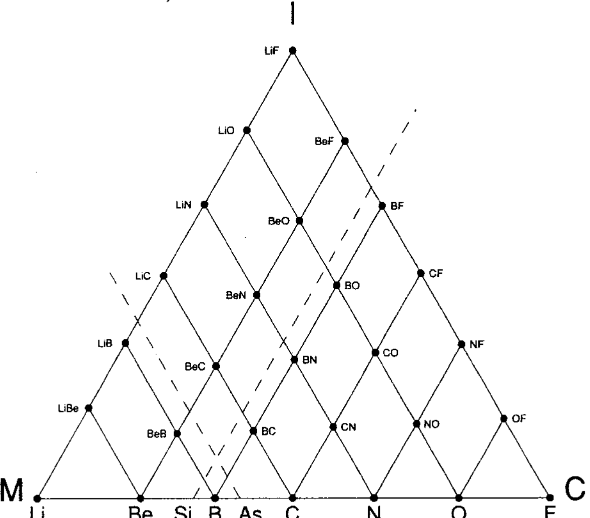
Four pre- and postprocessing functions, that default to `Identity`, which returns the argument unchanged, provide part of the flexibility.

1. `ipc` — this converts input statements into a preferred form that is recorded,
2. `ivc` — this converts input statements from a variant notation into valid MATHEMATICA,
3. `opc` — this acts on the immediate result of MATHEMATICA evaluation to give output of preferred appearance,
4. `tpc` — this operates on each block of text before it is displayed.

These parallel, in part the built-in `$PreRead`, `$Pre`, `$Post` and `$PrePrint` functions. `autorun` can document the full extent of activity by displaying any combination of the input as read, the restyled input, the input to MATHEMATICA, the output from MATHEMATICA and the restyled output, alongside tags that default to "`In`", "`in`", "`IN`", "`Out`" and "`out`", respectively.

10. Using graphics

MATHEMATICA has extensive graphics capabilities. It saves graphics objects as POSTSCRIPT files. `psfig` imports these to TEX documents and is very easy to use. The "van Arkel - Ketelaar" diagram below relates to binary compounds of the chemical elements and an atomic property called "electronegativity" (Allen, Leland et al. 1993).



I/M/C character of 1st row binaries

The letters I, M and C at the corners of the triangle stand for ionic, metallic and covalent. We wrote a MATHEMATICA procedure that reads a list of elements and produces the corresponding diagram. This example involves the algorithmic placement of text, with provision for interactive fine tuning.

The ability to import graphics helps check symbolic calculations. Computer algebra is fraught with possible error, though this is seldom discussed. The early stages of a symbolic calculation often produce results that are in the literature, and there is merit to making a comparison.

Our first efforts to check results this way showed that, often, in hand generated formulas, a particular subexpression is represented by a single variable in one place but not another. Machine generated formulas tend to be more consistent. `bilo` was developed, in part, to target the changes that are needed to bring the content and arrangement of computed results into agreement with published material.

Even when this has been done, however, the visual comparison can be tedious. For example, as part of a longer calculation (Barnett 1994a), we reconstructed a 33-term recurrence formula, that occupies half a page of Physical Review (Pekeris 1958). It can be written:

$$\sum_{(\lambda,\mu,\nu)\in S} c_{\lambda,\mu,\nu}(l,m,n)A(l+\lambda,m+\mu,n+\nu) = 0$$

where the c's are simple algebraic expressions in (l, m, n), and S consists of 33 triples whose elements are between -2 and 2. To help other workers see that the results are in agreement, the relevant passage in Physical Review was scanned, and the image dissected into about 40 pieces that contain separate terms or, in some cases, parts of terms, using a suite of image processing tools written locally. The pieces were imported into a document above the respective terms generated by MATHEMATICA, using another utility. This has been done here for terms 1-3 and 32, 33, by conversion to POSTSCRIPT and use of `psfig/tex`. Scaling is varied slightly.

$$4(l+1)(l+2)[-Z+\epsilon(1+m+n)]A(l+2,m,n)$$
$$4(l+1)(l+2)[-Z+\epsilon(1+m+n)]A(l+2,m,n)$$

$$+4(m+1)(m+2)[-Z+\epsilon(1+l+n)]A(l,m+2,n)$$
$$4(m+1)(m+2)[-Z+\epsilon(1+l+n)]A(l,m+2,n)$$

$$+4(l+1)(m+1)[1-2Z+\epsilon(2+l+m)]A(l+1,m+1,n)$$
$$4(l+1)(m+1)[1-2Z+\epsilon(2+l+m)]A(l+1,m+1,n)$$

...

$$+2ln[1-2Z+\epsilon(2m+n+1)]A(l-1,m,n-1)$$
$$2ln[1-2Z+\epsilon(2m+n+1)]A(l-1,m,n-1)$$

$$+2mn[1-2Z+\epsilon(2l+n+1)]A(l,m-1,n-1)$$
$$2mn[1-2Z+\epsilon(2l+n+1)]A(l,m-1,n-1)$$

This technique can be used to publish scanned pictures together with algorithmically constructed graphics in many areas of work.

Setting the output of symbolic computation, using special fonts and macros for chemical formulas, chess situations, musical scores and other iconic displays has still further potential. The use of built-up formulas by organic and biological chemists is well known. These, and the further needs of inorganic and solid state chemists are discussed in (Jensen 1989). We have made a start on interfacing MATHEMATICA with the work reported earlier (Haas and O'Kane 1987; Tutelaers 1992; Taupin 1992).

As a simple example of non-mathematical output produced by MATHEMATICA, two very short procedures that construct a word count and encode it as a table produced the following display from a 10-line passage in Macbeth.

a	3	adder's	1	and	8
bake	1	bat	1	blind	1
...					
trouble	2	wing	1	wool	1
worm's	1				

Distribution

The `bilo` package is in, and the `forTeX` material soon will be in the anonymous ftp library at:

`mondrian.princeton.edu (128.112.224.14)`

Acknowledgements

This work was started under support of NSF grants NAC-25001 and ASC-8500655 and conducted in part during MPB's sabbatical leave in the Chemistry Department and Interactive Computer Graphics Laboratory of Princeton University. We thank K. D. Alexander, L. C. Allen, J. F. Capitani, M. Ross, A. Shulzycki, J. S. Stamm, D. J. Thongs and P. Vince for their comments and cooperation.

References

Allen, Leland C. et al. "Van Arkel — Ketelaar triangles." *J. Molec. Struct.* **300**, pages 647–655, 1993.

Barnett, Michael P. *Computer Typesetting — Experiments and Prospects.* Cambridge, Mass.: MIT Press, 1965.

Barnett, Michael P. "Some simple ways to construct and to use formulas mechanically." *ACM SIGSAM Bulletin* **28** (2), pages 21–29, 1991.

Barnett, Michael P. "Summing $P_n(\cos\theta)/p(n)$ for certain polynomials $p(n)$." *Computers Math. Applic.* **21** (10), pages 79–86, 1991.

Barnett, Michael P. "Implicit Rule Formation in Symbolic Computation." *Computers Math. Applic.* **26** (10), pages 35–50, 1993.

Barnett, Michael P. and Kevin R. Perry. "Hierarchical Addressing in Symbolic Computation." *Computers Math. Applic.*, in press, 1994.

Barnett, Michael P. "Symbolic calculations for the He Schrödinger equation." to be published, 1994.

Cajori, Florian. *A history of mathematical notation.* vol. II, page 81, Chicago: Open Court, 1929.

Chen, Zhan-zhan. *Symbolic calculation of inverse kinematics of robot manipulators.* M.A. Thesis, Brooklyn College of the City University of New York, 1991.

Curry, Halsey B. and R. Feys. *Combinatory Logic* vol. 1, Amsterdam: Netherlands, North-Holland Publishing, 1958.

Haas, Roswithwa and Kevin C. O'Kane. "Typesetting chemical structure formulas with the text formatter TEX/LATEX." *Computers and Chemistry,* **11** (4), pages 252–271, 1987.

Hindley, J.R. and J.P. Seldin. *Introduction to Combinators and Lambda-Calculus.* New York: Cambridge University Press, 1986.

Jensen, William B. "Crystal coordination formulas." Pages 105–146 in *Cohesion and structure*, vol. 2, *The structure of binary compounds*, D. G. Pettifor and F. R. de Boer, eds. North-Holland, Amsterdam, 1989.

Knuth, Donald E. *The TEXbook*. New York: Addison-Wesley, 1986.

Lamport, Leslie *LATEX — A Document Preparation System*. 2nd edition, New York: Addison-Wesley, 1994.

Pekeris, Chaim L. "Ground state of two-electron atoms." *Phys. Rev.* **112** (5), pages 1649–1658, 1958.

Ruffini, Paolo. *Teoria generale delle equazioni*. Bologna, Italy, 1799.

Spivak, Michael D. *The Joy of TEX*. Providence, RI: American Mathematical Society, 1986.

Taupin, Daniel. "MusicTEX: using TEX to write polyphonic or instrumental music." *TUGboat* **14** (3), pages 203–211, 1993.

Tutelaers, Piet. "A font and style for typesetting chess using LATEX or TEX." *TUGboat* **13** (1), pages 85–90, 1992.

Wolfram, Stephen. *Mathematica, A System for Doing Mathematics by Computer*. 2nd edition, New York: Addison-Wesley, 1991.

Concurrent Use of an Interactive TeX Previewer with an Emacs-type Editor

Minato Kawaguti and Norio Kitajima
Fukui University, Department of Information Science, 9-1, Bunkyo-3, Fukui, 910 Japan
kawaguti@i1mpsl.fuis.fukui-u.ac.jp

Abstract

A new efficient method was developed for editing (LA)TeX source files. It uses the combination of an Emacs-type editor and a special version of xdvi. Source files may be edited while browsing through the dvi preview screen simultaneously on the X window screen. Whenever a position is selected by clicking the mouse on a page of the document on display on the screen, the corresponding location of the particular (LA)TeX source file is shown in the editor's buffer window, ready for inspection or for alteration. One may also compile and preview (and obviously edit as well) any part of the entire document, typically one of its constituent files, for efficiency's sake. Fundamental characteristics of the document, shaped by the specification of the document style and various definitions found mostly at its root file, are retained even under partial compilation.

The Editor for TeX

Since it is not easy to grasp what a document looks like by simply reading the TeX source files, the efficiency of editing a TeX document file can be enhanced significantly if we can edit the TeX file in close coordination with the viewing capability of the corresponding TeX dvi file linked dynamically to the editor.

There can be two approaches for the realization of this scheme. The first method leads to developing a special editor which is capable of displaying a TeX-processed result. The second method respects the user's preference for a general-purpose editor, opting for its enhancement with the efficient viewing capability of the TeX dvi files in the X window screen.

The advantage of the former is that the designer of the editor has ample freedom to bring in the novel features desirable both for presenting the dvi view on the screen and for editing the TeX source being worked on. The VORTEX project by M. A. Harrison's group is a notable example adopting the first approach.

On the other hand, it may be equally advantageous for many people if they could use an editor with which they are familiar, provided it is equipped with an interactive dvi viewing feature. This paper describes a simple scheme of the second category targeted to those people who prefer Emacs or one of its derivatives as their sole editor for everything, including TeX sources.

This scheme of synchronizing an editor of the finest breed, of Emacs-type to be specific, with an acclaimed previewer will help improve, among others, editing sessions for large TeX documents. A typical document written in TeX, say a book manuscript, may consist of many files. These may form a tree structure through a multi-layered \input hierarchy, based on the logical divisions.

With the traditional editing style using the Emacs editor, particularly when the document consists of many files forming complex \input layers, a laborious cut-and-try search to single out a file from many is almost inevitable before locating the given passage. In any event, the cursor in the editing buffer window would have to be moved to all these places more or less manually.

In contrast, our scheme eliminates most of these time-consuming chores, and a single mouse click is all that is needed.

Outline of the Operation

In short, the editor/previewer combination does the following:

a. Any TeX source file or a chained cluster of them, be it the root, a node, or a leaf of the \input tree, can be previewed without compiling the entire TeX tree.
b. The number of generations of \input files to be included in a partial compilation for previewing can be limited to a user-specified depth, both in the direction of descendants and of ancestors.

In so doing, the fundamental characteristics of the document will still be preserved and reflected on the pages shown on the screen, even if its root file could have been curtailed.

c. Selection of the compiler, TEX, LATEX or LATEX2ε (LATEX3), is automatic.
d. The cursor of the Emacs-type editor jumps to the line of the TEX source file corresponding to the location specified by clicking the mouse on the display screen of the previewer.
e. The editor accepts interactive commands from the user while the previewer is active on the display. That is, both of them coexist, and there is no need to terminate the previewer to regain control over the editor.

Combination of Two Tools

A straightforward way to achieve this scheme is to select an editor and a previewer from among the tools most frequently used. The combination of the Emacs editor (or a close cousin) and `xdvi` would surely be acceptable to the majority of users, particularly those in the academic and scientific communities where we find a heavy concentration of devoted TEX users.

The present paper is based on our experience in implementing this scheme for two kinds of Emacs-type editors: the original Emacs editor, `GNU Emacs`, and one of its derivatives, `Njove`.

The latter, having been the subject of development for some years at Fukui University, is based on Jonathan Payne's `JOVE` (*Jonathan's Own Version of Emacs*). Amongst its many unique editing features not found in the original `JOVE` or in Emacs, `Njove`'s TEX mode is an attractive asset for editing (LA)TEX files. Like `JOVE`, `Njove` is written entirely in the C language.

Except for the ways the new editing commands are added to the main body of the respective editors, the two version are almost identical. For `GNU Emacs` this portion is written in Emacs lisp.

`Njove` has been the primary testbed for new ideas in this project because of the present authors' familiarity with its internal details. As such the `Njove` version is, at the time of this writing, in a slightly more advanced phase of development. Some of the minor implementation details (such as the coding method of inter-process communications) to be described in what follows may reflect, therefore, those of the `Njove` version. Nevertheless it is hoped that the word `Njove` can be read as indicating a generic Emacs-type editor, including `GNU Emacs` itself.

To ease portability, a substantial part of the program consists of modules that can be run as parallel Unix processes, isolated from the editor itself.

The Previewer

`Njove` permits previewing the whole or part of the file being edited using a modified version of the standard `xdvi`. (To distinguish it from the original version, the modified version will henceforth be referred to as `xdvi+`.) `Njove`'s text buffer window and the `xdvi+` TEX viewing window are shown side by side on the screen.

When `xdvi+` is activated, it displays the image of the specified `dvi` file on the X window screen. `xdvi+` scans the `dvi` file sequentially, and places each character glyph or rule on a page one by one, just as any `dvi` device driver does. Simultaneously with drawing each page, however, `xdvi+` keeps track of the locations it encounters by using `\special` commands that have all a valid argument string (*parsing message*) with the following format:

`loc` *source-file-name source-line-number*

A correspondence table is created anew for each update of the displayed page. The table records the correspondence between this locational information (the x- and y-coordinates) of the document page and that for the source file, namely the source file name and the line number. The table can accept a generous amount of `\special` commands (by default, up to 4096 entries per page).

Each time `xdvi+` detects a mouse event for the page and identifies it as the newly implemented `xdvi` instruction to locate the source file, `xdvi+` searches for the closest tagged location upstream in the document from the point the mouse click occurred. The source file name and the line number are identified by consulting the correspondence table for that tag entry.

Upon notification by `xdvi+` about this information, `Njove` switches the displayed content of the editing buffer promptly to that of the (possibly newly opened) target file, and moves its cursor (that is, "point" in Emacs jargon) to the beginning of the line which is most likely to contain the passage the user specified with the mouse on the TEX preview screen. Incidentally, the buffer is ready to accept any editing command all the time.

The coordination between `Njove` (or its "agent", `texjump`, to be more rigorous, as will be discussed in a moment) and `xdvi+` can be outlined as follows:

1. Establish a link between Njove and xdvi+, so that they can communicate with each other in real-time in a typical X-window environment.
2. Let xdvi+ pick up the positional information where the mouse click event took place.
3. Interpret the click position and notify Njove of:
   a. the source file name, and
   b. the source line number.
4. Let Njove "find" the specified file, and position the cursor at the beginning of the designated line.

A New Editor Command

To integrate the interactive previewing capability of TeX's dvi file into the editor, a new command tex-jump was added to Njove.

When the Njove command

Esc-x tex-jump [*option switches*] [*target-file*]

is issued, Njove spawns a separate Unix process texjump, independently of the editor. (The presence or absence of the hyphen in the name texjump is used to differentiate between these two closely related but clearly distinct entities.) If the file name is not specified, the file associated with the buffer of the window, from which the command was issued, is selected as the default *target-file* (to be described later). Optional switches may also be specified. These are identical to the ones for texjump as a shell-executable command.

The standard I/O of texjump is connected to Njove via a pair of *pty*s, and its output stream is eventually sent to and stored in the newly created Njove buffer named "*texjump*".

texjump in turn spawns xdvi+. They communicate with each other through a Unix pipe. For each mouse click in the preview screen, xdvi+ sends back to texjump the locational information of the source file through the Unix pipe. texjump thereupon outputs a grep-like message (*parsing message line*) to the standard error stream, which Njove accepts through its *pty*.

Until texjump is eventually terminated, Njove intercepts all the input streams to its various buffers scrutinizing a stream destined to the buffer *texjump*. If a parsing message for texjump is found, Njove subsequently lets its newly added function ParseErrorOneLine() parse that single line, and displays the pertinent buffer (or opens a new file if it is not yet assigned to any of the existing buffers) in an appropriate working window, and lets its cursor (point) move to the beginning of the line specified by xdvi+ ("*point positioning*"). At the same time, the successive parsing message line is appended to a special buffer "*texjump*".

Positioning on the Screen

Whenever the left button of the mouse is clicked while holding down the control key of the keyboard at the same time, xdvi+ determines the corresponding current location in the source file, and transmits it to texjump. Njove, receiving this information from texjump through *pty*, selects the relevant buffer and advances the point to the beginning of the requested line.

Users user can pick any location at any time asynchronously until they quit xdvi+ with the q command.

When the command tex-jump is issued, Njove switches to the active state of "*error parsing*". Then Njove is ready to accept parsing commands from the keyboard. They are next-error (*C-x C-n*) and previous-error (*C-x C-p*), respectively, which step the point in the buffer *texjump* either one parsing message line downward or upward, followed by a new *point positioning*. (In the case of GNU Emacs, next-error is key-bound to *C-x* ', while previous-error is missing.) This active status persists even after xdvi+ is terminated through its *quit* command. Issuing *C-x C-c* finally lets Njove exit from its error parsing status.

The Tree Structure of TeX Source Files

A typical TeX document may be composed of multiple TeX source files forming a tree structure by means of the \input feature. Let its root file be *root*.tex.

A new tool, textree, analyzes the tree structure of the document by tracing recursively the existence of \input or \include commands. textree expects a single argument in the command line, the root of the document tree.

% textree *root*.tex

textree generates a file, Tex_Input_Tree by default, which indicates the mutual input dependency relationship of the document in a format akin to what the Unix make command understands. Therefore, as a byproduct, the created file, Tex_Input_Tree, may also be used to write the dependency rule of a Makefile for all sorts of (LA)TeX compilation in general.

Derivation of the Source Position from the DVI File

Since the TeX compiler does not leave any trace of the locational information about the original TeX source files in the dvi file, ordinary dvi device drivers have no way of correlating an arbitrarily chosen point on a processed page of the document with the specific file among a number of TeX source files forming that document, and the word/line position within that file in particular.

Therefore some means of forwarding the locational information to the device driver has to be incorporated. There are two alternatives:

The most straightforward scheme would be to modify the (LA)TeX compilers such that they either

- include the locational information of the source files within the dvi file they generate, or
- generate an additional auxiliary file which contains the information about the location in the document pages of the items found at the beginning of all the source lines.

One could envisage introducing a parallel to the optional switch -g found in the C compiler, used to add extra information for source level debugging. While there is no doubt as to the technical feasibility of this scheme, and obviously it is the most rational and robust of the two alternatives, in real life the modification had better be incorporated into the official circulating version of all the compilers by their original authors, lest the introduction of yet other variants go astray from the spirit of unification of (LA)TeX.

Although we would very much like to have this feature in future releases of the (LA)TeX compilers, we will look for another alternative that offers a practical solution for the time being. This approach uses the (LA)TeX compilers as-is, without any modification. It generates copies of the source files, and additional information (a "positional tag") is inserted into these files automatically prior to the (LA)TeX compilation. The positional tag is inserted at every "landmark location" of the source files, say at every location where a new paragraph begins ("paragraph mode"). Or it could as well be at the beginning of each non-empty source line ("line mode").

The applicable positional tags must never distort the original content of the document. Two TeX commands, \wlog and \special, satisfy this criterion.

One can insert a \special command with its message text consisting of:

- a unique ID code (default: loc) to distinguish this particular usage of the \special command from others;
- the source file name;
- the source line number.

This is the scheme adopted in texjump.

By comparison, one could insert a \wlog command, instead of \special, as the positional tag. The preprocessor (that is, the equivalent of texjump) would then generate the message text for \wlog as an ASCII string indicating the location as the line number of the source file at the point it inserts the \wlog command. With the help of a simple program that would analyze the log file written by the the (LA)TeX compiler, the page boundaries of the printed document could be identified in the source files.

The advantage, if any, of using \wlog would be that neither the (LA)TeX compilers nor the previewer need to be altered, offering the user a much wider selection of previewers. This benefit would, however, be offset in most cases by the drawbacks, in comparison with using \special:

- The editor can control the previewer page, but not vice versa, because the unmodified version of the previewer cannot communicate back to the editor.
- The positioning resolution one can expect cannot go beyond the page of the document displayed on the previewer screen.

Line Mode versus Paragraph Mode

texjump accepts two options to select the way \special commands are inserted, namely line mode and paragraph mode. When line mode is chosen, a mouse click in the previewer window can locate the source position within the range of a line or so. In paragraph mode, however, we deal with a scope no finer than the size of the paragraphs involved. The main motivation for paragraph mode comes from the need to make texjump much more robust if line mode fails for reasons discussed below.

Line mode. In this mode \special is inserted at the beginning of each non-empty line without otherwise altering the original context of each line. Since the original line number assigned to each line remains valid after the insertion, the dvi driver can identify the correct line number in the original source files, even though it extracts the data from a single file, namely the dvi file created by compiling the modified copy files containing the scattered \special commands.

Paragraph mode. In this mode \special is inserted exclusively at the beginning of the first line of each "paragraph". texjump recognizes a cluster of one or more empty lines as the paragraph delimiter. (Note that the definition of a paragraph is different from that of TeX or Emacs.)

Problems Associated with Tag Insertion

Even though a \special command supposedly causes no appreciable side effect other than merely forwarding a character string to the dvi driver as a communicative message, it does not mean we can insert it indiscriminately in any arbitrary position of the given source file.

As a typical example, consider the case of a TeX macro which expects one or more arguments, and there occurs a line break in the source file just in front of one of its arguments. One cannot insert the \special blindly at the beginning of the following line which starts with the expected argument.

For instance, within a \halign construct, the line with \noalign rejects \special. If the construct's final line begins with its outermost closing brace (}), \special is not permitted.

A more obvious example is LaTeX's verbatim environment, or its TeX equivalent. Insertion of a \special in the lines belonging to this environment does alter the content of the compiled document because there \special is nothing more than a plain character string. Needless to say, xdvi+ does not identify the "argument" as positional information.

Therefore texjump has to know about lines where \special insertion should be avoided. This means that texjump must be able to, ideally speaking, analyze the syntactical structures.

Realization of a full scope syntax analysis would be equivalent to almost fabricating a new (LA)TeX compiler. This kind of duplicated effort would not be justifiable, because the modification of the compilers mentioned before is clearly the rational way to do it. The current version of texjump analyzes, therefore, the syntactical structure of the source files only superficially.

If the (LA)TeX compiler complains about a syntactic error that originated from the insertion of the \special, the user may either switch to paragraph mode, which is more robust than line mode, or modify slightly the original source file, as will be discussed below, by adding some directives to texjump in the form of comment lines for the TeX compiler.

Where to Attach the Positional Tags

Since (LA)TeX refuses to accept the insertion of a positional tag at certain places, we have to discern these syntactically inappropriate circumstances. The current version of texjump interprets the syntactical structure superficially. Therefore it recognizes only the most obvious cases.

Tags are not attached to the beginning of the following lines:

1. a blank line, or a comment line;
2. within a verbatim environment;
3. from the line beginning with \def till the following blank line;
4. a line which begins with }, \noalign, \omit, \multispan;
5. the line following a non-blank line ending with %;
6. the preamble and postamble part of each file, if any;
7. lines for which explicit instructions tell texjump not to attach a tag;
8. each non-first line of each paragraph when in paragraph mode.

Otherwise the positional tag is attached to the very beginning of each line.

Manual Control of the Tag Insertion

texjump's algorithm for inserting positional tags works reasonably well for relatively simple TeX documents. For documents of a complex nature, however, one can only expect it to be marginally smart.

When texjump stumbles into a pitfall, particularly in line mode, some texjump directives can rescue it. Inserted manually in the source file by the user, they let texjump avoid potential hazardous spots in the file.

Each directive is a TeX comment line with a predefined format. It consists of a line beginning with three % characters followed by a symbol.

%%%<	Enter paragraph mode.
%%%>	Exit paragraph mode.
%%%!	Skip tagging the ensuing single line.
%%%~	Skip tagging until the next blank line.

File Inclusion

texjump lets the user specify the range of files to be included through three parameters:

1. the filename under consideration (*target-file*);
2. the number of generations, in the \input tree, corresponding to:

a. its ancestors from there upstream (*ans*);
b. its descendants from there downstream (*des*).

The default is *ans* = 0 and *des* = ∞; that is, the *target-file* and all of the files it includes in a cascade downstream.

texjump first looks for the file Tex_Input_Tree in the current working directory, and obtains from it the tree path which reaches the root file from *target-file*. If Tex_Input_Tree is missing, *target-file* is assumed to be the root file.

As for ancestors, inclusion is limited to only those files directly on that path. No siblings of the *target-file* or of its ancestors are included. If *ans* is smaller than the generation number to the root file, some of the files closer to the root, including the root itself, will be out of range for the file inclusion scope. texjump inspects the content of each of these files, and if any of them contains the preamble and/or postamble, these portions (not the entire file) are all extracted for inclusion despite the scope rule.

The file inclusion rule for the descendants is much simpler. If *des* is specified as the option parameter to the texjump command, up to *des* generations of direct descendants of *target-file* are included. Otherwise, all of its descendants are included.

texjump suppresses the file inclusion simply by altering the string \input or \include to

```
\par\vrule width 2em height 1ex
            \qquad{\tt \string\input}\quad
```

in the same line, which generates a line like

■ \input *input-file-name*

on the preview screen, thus making it clear that the \input command line is there.

Preambles and Postambles

texjump assumes that each file consists of three parts:

1. an (optional) preamble;
2. the main body;
3. an (optional) postamble.

The preamble, if any, is an arbitrary number of lines at the beginning of the file bounded by two lines with the unique signatures:

```
%**beginning_of_header
```

and

```
%**end_of_header
```

Likewise the postamble might be at the end of the file, bounded similarly by the lines:

```
%**beginning_of_tailer
```

and

```
%**end_of_tailer
```

The root file is exceptional in that both explicit and implicit definitions of both the preamble and postamble are permitted. For those files which do not have the above-mentioned signature line for preamble initiation, if the line \begin{document} is encountered within the first 100 lines then the region from the first line to this implicit preamble terminator line is treated as the preamble. The same is true with the postamble. The implicit postamble in most cases is from a line which contains

```
\end{document}
\bye
```

or

```
\end
```

till the very end of the file.

texjump inserts positional tags neither in the preamble nor in the postamble. Therefore any TeX codes, critical for the document but irrelevant for positioning, should be placed inside these regions. TeX macro definitions, variable parameters setting, or inclusion of system files are typical examples.

It should be noted that both preambles and postambles of all the files involved in the TeX tree are always included, irrespective of the scope rule.

Source Recompilation

Since texjump keeps showing the very same dvi file, and therefore the recent modifications are not reflected on the viewing screen, updating the screen may become desirable after some modification of the source files. Taking into account the time (LA)TeX takes to compile, however, it may hamper efficient editing work if we let texjump decide to initiate automatically the recompilation of the latest source files over and over again even at sporadic intervals. Therefore, unless the user instructs xdvi+ to do so explicitly, recompilation does not take place.

Sending a C character to the xdvi+ window signals it to perform a recompilation. xdvi+ conveys to texjump the acceptance of the user's request and waits for the renewed dvi file. When available, it redraws the screen using the new dvi file.

Intermediate Files

Since our scheme modifies the content of the (LA)TeX source files, we must work on the copied files. Therefore texjump first creates a new (temporary) working directory with the user-specified path-name. It then reproduces there the entire

directory structure of all the files involved, taking into account the file inclusion rule.

texjump identifies the "local root" for those files which fall outside of the clusters stemming from the original current working directory. texjump assigns to each of them a subdirectory in the above-mentioned working directory and gives it an arbitrary name. texjump keeps the entire record of the file mapping between original and copy.

In this manner, the intermediate files are effectively hidden from the user, thus creating the impression that one is dealing directly with the original files. In reality, what the previewer is showing on the screen, and giving the positional information for, corresponds to the copied files, while what the editor is showing in its window are the genuine original files.

In order to take this hiding process a step further, even the (LA)TEX compiler is manipulated by texjump on purpose. When the (LA)TEX compiler detects an error in a source file, the user usually calls for the editor by responding with an "e" character. The compiler then transfers control to the user-specified editor. It instructs the editor to open the temporary file, because this is where *it* found the error. texjump, however, swaps the shell's environmental variable $TEXEDIT temporarily with a fake editor, texjump_ed, just before (LA)TEX starts compilation of the tagged files. Therefore it is texjump_ed which receives information about the file (path-name) and the line number. Its sole role is to identify the original source file from the received information, and then to call in the real editor the user had requested, acting as if (LA)TEX had performed that job.

In order to enforce integrity we *minimize* the possibility of confusing the original and the copies by deleting the temporary subdirectory created by texjump each time xdvi+ is relinquished after previewing.

Option Switches

The Unix shell can execute texjump as a stand-alone process. It expects a (LA)TEX source file name, and optional switches may also be specified.

% texjump [-*opt* [*num*] [, ...]] *target-file*

If *target-file* does not specify a filename extension, texjump assumes it to be .tex.

Valid option switches -*opt* are:

-h Displays the entire list of switch options.

-p Instructs texjump to treat all files in paragraph mode.

-n *num* Lets the (LA)TEX compiler repeat the compilation *num* times (default is 1).

-t *num* Start from the ancestor *num* generations upstream in the input tree. *num* = 1 specifies that the parent immediately above the source file (*target-file*) should be included (default is 0, i.e., no ancestor is included).

-b *num* Include the \input files down to *num* generations of descendants. *num* = 1 means only the "child" files, included directly through an \input command in the source file (*target-file*), are to be included.

Generation of xdvi+

The source files for xdvi+, the extended version of xdvi, are generated through applying a patch to version 17 of xdvi. It modifies five files, Imakefile, dvi_draw.c, tpic.c, xdvi.c, and xdvi.h, and adds a new module, jump.c.

imake generates a Makefile, which takes care of the entire process of creating xdvi+. Note that xdvi+ preserves all features of the original xdvi.

Customization

Users can specify some of the critical parameters controlling texjump. They are described in a configuration file, whose default name is .texjumpcfg (this can be altered at installation time). texjump looks for this file successively in the current directory, then in the user's home directory, and finally it uses the parameters in the system default file. It specifies to texjump the choice of:

1. the temporary working directory to be created (and removed subsequently);
2. the previewer;
3. the name of the \input tree file generated by textree;
4. the (LA)TEX compiler;
5. the signatures signaling the end of the preamble part;
6. the signatures signaling the beginning of the postamble part.

An example of the default .texjumpcfg parameters is shown below:

Parameter	*Default*
TEXINPUTS	.:/import/TeX/inputs//\ :/import/TeX/lib//
WORKDIR	texjump-workdir
XDVI	xdvi+
XDVI_OPTION	-s 3

TREE	TeX_Input_Tree
TAG	loc ${FILE} ${LINE}
TEX	/import/TeX/bin/tex
LATEX	/import/TeX/bin/latex
LATEX2E	/import/TeX/bin/latex2e
FOILTEX	/import/TeX/bin/foiltex
BOH	%**beginning_of_header
EOH	%**end_of_header
BOT	%**beginning_of_tailer
EOT	%**end_of_tailer
BOT_TEX	\bye
	\end
EOH_LATEX	\begin{document}
BOT_LATEX	\end{document}
BEGIN_PAR_MODE	%%%<
END_PAR_MODE	%%%>
EXCLUDE_NEXT	%%%!
EXCLUDE_BLOCK	%%%~
SEL_TEX	%**plain_tex
	%**tex
	\input eplain
SEL_FOILTEX	\documentstyle{foils}
	%**foiltex
SEL_LATEX	%**latex
	\documentstyle
SEL_LATEX2E	%**latex2e
	\documentclass

Conclusions

(LA)TEX source files can be edited using an Emacs-type editor, say GNU Emacs. By clicking the mouse on an arbitrary page of the xdvi preview screen, the cursor (point) moves directly to the interesting spot in the Emacs window, that is displayed next to the xdvi window.

Two prototype versions, for Njove and GNU Emacs, are currently operational on workstations running the 4.3BSD and SunOS operating systems. Porting the software to other Unix platforms is expected to be straightforward. texjump assumes that the presence of the standard GNU development environment on the target machine. The program, written in the C language, can be compiled with GNU gcc.

The program will be available through anonymous ftp at ilnwsl.fuis.fukui-u.ac.jp in the directories texjump, xdvi+ and textree under /pub/tex/.

Njove is a bilingual editor, which supports both English (using the single-byte ASCII character code set) and Japanese (using the two-byte Japanese character code set). It can be found in the directory /pub/editor/njove at the same ftp site.

Acknowledgments

The authors thank Takayuki Kato for his contribution in making texjump worthy of real-world applications through improving its functionalities. Jun-ichi Takagi wrote the first rudimentary interface module for GNU Emacs using Emacs lisp.

Bibliography

Harrison, Michael A., "News from the VORTEX Project", *TUGboat* **10**(1), pages 11-14, 1989.

Cooper, Eric, Robert Scheifler, and Mark Eichin, "xdvi" on-line manual, June, 1993.

Payne, Jonathan, *JOVE Manual for UNIX Users*, 1986.

Kawaguti, Minato, "Dynamic Filling with an Emacs Type Editor", *Proc.* **jus** *10th Anniversary Intern. UNIX Symp.*, Japan UNIX Soc., pages 49-58, 1992.

Indica, an Indic preprocessor for TEX
A Sinhalese TEX System

Yannis Haralambous
Centre d'Études et de Recherche sur le Traitement Automatique des Langues
Institut National des Langues et Civilisations Orientales, Paris.
Private address: 187, rue Nationale, 59800 Lille, France.
Yannis.Haralambous@univ-lille1.fr

Abstract

In this paper a two-fold project is described: the first part is a generalized preprocessor for Indic scripts (scripts of languages currently spoken in India—except Urdu—, Sanskrit and Tibetan), with several kinds of input (LATEX commands, 7-bit ASCII, CSX, ISO 10646/UNICODE) and TEX output. This utility is written in standard Flex (the GNU version of Lex), and hence can be painlessly compiled on any platform. The same input methods are used for all Indic languages, so that the user does not need to memorize different conventions and commands for each one of them. Moreover, the switch from one language to another can be done by use of user-defineable preprocessor directives.

The second part is a complete TEX typesetting system for Sinhalese. The design of the fonts is described, and METAFONT-related features, such as metaness and optical correction, are discussed.

At the end of the paper, the reader can find tables showing the different input methods for the four Indic scripts currently implemented in *Indica*: Devanagari, Tamil, Malayalam, Sinhalese. The author hopes to complete the implementation of Indic languages into *Indica* soon; the results will appear in a forthcoming paper.

This paper will be published in the next issue of *TUGboat*.

Pascal pretty-printing: an example of "preprocessing within TEX"

Jean-luc Doumont
JL Consulting, Watertorenlaan 28, B-1930 Zaventem, Belgium
j.doumont@ieee.org

Abstract

Pretty-printing a piece of Pascal code with TEX is often done via an external preprocessor. Actually, the job can be done entirely in TEX; this paper introduces PPP, a Pascal pretty-printer environment that allows you to typeset Pascal code by simply typing `\Pascal` ⟨*Pascal code*⟩ `\endPascal`. The same approach of "preprocessing within TEX" — namely two-token tail-recursion around a `\FIND`-like macro — can be applied easily and successfully to numerous other situations.

Introduction

A pretty-printed piece of computer code is a striking example of how the typeset form can reveal the contents of a document. Because the contents are rigorously structured, an equally rigorous typeset form helps the reader understand the logic behind the code, recognize constructs that are similar, and differentiate those that are not. Not surprisingly, many programming environments nowadays provide programmers with a pretty-printed representation of the code they are working on. In the typesetting world, TEX seems an obvious candidate for a pretty-printing environment, thanks to its programming capabilities and its focus on logical — rather than visual — design.

The current standard for typesetting Pascal code with TEX seems to be TGRIND, a preprocessor running under UNIX. Useful as it may be, TGRIND also has limitations. While it can recognize reserved words, it does little to reflect logical content with indentation. In fact, it indents by replacing spaces in the original file by fixed `\hskip`'s. Of course, it can be used on the result produced by an ASCII-oriented pretty-printer, which generates the right number of spaces according to logical contents.

Alternatives to TGRIND are either to develop a dedicated preprocessor — a computer program that takes a piece of Pascal code as input and produces a TEX source file as output — or to do the equivalent of the preprocessing work within TEX. The first solution is likely to be faster, hence more convenient for long listings, but requires an intermediate step and is less portable. The second, by contrast, is rather slow, but also quite convenient: pieces of Pascal code can be inserted (`\input`) *as is* in a TEX document, or written directly within TEX. This solution is portable (it can run wherever TEX runs), requires no intermediate step (it does its job whenever the document is typeset), and, like other sets of macros, can be fine-tuned or customized to personal preferences while maintaining good logical design.

This article describes briefly the main features and underlying principles of PPP, a Pascal pretty-printing environment that was developed for typesetting (short) pieces of Pascal code in engineering textbooks. It then discusses how to use the same principles of "preprocessing within TEX" to quickly build other sets of macros that gobble up characters and replace them with other tokens, to be further processed by TEX. The complete PPP macro package will soon be found on the CTAN archives.

Of course, there are other ways of tackling the issue, with either a broader or a narrower scope. Structured software documentation at large can benefit from the literate programming approach and corresponding tools, with TEX or LATEX as a formatter — a discussion beyond the scope of this paper. Occasional short pieces of code, on the other hand, can also be typeset verbatim or with a few *ad hoc* macros, for example a simple tabbing environment, as shown by Don Knuth (1984, page 234). For additional references, see also the compilation work of Piet van Oostrum (1991).

Main features of macros

Basic use. PPP works transparently; you do not need to know much to run it. After `\input`ing the macros in your source, all you do is write

```
\Pascal
⟨Pascal code⟩
\endPascal
```

in `plain.tex` or

```
\begin{Pascal}
⟨Pascal code⟩
\end{Pascal}
```

in LaTeX, where ⟨*Pascal code*⟩ can be an `\input` command.

The PPP package then pretty-prints the corresponding Pascal code; by default, it

- typesets reserved words in boldface;
- indents the structure according to syntax (identifying such constructs as **begin** ... **end** and **if** ... **then** ... **else** ...;);
- typesets string literals in monospaced (`\tt`) font;
- considers comments to be TeX code and typesets them accordingly.

The Appendix illustrates these features.

Comments. Recognizing comments as TeX code is particularly powerful: side by side with a rather strict typeset design for the program itself, comments can be typeset with all of TeX's flexibility and power. Besides for adding explanatory comments to the program, this possibility can be used to fine-tune the layout. Extra vertical space and page breaks can be added in this way. Such comments can even be made "invisible", so no empty pair of comment delimiters shows on the page.

Accessing TeX within comments suffers a notable exception, though. Pascal comments can be delimited with braces, but Pascal compilers *do not match braces*: the first opening brace opens the comment and the first closing brace closes the comment, irrespective of how many other opening braces are in between. As a consequence, braces cannot be used for delimiting TeX groups inside Pascal comments (the result would not be legal Pascal code anymore). Other TeX delimiters must be used; by default, PPP uses the square brackets '`[`' and '`]`'.

Program fragments. PPP was taught the minimum amount of Pascal syntax that allows it to typeset Pascal code; it is thus not a syntax-checker. While some syntax errors (such as a missing **end**) will cause incorrect or unexpected output, some others (such as unbalanced parentheses) will be happily ignored.

However, the package was designed for inserting illustrative pieces of code in textbooks, including *incomplete* programs. PPP has facilities for handling these, though it needs hints from the author as to what parts are missing. These hints basically consist in supplying — in a hidden form — the important missing elements, so PPP knows how many groups to open and can then close them properly.

Customization. PPP is dedicated to Pascal. Though you can use the same underlying principles (see next section) in other contexts, you cannot easily modify PPP to pretty-print very different programming languages. There is, however, room for customizing the pretty-printing, and this at several levels.

At a high level, you can use the token registers `\everyPascal`, `\everystring`, as well as `\everycomment` to add formatting commands to be applied, respectively, to the entire Pascal code, to the Pascal string literals, and to the Pascal comments. If you want your whole Pascal code to be in nine-point roman, for example, you can say

```
\everyPascal{\ninerm
             \baselineskip=10pt⟨etc.⟩}
```

If you would rather use '`(`' and '`)`' instead of '`[`' and '`]`' as TeX grouping delimiters in Pascal comments, you can say

```
\everycomment{\catcode'\(=1 \catcode'\)=2}
```

Similarly, if you wish to reproduce the comments verbatim rather than consider them as TeX code, you can say

```
\everycomment{\verbatimcomments}
```

At an intermediate level, you can add reserved words by defining a macro with the same name as the reserved word prefixed with `p@`. If you want the Pascal identifier `foo` to be displayed in italics in your code, you can say

```
\def\p@foo{{\it foo}}
```

before your code and PPP will do the rest.

At a low level, you can go and change anything you want, providing you know what you are doing *and* you first save PPP under a different name.

Underlying principles

The PPP environment pretty-prints the code in one pass: it reads the tokens, recognizes reserved words and constructs, and typesets the code accordingly, indenting the commands according to depth of grouping. Specifically, PPP

- relies on tail-recursion to read a list of tokens: one main command reads one or several tokens, processes them, then calls itself again to read and process subsequent tokens until it encounters a stop token;
- decides what to do for each token using a modified version of Jonathan Fine's `\FIND` macro;
- recognizes words as reserved by checking for the existence of a TeX command with the corresponding name and acts upon reserved words by executing this command;

- typesets the code by building a nested group structure in TeX that matches the group structure in Pascal.

Tail-recursion. Jonathan Fine (1993) offers useful control macros for reading and modifying a string of tokens. Rewritten with a ';' instead of a '*' (to follow the Pascal syntax for a **case**), his example for marking up vowels in boldface, a problem introduced in *Einführung in TeX* by Norbert Schwarz (1987), becomes:

```
\def\markvowels #1
{
  \FIND #1
    \end:;
    aeiou AEIOU:{\bf#1}\markvowels;
    #1:{#1}\markvowels;
  \END
}
```

so that `\markvowels Audacious \end` produces "**Au**d**a**c**iou**s". `\FIND` is a *variable delimiter macro* (as Fine puts it), defined as

```
\long\def\FIND #1{%
  \long\def\next##1#1##2:##3;##4\END{##3}%
  \next}
```

It extracts what is between the ':' and the ';' immediately following the first visible instance of #1 and discards whatever is before and whatever is after (up to the following \END). The same idea is used in the *Dirty Tricks* section of the *The TeXbook* (Knuth 1984, page 375). The generic use of \FIND is thus

```
\FIND ⟨search token⟩
  ⟨key⟩⟨key⟩...⟨key⟩:⟨action⟩;
  ⟨key⟩⟨key⟩...⟨key⟩:⟨action⟩;
  ...
  ⟨key⟩⟨key⟩...⟨key⟩:⟨action⟩;
  ⟨search token⟩:⟨default action⟩;
\END
```

PPP brings the following three basic changes to Fine's scheme:

- first, it uses a tail-recursion scheme that reads tokens two by two rather than one by one; this extension makes it easier to recognize and treat character pairs such as '>=', '..', and '(*'.
- next, it moves the tail-recursion command (the equivalent of `\markvowels` in the example above) to the end of the macro, to avoid having to repeat it for each entry in the \FIND list. This move also simplifies brace worries: whatever is specified between the ':' and the ';' in the above definition can now be enclosed in braces. These protect a potential #1 in the ⟨*action*⟩ (they make it invisible when `\next` scans its argument list), but do not produce an extra level of grouping (they are stripped off when `\next` reads its argument #3).
- finally, it replaces ':' and ';'—which need to be recognized explicitly when reading the Pascal code—respectively by '?' and '!'—which do not. (Other tricks are possible; see for example Sections 4 and 6 in Fine (1993).)

To consider all pairs of tokens, the new scheme spits out the second token before calling the recursive command again, so this second token is read as the first token of the new pair. While this double-token system has proved very convenient in many applications I developed, it has one inherent limitation: because the spit-out character has been into TeX's mouth, it has already been tokenized (assigned a character code). If the action corresponding to the first token read is to redefine character codes, then the second token will not reflect these new codes. When such a recoding is an issue, alternative constructs using `\futurelet` can be devised to consider pairs (i.e., to take the next token into account in deciding what to do), but such constructs are rather heavy.

With these changes, the tail-recursion core of the Pascal pretty-printer looks something like this:

```
\long\def\Find #1{
  \long\def\next##1#1##2?##3!##4\END{##3}
  \next}

\def\Pascal{\pascal \relax}
  % \relax is passed as first token
  % in case the code is empty
  % i.e., the next token is \endPascal

\def\pascal#1#2{\def\thepascal{\pascal}%
  \Find #1
    ⟨key⟩⟨key⟩...⟨key⟩?{⟨action⟩}!
    ⟨key⟩⟨key⟩...⟨key⟩?{⟨action⟩}!
    ...
    ⟨key⟩⟨key⟩...⟨key⟩?{⟨action⟩}!
    #1?{⟨default action⟩}!
  \END
  \ifx\endPascal#2
    \def\thepascal##1{\relax}\fi
  \thepascal#2}
```

with the typesetting taking the form

```
\Pascal ⟨Pascal code⟩ \endPascal
```

In this two-token scheme, the end-of-sequence test must now be done on the second token read, so the tail recursion does not read past the end-of-sequence token (`\endPascal`). The sequence is ended by redefining `\thepascal` to gobble the next token and do nothing else.

Hmm ... it is a little more complicated than that. The `\pascal` macro (which is really called `\p@sc@l`) must be able to recognize and act upon

braces, used as comment delimiters in Pascal. These braces are recatcoded to the category `other` by saying `\catcode'\{=12 \catcode'\}=12` somewhere in `\Pascal`, so they lose their grouping power when TeX scans Pascal code. Because the `\FIND` macro identifies *tokens*, category codes must match. In other words, '{' and '}' must be of category 12 when `\p@sc@l` is defined, so we must use another pair of characters as group delimiters for defining `\p@sc@l`. I use the square brackets '`[`' and '`]`'.

Accumulating words. Identifiers in Pascal are composed of letters, digits, and the underscore character '_', but must start with a letter. Correspondingly, PPP identifies words in the following way. It uses an `\ifword` switch to indicate whether a word is currently constructed and an `\ifreserved` switch to indicate whether the accumulated word is a candidate reserved word. Starting on a situation in which `\ifword` is false, it does the following:

- if the token read is a letter, set `\ifword` and `\ifreserved` to true, empty the token register `\word`, and accumulate the letter in it.
- if the token read is a digit, look at `\ifword`. If true, accumulate the digit in the token register `\word` and set `\ifreserved` to false (reserved words contain no digit); if false, treat as a number.
- if the token read is an underscore, look at `\ifword`. If true, accumulate the underscore in the token register `\word` and set `\ifreserved` to false (reserved words contain no underscore); if false, treat as an underscore.
- if the token is not a letter, a digit, or an underscore, look at `\ifword`. If true, set to false and take care of the word so terminated. If false, pass token to other macro for further processing.

Recognizing reserved words. PPP recognizes reserved words by checking words composed of letters only against a list. This list is in reality a set of macros, the names of which are formed by prefixing Pascal reserved words with 'p@'. These macros have thus a double role:

- by their existence, they identify a word as reserved; for example, the existence of a macro named `p@begin` indicates that **begin** is a reserved word.
- by their definition, they tell what to do when the corresponding reserved word has been identified; for example, `\p@begin` takes care of what needs to be done when the reserved word **begin** is encountered.

The actions to perform when a reserved word has been identified depend of course on the word, but are within a small set, namely

- typesetting the word as a reserved word, possibly with space before or after;
- opening a group and increasing the indentation;
- closing a group, thus going back to the level of indentation present when that group was opened; or
- turning flags on or off.

Because many reserved words require the same action, the corresponding TeX macros can all be `\let` equal to the same generic macro. For example, `\r@serv` simply typesets the last reserved word accumulated (without extra space), so reserved words like **string** or **nil** can be taken care of simply by saying

```
\let\p@string=\r@serv
\let\p@nil=\r@serv
```

Grouping and indenting. PPP manages the levels of indentation by creating a nested group structure that matches the structure of the program. A **begin**, for example, opens a group and increments the indentation by one unit within the group; an **end** closes the group, thus returning to the level of indentation in effect before the group was opened.

Of course, grouping is not always that simple. All the declarations that follow a **var**, for example, should be within an indented group, but there is no reserved word to mark the end of the group. Such cases are treated by setting a flag to true, to indicate that a group without terminator is open. The next of a subset of reserved words can then close that group before performing its own task.

Other examples of "preprocessing"

A tail-recursion engine based on a `\FIND`-like macro does pretty much what one would expect a preprocessor to do: it gobbles the characters one by one and replaces them with other, possibly very different tokens. This similarity is what leads me to refer to such a scheme as "preprocessing within TeX" (though, strictly speaking, this is a contradiction in terms).

The one-token examples presented in Fine (1993) are the simplest case of this preprocessing: decisions are taken each time on the basis of a single token. Such a scheme is simple, straightforward, and sufficient in many applications. And when following tokens must be taken into account, it can be extended with `\futurelet` constructs, though these quickly become quite heavy. For

example, the \markvowels macro can be modified in the following way to mark, say, "i before e" combinations:

```
\def\spellcheck#1{%
  \FIND #1
    \end:;
    i:\ie;
    #1:{#1}\spellcheck;
  \END}

\def\ie{\futurelet\nextchar\iiee}
\def\iiee{\let\etemp=e%
  \ifx\etemp\nextchar
    {\bf ie}%
    \let\temp\gobblenexttoken
  \else
    i%
    \let\temp\spellcheck
  \fi
  \temp}

\def\gobblenexttoken#1{\spellcheck}
```

so that typing

```
{\obeyspaces
\spellcheck  I receive a piece of pie\end}
```

yields "I receive a p**ie**ce of p**ie**".

The two-token example presented in this paper is a convenient extension of the scheme. True, it has as inherent limitation that the second parameter is tokenized (assigned a character code) one step earlier than it would in the one-token case. On the other hand, the corresponding code is particularly readable (thus easy to program and easy to maintain). The above example becomes, with a two-token model,

```
\def\check#1#2{\def\nextcheck{\check}%
  \FIND #1
    i:{\FIND #2
         e:{{\bf ie}\gobbleone};
         #2:{i};
       \END};
    #1:{#1};
  \END
  \ifx\end#2
    \def\nextcheck##1{\relax}\fi
  \nextcheck#2}

\def\gobbleone{\def\nextcheck##1%
  {\check \relax}}
\def\spellcheck{\check \relax}
```

where \gobbleone gobbles the next token and replaces it with \relax. The nested \FIND structure makes it easy to see the underlying idea of "once you know the first letter is an i, see whether the second is an e". Clearly, the mechanism can be extended to take into account three, four, or even more tokens at the same time, with limitations and advantages similar to those in the two-token case.

Two-token tail-recursion can also be achieved with other constructs, for example Kees van der Laan's \fifo macro. In van der Laan (1993) he underlines the importance of the *separation of concerns*: going through the list is separated from processing each element of the list. This elegant programming principle is sometimes hard to achieve in practice: in the case of string literals, for example, \Pascal reacts to a single quote by interrupting token-by-token progression and reading all tokens to the next single quote — progressing and processing are thus closely linked. For the "i before e" example, the separation is clearer and the use of the \FIND structure for processing the elements is largely unchanged:

```
\def\fifo#1#2{\check#1#2%
  \ifx\ofif#2\ofif\fi\fifo#2}
\def\ofif#1\ofif{\fi}

\newif\ifgobbleone

\def\check#1#2{\ifgobbleone
  \gobbleonefalse
  \else
  \FIND #1
    i:{\FIND #2
         e:{{\bf ie}\gobbleonetrue};
         #2:{i};
       \END};
    #1:{#1};
  \END
  \fi}
```

I have used the two-token scheme successfully in a variety of situations. For the same engineering textbook format, I devised an elementary chemistry mode, so that

```
\chem CH4+2O2<>CO2+2H2O \endchem
```

yields

$$CH_4 + 2\,O_2 \rightleftharpoons CO_2 + 2\,H_2O,$$

and a unit mode, so that

```
\unit 6.672,59e-11 m3.kg-1\endunit
```

yields the ISO representation

$$6.672\,59 \times 10^{-11}\,\mathrm{m}^3\!\cdot\!\mathrm{kg}^{-1}.$$

Actually, mentioning that the \FIND-like tail-recursion applies to tokens is not entirely correct. Because it reads arguments, it will also gobble up as one object a group delimited by braces (or by the current TeX delimiters), not a single token. This case cannot happen with the \Pascal macro, for there are no current group delimiters during tail-recursion ('{' and '}' are given category code 12), but it can happen in other situations. When a group is read as argument #1 by \check, the first level of grouping is removed, so the \FIND selection is actually performed on the first token (or group) within the original group. Whether this characteristic is a

feature or a bug depends on your application. Sometimes, it is quite useful: for the chemistry mode above, it enables `\chem C{60}H{60}\endchem` to give the correct output $C_{60}H_{60}$, with the `\FIND` recognizing the '6', but acting on the group 60; by contrast, `\chem C60H60\endchem` yields the incorrect $C_6{}_0H_6{}_0$, with the '0' being a subscript to an empty subformula and hence being too far away from the '6'. Sometimes, however, you may prefer strict token-per-token processing; in the FIFO paper mentioned above, Kees van der Laan shows a way of acting on each token by assigning it to a temporary variable instead of reading it as an argument.

Conclusion

Preprocessing within TeX—reading a list of tokens (or brace-delimited groups) and replacing them with others for TeX to process further—has unlimited applications for TeX users and macro-writers. A processing based on a `\FIND` macro (Fine, 1993) is powerful, especially when nested and applied on two tokens. The progression along the list can be built in the same macro or can be separated, for example using the `\fifo` macro (van der Laan 1993). The approach is powerful enough to handle such tasks as pretty-printing of Pascal code fragments.

Maybe the main advantage of these preprocessing schemes is that they are fast and easy to implement. They are not reserved to large-scope application, but can be used for one-off, *ad hoc* macros as well. I once had to typeset phone numbers on the basis of the following syntax: the code

```
\phone{725.83.64}
```

should yield 725 83 64, that is, periods must be replaced by thin spaces and pairs of digits must be slightly kerned (it looked better for the particular font at that particular size). The corresponding tail-recursion scheme is easy to implement:

```
\catcode'\@=11

\def\k@rn#1#2{\let\thek@rn=\k@rn
  \FIND #1
    0123456789:{#1%
      \FIND #2
        0123456789:{\kern-0.0833em};
        #2:{\relax};
      \END};
    .:{\thinspace};
    #1:{#1};
  \END
  \ifx\end#2\def\thek@rn##1{\relax}\fi
  \thek@rn#2}

\def\phone#1{\k@rn#1\end}

\catcode'\@=12
```

It may not be the fastest piece of TeX code in the world (and some would doubtlessly qualify it as "syntactic sugar"), but it made optimal use of my time, by allowing me to get the job done fast and well.

Bibliography

Fine, Jonathan, "The \CASE and \FIND macros." *TUGboat* **14** (1), pages 35–39, 1993.

Knuth, Donald E., *The TeXbook*. Reading, Mass.: Addison-Wesley, 1984.

Laan, C.G. van der, "\FIFO and \LIFO sing the BLUes." *TUGboat* **14** (1), pages 54–60, 1993.

Laan, C.G. van der, "Syntactic sugar." *TUGboat* **14** (3), pages 310–318, 1993.

Oostrum, Piet van, "Program text generation with TeX/LaTeX." MAPS91.1, pages 99–105, 1991.

Schwarz, Norbert, *Einführung in TeX*. Addison-Wesley, Europe, 1987. Also available as *Introduction to TeX*. Reading, Mass.: Addison-Wesley, 1989.

Appendix: example of use

(The following program may not be particularly representative of code fragments inserted in a textbook with the PPP package, but it has been designed to illustrate as many features of the \Pascal environment as possible.)

```
% Filename="ppptest"

\input ppp

\everycomment={\sl}

\Pascal

Program demo;
const pi=3.141592;
type date=record year: integer;
month:1..12; day:1..31;end;
flags=packed array[0..7] of boolean;
var MyDate:date; MyFlags:flags;
i1,i2:integer;
last_words:string[31];

function factorial (n:integer):integer;
begin
if n<=1 then factorial:=1
else factorial:=n*factorial(n-1);
end;
{\invisible\vadjust[\medskip[\it
$\langle$more code here$\rangle$]\medskip]}

function Days_in_month(theDate:date);
begin
case theDate.month of 1:Days_in_Month:=31;
2:with theDate
do {check if leap year} begin
 if (0=(year mod 4))
then Days_in_Month:=29 else
Days_in_Month:=28;end;
3:Days_in_Month:=31;{\invisible
\vadjust[\hbox[\hskip8em$\vdots$]]}
12:Days_in_Month:=31;
end;

begin
last_words:='That''s all, folks';
end.{Et voil\`a\thinspace!}

\endPascal
```

```
program demo;
const
  pi = 3.141592;
type
  date = record
      year: integer;
      month: 1..12;
      day: 1..31;
    end;
  flags = packed array[0..7] of boolean;
var
  MyDate: date;
  MyFlags: flags;
  i1, i2: integer;
  last_words: string[31];

function factorial(n: integer): integer;
begin
  if n <= 1 then
    factorial := 1
  else
    factorial := n ∗ factorial(n − 1);
end;

⟨more code here⟩

function Days_in_month(theDate: date);
begin
  case theDate.month of
    1:
      Days_in_Month := 31;
    2:
      with theDate do{check if leap year}
        begin
          if (0 = (year mod 4)) then
            Days_in_Month := 29
          else
            Days_in_Month := 28;
        end;
    3:
      Days_in_Month := 31;
              ⋮
    12:
      Days_in_Month := 31;
  end;

  begin
    last_words := 'That''s all, folks';
  end.{Et voilà !}
```

Towards Interactivity for TEX

Joachim Schrod
Technical University of Darmstadt, WG Systems Programming
Alexanderstraße 10, D-64283 Darmstadt, Germany
schrod@iti.informatik.th-darmstadt.de

Abstract

Much work has been done to improve the level of interactivity available to TEX users. This work is categorized, and probable reasons are discussed why it is not really widespread. A more general view of "interactivity" may also lead to other tools. A common prerequisite for all these tools is the need to know about TEX's functionality. The description of TEX should be formal, since the available informal descriptions have not given satisfactory results.

After an abstract decomposition of TEX, an approach for the formal specification of one subsystem (the macro language) is presented. This specification may be interpreted by a Common Lisp system. The resulting Executable TEX Language Specification (ETLS) can be used as the kernel of a TEX macro debugger.

Variations on A Theme

"Interactive TEX" is the oldest theme on TUG meetings: Morris (1980, p. 12) reports that D. E. Knuth started his opening remarks at the first TUG meeting with it.

> [E]arly on he thought an interactive TEX would be useful, but finds now that TEX users internalize what TEX will do to such an extent that they usually know what TEX is going to do about their input and so have no pressing need to see it displayed on a screen immediately after the input is finished.

Already here a precedent is set for most future reflections on an interactive TEX: A user interface for an author is anticipated that gives feedback on the formatting of the document. Actually, many TEX users don't agree with Knuth, they want to see their formatted document displayed. With the arrival of WYSIWYG-class desktop publishing systems, some of them even want to get it displayed while they are editing, and effectively to edit the formatted representation.

It is worth noting that early usage of the term "interactive formatter" concerns mostly immediate feedback, i. e., the ability to see the formatted representation while the document is input (Chamberlin et al. 1982). In the TEX domain this approach was presented first by Agostini et al. (1985), still on an IBM mainframe at this time. Blue Sky Research invested work in that direction, their product Textures is now advertised as an "Interactive TEX."

The most advanced approach in the connection of TEX input with formatted output was explored by the VORTEX project (Chen and Harrison 1988). In principle, it was possible to edit both the TEX source and the formatted representation as the respective entities were linked to each other. A full implementation of this principle is done in the Grif system (Roisin and Vatton 1994), but this is not related to TEX.

The need to work with the formatted document representation was and is particularly motivated by the error-proneness of creating TEX input. Simple errors (e. g., forgetting a brace) occur very often and may lead to complaints in places that are far away from the error's source. In addition, the time lag between the creation of the error and the notification about it is too large for a smooth work flow. Direct manipulation (DMP) systems are environments that couple actions with reactions of the system and provide immediate feedback (Shneiderman 1983). They encourage one to create and change documents in an ad-hoc manner, without the need for much preplanning of abstractions and structures. (One may argue that this is a disadvantage for the task of writing; but this is not an argument I want to address in this article.)

It is important to emphasize that two terms mentioned above, WYSIWYG and DMP, concern completely different abstractions. A WYSIWYG system allows one to manipulate the presentation of a document; it concentrates on the task of creating and changing this presentation, it focuses on formatting. The term WYSIWYG is domain specific, it is tied to software systems that do layout (in the broadest sense, not only of documents). The category DMP is much more general and such on a different abstraction level: It classifies a set of interfaces that enables users to directly manipulate the objects they are working with, and where immediate feedback is given to them concerning these manipulations. DMP interfaces are often realized by means of windows, icons, menus, and pointing devices (e. g., mice); a member of this subclass of DMP interfaces is also

called a WIMP interface (Chignell and Waterworth 1991).

Since DMP interfaces have been shown to ease learning (Svendsen 1991), some approaches to yield interactivity for TEX document creation use separate systems with a DMP interface for document editing. They generate TEX input, to use the power of TEX's typesetting engine. These systems are pure front-ends, it's not possible to read TEX source and edit it. In early systems like Edimath (André et al. 1985) or easyTEX (Crisanti et al. 1986) the eventual target—the TEX language—is still visible. Newer systems like VAX DOCUMENT (Wittbecker 1989) or Arbortext's SGML Publisher hide this detail from the user. (It's quite interesting that these systems don't use TEX any more in the strict sense. Arbortext and Digital have modified the program to enhance its capabilities or to be able to integrate it better into the overall environment.)

Quint et al. (1986) noted early that such systems are, in fact, not TEX specific. Something that one can really call an interactive writer's front-end to TEX must be able to read TEX source, to enable not only the creation of documents but also their change. They presented the usage of Grif in such a context, but Grif is only able to understand a very limited subset of TEX markup. Similarly, Arnon and Mamrak (1991) presented the automatic generation of an editing environment for a fixed subset of plain TEX math, by formal specification of this subset.

But there are more usages of the term "interactivity". It is used often to characterize *TEX shells*, too. Developers recognized that the task of writing a document is more than editing and formatting; one has to handle bibliographies, create index, draw figures, etc. Tools are available for many of these tasks, but their existence and the respective handling (e. g., syntax of the command line options) has to be remembered. Environments that integrate these different tools into one coherent representation can release the author from that cognitive burden and can help to concentrate on the real tasks (Starks 1992). Sometimes such environments are labeled "interactive", in particular, if they have a WIMP interface. *Visible interface* is a better attribute for such systems as they do not provide a new level of interactivity—they merely make the current possibilities visible. (This terminology is due to Tognazzini (1992).)

As outlined above, the past has seen many attempts to increase the interactivity level of TEX systems for authors. Nevertheless the typical TEX user still writes the complete text with a general-purpose editor, not using any TEX-specific editing software. Even the low level of an immediate preview (or an early one, i. e., concurrently to TEXing) is not common in use.

The question must be posed why this happens. In my opinion, several reasons may be given:

- Some systems are very ambitious, actually they want to provide new publishing systems that replace TEX. Those systems that have been completed are proprietary and not freely distributable. Since they are not targeted to the mass market, they do not get the initial user base that would make them as widespread as TEX is today. The hypothesis that innovative non-mass systems will not be widespread without being freely distributable is backed up by HOPL-II (1993), the similarity between programming and authoring environments is assumed to exist in this regard.
- Those systems that restrict themselves to a certain subtask (e. g., editing of a formula) are often not prepared to communicate with other tools from the author's workbench. The developers often place unreasonable demands on authors (e. g., to place each formula in a separate file).
- Developers underestimate the inertia of users to stay with their known working environment. They are proud of their "baby", and often don't see that the benefit from their new system does not outweigh the costs of learning it. As an example, most UNIX users won't accept a TEX-specific editor that is not as powerful, flexible, and comfortable as Lucid (GNU) Emacs with AUC-TEX (Thorub 1992)—and that's hard to beat.
- Developers are unaware that there is more to interactivity than the creation of structured editing systems or full-blown WYSIWYG publishing systems. In particular, there exist more tasks in the production of a publication and there are lower levels of interactivity that are probably easier to implement.
- The TEX user interface (i. e. its markup language) is realized as a monolithic Pascal program together with a bunch of non-modular macros. It is not possible to incorporate parts of it (e. g., a hypothetical math typesetting module) into an interactive system. Each system rebuilds its needed abstractions anew, often incompatible with others and only approximating TEX's behavior.

Let's sort these issues out in the rest of this article. First, I will be more specific in the definition of "interactivity" and categorize different forms of it, thereby spotlighting interfaces that I think are needed and possible to create. Then preconditions for an easy realization of such systems will be shown, and the results of preliminary work to illustrate these preconditions will be presented.

On Interactivity

Interactivity means (1) that a user may control at run time what the system does, and (2) that feedback to the user's actions happens as soon as possible. If a user may just start a program and cannot control its progress, it is called a *batch program*. If a user may trigger an action at any time, even if another action is still running, the software system has an *asynchronous* user interface, and is regarded as highly interactive. Such user interfaces are usually WIMP-style, this is the reason why command-line oriented system are considered to have a lower level of interactivity—the user may act only at specific points in time, when asked by the system. Interactive systems are notoriously difficult to create, Myers (1994) has shown that this difficulty is inherent in the problem domain.

In the TEX area, we can identify at least the following forms of interaction that might be supported or enabled by software systems, to increase the level of interactivity available to users:

- Full-fledged publishing systems; with DMP, preferably WIMP-style, user interfaces
- Structural editing facilities for specific document parts
- Program visualization for educational purposes (e. g. training)
- Support for TEX macro development

This list is ordered by the additional abstraction level these interfaces provide and the difficulty of producing them. (Of course, the correspondence is not by chance.)

It seems that full-fledged publishing systems are the dream of many developers. But they tend to neglect two facts of life: Such systems have to be much better than existing ones, and their development will not succeed in the first attempt. Systems of the size one has to expect will never be written at once, they have to be developed incrementally. This is the case with all successful middle-sized software systems; it's worth to note that TEX is not an exception to that rule. (The current TEX is the third completely rewritten version, not counting TEX3, according to Knuth (1989).)

To improve on an existing system, one has to address at least the full production cycle. To be concerned only with the demands of authors is not enough any more; document designers, editors, supervisors, etc. work with documents as well. For instance, more appropriate help for designers can be supplied by better layout description facilities (Brüggemann-Klein and Wood 1992). Such facilities need better input methods as well, as designers are usually not trained to work with formal description methods. Myers (1991) shows convincingly that the paradigm of *programming by demonstration* may help here.

In addition, one must not restrict users too much, contrary to the belief of many software developers they are not unintelligent. That means that the straitjacket of pure structural editors, where everything *must* be done via menu and mouse, is not necessarily the right model to use. Research in programming environments (where such straitjacket interfaces did not succeed either) shows that it is possible to build hybrid editors that combine support for structured editing with free-format input, providing immediate feedback by incremental compilation (Bahlke and Snelting 1986).

Last, but not least, one should not forget to scrutinize persuasions we've grown fond of. For example, the concept of markup itself might be questioned, as shown by Raymond et al. (1993). Let's look outside the goldfish bowls we are swimming in, and build new ones.

If one does not have the facilities to produce a new publishing system, one might at least create tools that help users with specific tasks. Even on the author's task domain, one still needs editing facilities that fully understand *arbitrary* TEX math material or tables, and provide appropriate actions on them that are beyond the realm of a text-oriented editor.

Such tools must be able to communicate with other tools, preferably they should provide flexible means to adapt to different protocols. It's in the responsibility of the developer to provide the user with configurations for other tools to access this new one; the best tool will be tossed away if its advantages are too difficult to recognize.

In theory, tools for subtasks can also be used as building blocks of a complete system. In practice, this approach needs further study before one can rely on it. Good starting points for the management of such a tool integration approach are the ECMA standards PCTE and PCTE+ (Boudie et al. 1988).

Development of subtask tools is a hazard; it may be that one constructs a tool that will not be used because it does not enhance productivity enough. Therefore one should make provisions, so that the time spent for development should not be thrown away. The target should be a collection of modules that may be reused for further development projects. This must be taken into account very early, reusability is a design issue and cannot be handled on the implementation level alone (Biggerstaff and Richter 1989).

TEX is here to stay and will be used for a long time. Even the construction of a system that is ultimately better will not change this fact.[1] Experienced users must not forget the difficulties they had in

[1] It may be argued that TEX will be the FORTRAN or C of document markup languages—not the best tool available, but widely used forever.

learning TeX, even though they might by now have internalized how to prevent typical problems—as predicted by Donald Knuth. A topic of research might be the creation of systems that help to explore the functionality of TeX for novice users. For instance, the comprehension of the way TeX works may be made easier by program visualization (Böcker et al. 1986). A tool that visualizes the state of TeX's typesetting engine, allows one to trigger arbitrary actions interactively, and gives immediate feedback on state changes would enhance the understanding considerably. Similarly, advanced TeX courses will be able to make good use of a tool that visualizes the data entities of TeX's macro processor and allows the interactive, visible, manipulation of such entities. Such visualization tools may even be the kernel of a whole TeX macro programming environment (Ambler and Burnett 1989).

Let's not forget the poor souls in TeX country: those who develop macro packages and have to work in a development environment that seems to come from the stone age. This is not necessarily meant as a critique of Donald Knuth's program or language design, as it is reported that he did not anticipate the usage of TeX in the form it's done today. In fact, creating macro packages is programming; programming with a batch compiler.

But even for command-line based batch compilers (e. g., classic compilers for imperative languages) we're used to have a debugger that allows us to interact with the program while it is running. Each programming language defines an abstract machine, the state of this machine can be inspected and changed by the debugger. Execution of a program can be controlled by breakpoints, single stepping, etc. The debugging support available in the TeX macro interpreter is minimal. A first improvement would be an interpreter for the TeX macro language without the typesetting engine, since many errors already happen on the language semantics level.

Preconditions for Realizations

All presented aspects to increase interactivity have one need in common: they rely on access to information that one usually considers internal to TeX. Access to intermediate states of the typesetting process, values of the macro processor, etc., is crucial to build maintainable interactive systems. Since the production of reusable modules is also an aim, the access should not be by ad-hoc methods or heuristic inverse computations. Instead, well defined interfaces are preferred. Actually, before we may define interfaces we need a precise description what TeX "does" at all. In this context, precise explicitly means formal. Informal descriptions are not an adequate tool, after all we want to create a base of understanding, to be used as the underlying model and the terminology of module interface definitions.

The formal description must classify and categorize subsystems of TeX. It's important to take a system point of view in such a classification. Informal specifications that describe the functionality of TeX from a user's point of view exist—but they have not been of much use for the construction of further TeX tools. The target group of a formal description is different: it is not intended to be understood (or even read) by authors, software professionals will use it.

A System View on TeX

The classification of our formal description will be guided by a general model of TeX: It may be considered as an abstract machine. The v. Neumann model—processing unit, data storage, and control unit—is suited also as a model for TeX.

Here the *processing unit* is the typesetter engine, the parts of TeX that break paragraphs and pages, hyphenate, do box arrangements, transform math materials into boxes, etc.

The *data storage unit* is a set of registers that can save values (glues, boxes, etc.) for later usage. The storage and the processing unit work both with a set of abstract classes. These classes are basic elements of a "TeX base machine" abstraction, their instances are the things that are passed to the processing unit to parameterize its actions. We can see them as the primitive, assembler-level data types of the TeX computer.

The *control unit* allows us to access the registers and to trigger operations of the typesetter engine. In TeX, this control unit is hidden beneath a macro language. It is important to be aware of the fact that the macro language is *not* identical with the control unit, it is even not on the same semantic level. In particular, primitive types like boxes or even numbers don't exist in the macro language. (Numbers may be represented by token lists; i. e., the macro language handles only representations—sequences of digits—not number *entities*.) Furthermore, often the macro language only permits us to trigger many typesetting operations at once. These are typical signs of a high-level language.

In terms of our demand for a system-view specification of TeX this is important: *We don't have access to the assembler level of the TeX computer.* That implies that this level is not described informally in available documents; it must be deduced from fragmentary remarks in the TeXbook and there might even be more than one "correct" model of that level.

Eventually, we have a coarse categorization, a decomposition of TeX that is presented in figure 1. The typesetting engine and the storage unit are considered as a component, together with basic object classes. This *TeX base machine* is the basis of the system; in other environments such a component is called a *toolkit*. It is accessible through the control

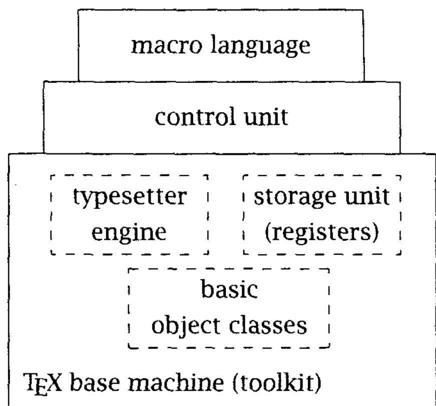


Figure 1: Subsystems of TEX

unit, the subsystem that allows the definition and evaluation of macro language primitives.

Actually, it might be of interest to compare the result of this data-driven analysis with the TEX modularization. (At least for the SAIL version a system structure is reported by Morris (1980).) Since Knuth used the method of structured programming, his modularization is algorithm-driven. The data-driven approach is preferred here since it will allow an easier isolation of subsystems.

Further work will have to analyze the subsystems, to identify modules thereof. The collection of module specifications will provide us with the formal description of the respective subsystem. The rest of this article will face only one subsystem: the macro language. A formal definition of it is useful if we want to export and import TEX documents into other tools.

Basic Terminology

Before the approach used to specify the macro language is presented, we must settle on a precise terminology that is needed for this presentation. While TEXnical jargon and anthropomorphic terms like "mouth" and "stomach" might make for some light and enjoying reading hour, I would like to use the dull terminology of computer science and introduce a few definitions:

Characters are read by TEX from a file. They are transformed to TEX-chars. With the transformation, a character disappears from the input stream and cannot be accessed further on.

Characters are not accessible at the macro language level.

A **TEX-char** is a pair (*category*, *code*). The code of a TEX-char is the xchr code of the read character (wlog. ASCII). The category is determined by the *catcode* mapping on codes. Sequences of TEX-chars are transformed to tokens; most often such a sequence is of length 1. If a TEX-char is transformed, it disappears and cannot be accessed further on.

TEX-chars are not accessible at the macro language level.

A **token** is a pair (*type*, *name*). A name is either an ASCII string or a character; strings of length 1 and characters are distinguished. A token is immutable, neither its type nor its name can be changed.

Token types are *not* TEX-char categories, even if they are often presented as such. The type of a token constructed from exactly one TEX-char is analogous to the category of this TEX-char. But there are categories that have no corresponding types and there is also one type that has no corresponding category. (This token type is symbol, a canonical term for the entities usually called control sequences or active characters.) Since we need to distinguish these two entities, we cannot use the same term for both (as done, for example, in the TEXbook).

In this document, we use the typographic convention (type.name) for token types.

An **action** is a tuple of the form (*semantic function*, param-spec *list*, *primitive*, *expandable*, *value*). It is a basic operation of the TEX macro language, the computational unit a programmer may use, the smallest syntactical unit of a program.

An action may be evaluated to trigger the respective semantic function. The evaluation of an expandable action results in a list of tokens.

An action has an associated parameters specification, the param-spec list. Each param-spec denotes a token list that conforms to some pattern. If an action is evaluated, an argument is constructed for each param-spec, in general by reading tokens from the input stream. These arguments are passed to the primitive.

In addition, an action may yield a value. The computation of the value may need arguments as well, the corresponding param-spec is considered part of the value tuple element.

Users can create new actions by means of macro definitions, the primitive tuple element is used to distinguish builtin (aka primitive) actions and user-defined ones.

In this document, we use the typographic convention action for actions.

A **binding** is a mapping *token* → *action*; every token has an associated action. The action bound to a token that is not of type symbol is fixed, it cannot be changed by the programmer. The macro language defines a set of bindings for symbol tokens, each token not in this set is bound to the action undefined.

These definitions allow a precise description how TEX processes its input:

1. A token is taken from the input stream.
2. The action bound to this token is determined.
3. The arguments for the action are constructed, as defined in the action's param-spec list. More tokens might be read from the input stream for this purpose.
4. The action is evaluated.
5. If the action was expandable, the result of the evaluation is pushed on front of the input stream. I. e., the next token taken from the input stream will be the first token of the result's list.

These steps are repeated until the action end is evaluated. The semantic function of this action will terminate the process.

A very good, and longer, explanation of the way TEX processes its input, may be found in a tutorial by Eijkhout (1991). In contrast to the explanations above, this tutorial takes a process-oriented view, whereas my analysis is data-centered.

Formal Language Specification

The TEX macro language (TML) has neither a common syntactic structure nor a "standard semantics", like those found in imperative or functional programming languages. The formal specification of such a language is not to be taken as an easy task; we are warned by Knuth (1990, p. 9):

> In 1977 I began to work on a language for computer typesetting called TEX, and you might ask why I didn't use an attribute grammar to define the semantics of TEX. Good question. The truth is, I haven't been able to figure out *any* good way to define TEX precisely, except by exhibiting its lengthy implementation in Pascal. I think that the program for TEX is as readable as any program of its size, yet a computer program is surely an unsatisfying way to define semantics.

Of course, one is well advised to take his statement seriously and to be specifically cautious in applying the attribute grammar framework. This difficulty is primarily caused by the inadequacy of context free grammars to describe the TML syntax in an elegant way, see below. Besides attribute grammars (Knuth 1968), other methods for formal language specification are the operational approach, axiomatic specification, and denotational semantics.

In the operational approach (Ollongren 1974), a transformation of language constructs to a prototypical computer model is done, i. e., the language semantics are explicated by construction. That is the earliest approach to define formal language semantics, it was used in the definition of PL/I. The method is particularly suited for languages that are to be compiled.

Axiomatic specifications, usually used for correctness proofs of algorithms, are also applicable to formal language definition; Hoare (1969) mentioned that already in his seminal paper. This approach has not been used often, due to the very complicated descriptions that result. Even Hoare and Wirth (1973) ignored hairy parts when they specified Pascal.

The denotational semantics method specifies a language by defining mappings of its syntactic constructs into their abstract "meaning" in an appropriate mathematical model (Stoy 1977). (Typically, that model is based on the lambda calculus.) The mapping is called the syntax construct's *semantic function*.

Since TML is not compiled, we will use a specification method that belongs to the denotational semantics category. First, we have to identify the syntactic elements of TML. The previous section explained that the computational model of TML is that of evaluation of actions, expanding macros as a side-effect. That implies that we can regard actions as top-level syntactic elements, there is no element that is created by combining several actions. Therefore we have to supply exact syntactic definitions for each action, supply the appropriate semantic function, and will get a full TML definition this way.

In previous work, the TML syntax was formulated partially by a context free grammar (in particular, in BNF format). Of course, the first approach is the incomplete specification given in the summary chapters of the TEXbook (Knuth 1986). Later, Appelt (1988, in German) tried to complete this grammar. Both show the same problems:

- The construction of a token may be configured by the programmer, via the catcode mapping. This is neglected in both grammars, they use exemplary notations for tokens. While this is described exactly by Knuth, Appelt somewhat vaguely introduces the notion of a *concrete reference syntax* for plain TEX (actually, the SGML term is meant) that he uses in his grammar.
- A rather large set of terminal syntactic categories is described by prose (36 in Appelt's grammar, even more in the TEXbook). Sometimes it's even not clear why these syntactical categories are terminal at all, e. g., a BNF rule for ⟨balanced text⟩ is easy to define and not more complicated to read than other definitions that are given.
- The difference between tokens and actions is not explained. Many syntactic structures don't look at specific tokens at all, they care only for the action that is bound to a token.

Most prominently, that happens with the definition of actions (*commands* in TEXbook

terminology) itself. If a terminal token like \parshape appears in the grammar, that does not denote the token (*symbol*."parshape")—an arbitrary token with the bound action parshape is meant instead.

Knuth start the presentation of his grammar fragments with a general explanation of this fact. In addition, every exception—when really a token was meant this time—is mentioned explicitly in the accompanying explanation. He even introduces the notion of implicit characters only for that explanation. (An *implicit character* is a token with type *symbol* where the bound action is an element of the set of actions bound to non-symbol tokens. By the way, the incompleteness of Knuth's prose definition clearly shows the advantage of formal definitions.)

Appelt even ignores that distinction: He uses token notations like '{' both for the description of a token of type *begin-group* and of an arbitrary token with the bound action start-group.

- If arguments for an action are constructed, they may be either *expanded* or *unexpanded* (the tokens that are collected will have been expanded or not). In fact, that is an attribute of a param-spec.

  Knuth notes only in the prose explanation which param-spec category is used for an argument; in addition, this explanation is scattered over the whole TEXbook. Appelt doesn't note this difference at all, e.g., in his grammar def and edef have the same syntax.

These examples should also show the value of a full formal language specification; discussions about the "structure" of a TML construct should not be necessary any more.

ETLS: The Executable TEX Language Specification

ETLS is a denotational semantics style language specification of TML. The mathematical model to which actions (the TML syntactic constructs) are mapped, is a subset of Common Lisp (CL). A set of appropriate class definitions for object classes from the TEX base machine is used as well. The computational aspect of the used CL subset (no continuation semantics or other imperative-style features) is well described and close enough to the lambda calculus to be used as a target model even in the usual sense of denotational semantics.

An action syntax is specified by a param-spec description for each argument. A param-spec is a pair ⟨*expanded*, *pattern*⟩. If a param-spec has the attribute expanded, all tokens that are used to construct an argument are fully expanded first. A pattern is either an identifier from a fixed set, or an alternative of a set of patterns, or an optional pattern. Pattern identifiers either denote a predicate function or an expression on token lists. The actual token list used as the argument will be checked by the predicate or matched by the expression.

Patterns defined by expressions on token lists (e.g. numbers) are specified by context free grammars. Of course, the specification of these parts must not ignore the problematic issues outlined in the previous section: Tokens are explicated as pairs, thereby providing a clear definition for grammar terminals. A special notation for "arbitrary token with a specific action bound to" is introduced. It can be ignored whether the token lists for the argument shall be expanded or unexpanded, though; this is mentioned already in the param-spec description.

A CLOS-style syntax is used for a full action specification. The param-spec list is given like a class slot list. The semantic function is the definition body. The additional attributes (primitive, expandable, and the value function) are put in between, like class attributes.

As an example, consider the specification of the action expandafter:

```
(define-action expandafter
    (:expanded-args
        (skip :token)
        (to-expand :token))
    (:primitive t
     :expandable-action t
     :value nil)
  "Expands the next-after-next token
     in the input stream."
  (cons skip (expand to-expand)))
```

Since this action is expandable, it has to return a list of tokens. That list is the replacement for the token this action was bound to and for the two argument tokens. We create it by prepending the first argument to the top-level expansion of the second argument. A value element of nil specifies that this action does not have any value semantics. (E.g., it is of no use as an argument to the action the.)

Action definitions like above may be embedded in a Common Lisp interpreter. That way we can interpret them directly and test if they have the same semantics as in the TEX processor. But it should be noted that these definitions do not trigger the same error handling as TEX—in case of an error condition they just signal an exception and the surrounding system must supply appropriate handlers.

Application of ETLS

Many people regard the formal definition of a programming language as an exotic goal pursued only by ivory-towered academics. But such work *is* practical and can even lead to immediate results.

As an example, consider the need for a TEX macro debugger. I.e., a tool that provides breakpoints with associated actions, stepwise execution,

tracing of particular macros and argument gathering, full access (read & write) to the state of the macro processor, etc. Everybody who has developed TEX macros at some time will have missed it.

The ETLS already realizes a large part of such a debugger. Since it is embedded in a Common Lisp system, the CL debugger can be fully applied to TEX macro processing. (In some of the better CL systems, even a GUI for the debugger is available.) If the result of an operation from the TEX base machine is needed, the ETLS limits are reached, though. But the implementation of some modules from this level (most prominently, the data storage unit with the basic object classes) allows us already to debug many typical error-prone macros. Of course, if typesetting problems have to be checked, one needs modules that do not yet exist.

The handling of syntax or static semantic errors is a further point where work is to be done. In case of an error, one is not greeted by the well-known "gentle" error messages of TEX, but is confronted with the Lisp fallback handler for a signaled exception. Then one can issue all kinds of Lisp commands (including the continuation of one's macro code). Of course, a better error handling, on the semantic level of a macro writer, can be easily imagined. (Traditionalists may want to have the TEX error loop available as well.)

Conclusion

Often the wish for interactive tools for TEX is mentioned. This covers author tools that can be used with arbitrary TEX documents, or developer tools that help to program in TEX and to understand the way TEX works. A precise description of TEX is a prerequisite for building such tools.

I have presented an abstract decomposition of TEX that sets an agenda for the specification of subsystems. In particular, one subsystem (the macro language) was further analyzed and an approach for its formal specification was presented. The resulting Executable TEX Language Specification (ETLS) is embedded in a Common Lisp interpreter and may be used to parse and partially interpret TEX source code. The immediate applicability of such an executable specification has been described as well, minimal effort is needed to enhance it to a TEX macro debugger.

Further work has to be done to add the (preferably formal) description of more subsystems. A first aim would be an analysis of the respective subsystems and the documentation of a modularization resulting from that analysis.

In addition, the utility of ETLS should be explored further. The TEX debugger needs the addition of error handlers to be of pragmatic use; a better user interface would be valuable as well. The semantic recognition of some substructures (e.g., the contents of haligns and formulas) is minimal and should be improved.

The work presented here is only a first step, but it may be used as the starting point to enhance interactivity for TEX users; though much remains to be done.

Technical Details & Administrivia. CLISP, a freely distributable Common Lisp implementation from the Karlsruhe University, was used for the actual realization of ETLS. CLISP has been ported to many platforms, Unix workstations, and PC-class microcomputers. No other Lisp system has been used until now.

Both systems are available by anonymous ftp from `ftp.th-darmstadt.de`. You find CLISP in the directory `/pub/programming/languages/lisp/clisp` (executables are there as well). ETLS is placed in the directory `/pub/tex/src/etls`.

Acknowledgments. CHRISTINE DETIG provided invaluable comments and helped to improve the document's structure.

References

Agostini, M., Matano, V., Schaerf, M., and Vascotto, M. "An Interactive User-Friendly TEX in VM/CMS Environment". In (EuroTEX85 1985), pages 117-132.

Ambler, Allen L. and Burnett, Margaret M. "Influence of Visual Technology on the Evolution of Language Environments". *IEEE Computer* **22**(10), 9-22, 1989.

André, Jacques, Grundt, Yann, and Quint, Vincent. "Towards an Interactive Math Mode in TEX". In (EuroTEX85 1985), pages 79-92.

Appelt, Wolfgang. *TEX für Fortgeschrittene*, Anhang "TEX-Syntax", pages 149-171. Addison Wesley, 1988.

Arnon, Dennis S. and Mamrak, Sandra A. "On the Logical Structure of Mathematical Notation". In *Proceedings of the TUG 12th Annual Meeting*, pages 479-484, Dedham, MA. TEX Users Group, Providence, RI, 1991. Published as TUGboat 12(3) and 12(4).

Bahlke, Robert and Snelting, Gregor. "The PSG System: From Formal Language Definitions to Interactive Programming Environments". *ACM Transactions on Programming Languages and Systems* **8**(4), 547-576, 1986.

Biggerstaff, Ted J. and Richter, Charles. "Reusability Framework, Assessment, and Directions". In *Software Reusability*, edited by T. Biggerstaff and A. Perlis, volume I (Concepts and Models), pages 1-18. ACM Press, 1989.

Böcker, Heinz-Dieter, Fischer, Gerhard, and Nieper, Helga. "The Enhancement of Understanding through Visual Representations". In *Proceedings*

of CHI '86 Human Factors in Computing Systems, pages 44–50, Boston, MA. ACM SIG on Computer & Human Interaction, 1986.

Boudie, G., Gallo, F., Minot, R., and Thomas, I. "An Overview of PCTE and PCTE+". In *Proceedings of the 3rd Software Engineering Symposium on Practical Software Development Environments*, pages 248–257, Boston, MA. ACM SIG on Software Engineering, 1988. Published as Software Engineering Notes 13(5), 1988.

Brüggemann-Klein, Anne and Wood, Derick. "Electronic Style Sheets". Bericht 45, Universität Freiburg, Institut für Informatik, 1992. Also published as technical report 350 at University of Western Ontario, Computer Science Department.

Chamberlin, D. D., Betrand, O. P., Goodfellow, M. J., King, J. C., Slutz, D. R., Todd, S. J. P., and Wade, B. W. "JANUS: An interactive document formatter based on declarative tags". *IBM Systems Journal* **21**(3), 250–271, 1982.

Chen, Pehong and Harrison, Michael A. "Multiple Representation Document Development". *IEEE Computer* **21**(1), 15–31, 1988.

Chignell, Mark H. and Waterworth, John A. "WIMPs and NERDs: An Extended View of the User Interface". *SIGCHI Bulletin* **23**(2), 15–21, 1991.

Crisanti, Ester, Formigoni, Alberto, and La Bruna, Paco. "EasyTEX: Towards Interactive Formulae Input for Scientific Documents Input with TEX". In (EuroTEX86 1986), pages 55–64.

Eijkhout, Victor. "The structure of the TEX processor". *TUGboat* **12**(2), 253–256, 1991.

EuroTEX85. *TEX for Scientific Documentation. Proceedings of the 1st European TEX Conference*, Como, Italy. Addison Wesley, 1985.

EuroTEX86. *TEX for Scientific Documentation. Proceedings of the 2nd European TEX Conference*, number 236 in Lecture Notes in Computer Science, Strasbourg, FRA. Springer, 1986.

Hoare, C. A. R. "An Axiomatic Basis for Computer Programming". *Communications of the ACM* **12**(10), 576–580, 1969.

Hoare, C. A. R. and Wirth, Niklaus. "An Axiomatic Definition of the Programming Language PASCAL". *Acta Informatica* **2**, 335–355, 1973.

HOPL2. *Proceedings of the 2nd History of Programming Languages Conference (HOPL-II)*, Cambridge, MA. ACM SIG on Programming Languages, 1993. Preprint published as SIGPLAN Notices 28(3).

Knuth, Donald E. "Semantics of Context-Free Languages". *Mathematical Systems Theory* **2**(2), 127–145, 1968.

Knuth, Donald E. *The TEXbook*, volume A of *Computers and Typesetting*. Addison Wesley, 1986.

Knuth, Donald E. "The Errors of TEX". *Software: Practice and Experience* **19**(7), 607–685, 1989.

Knuth, Donald E. "The Genesis of Attribute Grammars". In *Attribute Grammars and Their Applications*, number 461 in Lecture Notes in Computer Science, pages 1–12, Paris, FRA. INRIA, Springer, 1990.

Morris, Robert. "Minutes of the First TUG Meeting". *TUGboat* **1**(1), 12–15, 1980.

Myers, Brad A. "Text Formatting by Demonstration". In *Proceedings of CHI '91 Human Factors in Computing Systems*, pages 251–256, New Orleans. ACM SIG on Computer & Human Interaction, 1991.

Myers, Brad A. "Challenges of HCI Design and Implementation". *interactions* **1**(1), 73–83, 1994.

Ollongren, Alexander. *Definition of Programming Languages by Interpreting Automata*. Academic Press, 1974.

Quint, Vincent, Vatton, Irène, and Bedor, Hassan. "Grif: An Interactive Environment for TEX". In (EuroTEX86 1986), pages 145–158.

Raymond, Darrell R., Tompa, Frank Wm., and Wood, Derick. "Markup Reconsidered". Technical Report 356, University of Western Ontario, Computer Science Department, London, Canada, 1993. Submitted for publication.

Roisin, Cécile and Vatton, Irène. "Merging logical and physical structures in documents". In *Proceedings of the 5th International Conference on Electronic Publishing, Document Manipulation and Typography*, pages 327–337, Darmstadt, FRG. John Wiley, 1994.

Shneiderman, Ben. "Direct Manipulation: A Step Beyond Programming Languages". *IEEE Computer* **16**(8), 57–69, 1983.

Starks, Anthony J. "Dotex—Integrating TEX into the X Window System". In (TUG92 1992), pages 295–303. Published as TUGboat 13(3).

Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

Svendsen, Gunnvald B. "The Influence of Interface Style on Problem Solving". *International Journal of Man Machine Studies* **35**(3), 379–397, 1991.

Thorub, Kresten Krab. "GNU Emacs as a Front End to LATEX". In (TUG92 1992), pages 304–308. Published as TUGboat 13(3).

Tognazzini, Bruce. *Tog On Interface*. Addison Wesley, 1992.

TUG92. *Proceedings of the TEX Users Group Thirteenth Annual Meeting*, Portland, OR. TEX Users Group, Providence, RI, 1992. Published as TUGboat 13(3).

Wittbecker, Alan E. "TEX Enslaved". In *Proceedings of the TUG 10th Annual Meeting*, pages 603–606, Stanford, CA. TEX Users Group, Providence, RI, 1989. Published as TUGboat 10(4).

The Floating World

Frank Mittelbach
Zedernweg 62, D55128 Mainz-Bretzenheim, Germany
Mittelbach@mzdmza.zdv.Uni-Mainz.de

Chris Rowley
The Open University, Parsifal College, Finchley Road, London NW3 7BG, Great Britain
C.A.Rowley@open.ac.uk

Abstract

Working for the past year on a thorough investigation of the output routine of LaTeX has helped to clarify many issues which arise in the modelling and specification of page layout.

How well does LaTeX cope with the job? How does it compare with other software? Can it be improved significantly without a complete redesign?

Looking at the wider picture: how should the designers of future typesetting software approach this aspect of their task?

Similar work will be published in the proceedings of the PoDP'94 Conference.

Sophisticated Page Layout with TeX

Don Hosek
Quixote Digital Typography, 349 Springfield, #24, Claremont, CA 91711, U.S.A.
dhosek@pitzer.edu

Abstract

Page make-up has always been considered one of TeX's weak points. A big part of this is the difficulty of working with traditional output routines for handling special situations. The LaTeX output routine is just one example of how complicated such an approach can be and it is far from being a universal solution. That said, I found myself, in creating my typography magazine, *Serif*, with the problem of handling a fairly complicated page layout that would not be easily addressed with a variation of the plain or LaTeX output routines. This plus a bit of curiosity about what could be done with the line-by-line technique mentioned at the end of the infamous Appendix D led me to consider this approach.

This paper will be published in a future issue of *TUGboat*.

Progress on the Omega Project

John Plaice
Département d'informatique, Université Laval, Ste-Foy, Québec, Canada G1K 7P4
plaice@ift.ulaval.ca

Abstract

Omega (Ω) consists of a series of extensions to TeX that improve its multilingual capabilities. It allows multiple input and output character sets, and will allow any number of internal encodings. Finite state automata can be defined, using a flex-like syntax, to pass from one coding to another. When these facilities are combined with large (16-bit) virtual fonts, even scripts that require very complex contextual analysis, such as Arabic and Khmer, can be handled elegantly.

A year ago (Plaice, 1993), a proposal was made to add the notion of *character cluster* to TeX, and that in fact this notion would be included in an extension of TeX called Ω. The fundamental idea would be that any sequence of letters could be clustered to form a single entity, which could in turn be treated just like a single character. Last year's proposal was not accompanied with an implementation. That is no longer the case, and so the notion of character cluster is now much clearer. Essentially, the input stream passes through a series of filters (as many as are needed), and all sorts of transformations become possible; for example, to handle different character sets, to do transliterations or to simplify ligature mechanisms in fonts. In addition, TeX's restrictions to eight-bit characters have been eliminated.

Encodings and recodings

If we abstract ourselves from the problems associated with layout, typesetting can be perceived as a process of converting a stream of characters into a stream of glyphs. This process can be straightforward or very complex. Probably the simplest case is English where, in most cases, the input encoding and the font encoding are both ASCII; here, no conversion whatsoever need take place. At the other extreme, we might imagine a Latin transcription of Arabic that is to generate highly ligatured, fully vowelized Arabic text; here, the transliteration must be interpreted, the appropriate form of each consonant selected, then the ligatures and vowels chosen — the process is much more complex.

TeX supposes that there are two basic encodings: the input encoding and the internal encoding, each of which uses a maximum of eight bits. The conversion from the input encoding to the internal encoding takes place through an array lookup (xord). An input character is read and converted according to the xord array. The font encoding is the same as the internal encoding, except of course for the fact that several characters can combine to form ligatures.

Suppose that one works in a heterogeneous environment and that one regularly receives files using several different encodings. In this case, one is faced with a problem, because the TeX conversion of input to internal encoding is hard-wired into the code. To change the input encoding, one actually has to change TeX's code — hardly an acceptable situation.

So how does one get around this problem? The first possibility is to use preprocessors, which might themselves be faulty, before actually calling TeX. The second is to use active characters: at the top of every file, certain characters are defined to be macros. However, this process is unreliable, since other macros might expect those characters to be ordinary letters.

Much more appropriate would be to have a command that states that the input encoding has changed and, on the fly, that TeX switches conversion process, maintaining the same internal coding (if we are still in the same document, we probably want to use the same font).

It would probably not be too much trouble to adapt TeX so that it could quickly switch from one one-octet character encoding to another one. However, there are now several multi-octet character sets: JIS, Shift-JIS and EUC in Japan, GB in China and KSC in Korea. Some of these are fixed-width, stateless 16-bit codes, while others are variable-width codes with state. Also, now that the base plane of ISO-10646-1.2 (Unicode-1.1) has been defined, we have a 16-bit character set that can be used for most of the world's languages. However, for reasons of compatibility, we may often come across files in UTF format, where up to 32 bits can be stored in a variable width (1-6 octets) encoding, but for which ASCII bytes remain ASCII bytes. In other words, the conversion process from input to internal encoding is not at all simple.

To complicate matters even further, it is not at all clear what the internal encoding should be. Should it be fixed, in which case the only reasonable possibility is ISO-10646-1.2? Or should the internal

coding itself be variable? If the internal coding is fixed, that will mean that in most cases a conversion from internal encoding to font encoding will have to take place as well. For example, few Japanese fonts are internally encoded according to Han Unification, the principle behind ISO-10646-1.2. Rather, the internal encoding would be by Kuten numbers or by one of the JIS encodings. If that is also the case for the input encoding, then a double conversion, not always simple, nor necessary, would have to take place.

To make matters even worse, one's editor may not always have the right fonts for a particular language. Transliteration becomes a necessity. But transliteration is completely independent from character encodings; the same Latin transliteration for Cyrillic can be used if one is using ISO-646 or ISO-10646. Nor does transliteration have anything to do with font encodings. After all, one would want to use the same Arabic fonts, whether one is typing using a Latin transliteration in ISO-8859-1, or straight Arabic in ISO-8859-6 or ISO-10646.

And, to finish us off, the order of characters in a stream of input may not correspond to the order in which characters are to be put on paper or a screen. For example, as Haralambous (1993) has explained, many Khmer vowels are split in two: one part is placed to the left of a consonantal cluster, and the other part is placed to the right. He has faced similar problems with Sinhalese (Haralambous 1994).

Finally, we should remember that error and log messages must also be generated, and these may not necessarily be in the same character set as either the input encoding or the internal encoding.

Transliteration and contextual analysis. It seems clear that the only viable internal encoding is the font encoding. However, there is no reason that the conversion from input encoding to internal encoding should take but one step. Clearly one can always do this, and in fact, if our fonts are sufficiently large, we can always do all analysis at the ligature level in the font. However, such a decision prevents us from separating distinct tasks, such as — say, for Arabic — first converting all text to ISO-10646, then transliterating, then computing the appropriate form of each letter, and only then having the font's ligature mechanism take over.

In fact, what we propose is to allow any number of filters to be written, and that the output from one filter can become the input to another filter, much like UNIX pipes.

Ω Translation Processes

In Ω, these filters are called Translation Processes (ΩTPs). Each ΩTP is defined by the user in an `.otp` file: with a syntax reminiscent of the Flex lexical analyzer generator, users can define finite state Mealy automata to transform character streams into other character streams.

These user-defined translations tables are not directly read by Ω. Rather, compact representations (`.ctp` files) are generated by the `OTPtoCTP` program. A `.ctp` file is read using the Ω primitive `\otp` (see below). Here is the syntax for a translation file:

`in:`	n;
`out:`	n;
`tables:`	T^*
`states:`	S^*
`aliases:`	A^*
`expressions:`	E^*

where n means any number. Numbers can be either decimal numbers, WEB octal (`@'...`) or hexadecimal (`@"...`) numbers, or visible ISO-646 characters enclosed between a grave accent and a single quote (`` ` ``*c*`'`).

The first (second) number specifies the number of octets in an input (output) character (the default for both is 1). These numbers are necessary to specify the translation processes that must take place when converting to or from character sets that use more than one octet per character.

Tables are regularly used in character set conversions, when algorithmic approaches cannot be simply expressed. The syntax for a table T is:

id`[`n`] = {`$n, n, \ldots, n$`};`

The ΩTPs, as in Flex, allow a number of states. Each expression is only valid in a given state. The user can specify when to change states. States are often used for contextual analysis. The syntax for a set S of states is:

id, *id*, ..., *id*;

Expressions are pattern-action pairs. Patterns are written as simple regular expressions, which can be aliased. The syntax for an alias A is:

id = L;

where L is a pattern.

If only one state is used, then an expression E consists of a pattern and an action:

L => R^*;

where the syntax for patterns is:

L	::=	n	
	|	n-n	range
	|	.	wildcard
	|	LL	concat.
	|	$L\{n, m\}$	occurrences
	|	(L | ... | L)	choice
	|	^(L | ... | L)	negative choice
	|	{*id*}	abbreviation;

and where the simplified syntax for actions is:

```
R  ::=  string
     |  n
     |  \n
     |  \($ − n)
     |  \(* + n − n)
     |  #(R)
     |  id[R]
     |  R op R          arithmetic;
```

Patterns are applied to the input stream. When a pattern has matched the input stream, the action to the right is executed. A *string* is simply put on the output stream. The \n refers to the n-th matched character and the \$ refers to the last matched character. The * refers to the complete matched substream, while \(*−n) refers to all but the last n characters. Table lookup is done using square brackets. All computations must be preceded by a #.

Here is a sample translation from the Chinese GB2312-80 encoding to ISO-10646:

```
in:  1;
out: 2;
tables: tabgb[8795] = {...};
expressions:
(@"00-@"A0)              => \1;
(@"A1-@"FF)(@"A1-@"FF)   =>
     #(tabgb[(\1-@"A0)*@"64 + (\2-@"A0)]);
.                        => @"FFFF;
```

where we use @"FFFF as the error character. And here is a common transliteration in Indic scripts:

```
{consonant}{1,6} {vowel}  => \$ \(*-1);
```

The vowel at the end is placed before the stream of consonants.

The complete syntax for expressions is more complicated, as there can be several processing states. In addition, it is possible to push values back onto the input stack. Here is the complete syntax:

<state> L => R <= R* <newstate>*

The *state* means that if the ΩTP is in that state then this pattern-action pair can possibly be used. The *newstate* designates the new state if this pattern-action pair is chosen.

Here is an example from the contextual analysis of Arabic:

```
<MEDIAL>{QUADRIFORM}{NOT_ARABIC_OR_UNI}
        => #(\1 + @"DD00)
        <= \2
        <pop:>
        ;
```

When in state MEDIAL (in the middle of a word), a letter with four possible forms is followed by a non-Arabic letter, then the output is the quadriform letter plus the value @"DD00. The non-Arabic letter is placed back on the input stack. Then the current state is popped and the ΩTP returns to the previous state, whatever it was.

Loading ΩTPs. Loading an ΩTP is similar to loading a font. The instruction is simply:

\otp*newname* = *filename*

The .ctp file *filename*.ctp is read in and stored in the otp info memory, similar to the font info memory. A number is assigned to the control sequence *newname*, as for fonts. Thereafter, one can refer to that ΩTP either through the generated number or through the newly-defined control sequence.

Input encodings. When reading a file from an unknown source, using an unknown character set, some sort of mechanism is necessary to determine what the character set is. There are two possibilities: either use a default character set or have some way of quickly recognizing what the character set is.

Fortunately, most character sets contain ISO-646 as a subset. The ISO-10646-1.2 character set, in both its 16- and 32-bit versions, retains ISO-646 as its original 128 characters. The only widely-used character set that does not fit this mold is IBM's EBCDIC.

We therefore provide the means for automatically detecting the character set family. It suffices that the user place a comment at the very beginning of each file: the % character is sufficient to distinguish each of the families. A file using an 8-bit extension of ISO-646 begins with the character code 0x25; a file with 16-bit characters begins with 0x00 0x25.[1] Finally, a file using the EBCDIC encoding begins with 0x6C. Should there be no comment character, then the default input encoding (ISO-646) is assumed.

Once Ω knows how to read the basic Latin letters, it is possible to *declare* what translation the input must undergo. This is done with the command \InputTranslation, e.g. \InputTranslation 1 states that the entire input stream, starting immediately *after* the newline at the end of this line, will pass through the first ΩTP process.[2]

It is also possible to change the character set within a file. This process is more difficult, as it is not always clear where *exactly* the change is to take place. Suppose that we pass from an 8-bit character set to a 16-bit character set. It is important that we know what the *last* 8-bit character is and what the *first* sixteen-bit character is.

This question can be resolved by specifying a particular character as being the one which changes.

[1] A file with 32-bit characters would begin with 0x00 0x00 0x00 0x25, but the current version of Ω does not support 32-bit characters.

[2] The syntax for the new primitives has not been finalized. In particular, it is not clear that the explicit numbering of filters and translation processes is simple to manipulate. Those who wish to use Ω should check the manual for the exact syntax.

However, to simplify matters, we assume that all input translation changes take place *immediately after* the newline at the end of the line in which the \InputTranslation command appears.

Transliteration. Once characters have been read, most likely to some universal character set such as ISO-10646, then contextual analysis can take place, independently of the original character set. This analysis might require several filters, each of which is similar to the translation process undergone by the input.

Since the number of filters that we might want to use is arbitrarily large, there are two commands to specify filters:

\NumberInputFilters n

states that the first n input filters are active. The output from the i-th filter becomes the input for the $i+1$-th filter, for $i < n$.

\InputFilter $m\ i$

states that the m-th input filter is the i-th ΩTP.

Sequences of characters with character codes 5, 10, 11 and 12 successively pass through the translation processes n translation processes. It should be understood that the result of the last translation process should be the font encoding itself; it is in this encoding that the hyphenation algorithm is applied.

Our Arabic example then looks like this:

```
\otp\trans      = ISO646toISO10646
\otp\translit   = TeXArabicToUnicode
\otp\fourform   = UnicodeToContUnicode
\otp\genoutput  = ContUnicodeToTeXArabicOut
\InputFilter 0 \translit
\InputFilter 1 \fourform
\InputFilter 2 \genoutput
\NumberInputFilters 3
```

The TeXArabicToUnicode translator takes the Latin transliteration and converts it into Arabic. As for UnicodeToContUnicode, it does the contextual analysis for Arabic; that is, it takes Arabic (in ISO-10646) and, using a private area, determines which of the four forms (isolated, initial, medial or final) each consonant should take. Finally, ContUnicodeToTeXArabicOut determines what slot in the font corresponds to each character. Of course, nothing prevents the font from having its own sophisticated ligature mechanism as well.

Output and special encodings. TeX does not just generate .dvi files. It also generates .aux, .log and many other files, which may in turn be read by TeX again. It is important that the output mechanism be as general as the input mechanism. For this, we introduce the analogous operations:

```
\OutputTranslation
\OutputFilter
\NumberOutputFilters
```

with, of course, the appropriate arguments.

Similarly, in its .dvi files TeX can output commands that are device-driver specific, using \special commands. Since the arguments to \special are themselves strings, it seems appropriate to also allow the following commands:

```
\SpecialTranslation
\SpecialFilter
\NumberSpecialFilters
```

Large fonts

TeX is limited to fonts that have a maximum of 256 characters. However, on numerous occasions, a need has been shown for larger fonts. Obviously, for languages using ideograms, 256 characters is clearly not sufficient. However, the same holds true for alphabetic scripts such as Latin, Greek and Cyrillic; for each of these, ISO-10646-1.2 defines more than 256 pre-composed characters. However, many of these characters are basic character-diacritic mark combinations, and so the actual number of basic glyphs is quite reduced. In fact, for each of these three alphabets, a single 256-character font will suffice for the basic glyphs.

We have therefore decided, as a first step, to offer the means for large (16-bit) virtual fonts, whose basic glyphs will reside in 8-bit real fonts. This is clearly only a first step, but it has the advantage of allowing large fonts, complete with ligature mechanisms, without insisting that all device drivers be rewritten.

In addition to changing TeX, we must also change DVIcopy and VPtoVF, which respectively become XDVIcopy and XVPtoXVF. The .tfm, .vp and .vf files are replaced by .xfm, .xvp and .xvf files, respectively. Of course, the new programs can continue to read the old files.

.xfm files. The .xfm files are similar to .tfm files, except that most quantities use 16 or 32 bits. Essentially, most quantities have doubled in size. The header consists of 13 four-octet words. To distinguish .tfm and .xfm files, the first four octets are always 0 (zero). The next eleven words are the values for *lf*, *lh*, *bc*, *ec*, *nw*, *nh*, *nd*, *ni*, *nl*, *nk*, *ne*, and *np*; all of these values must be non-negative and less than 2^{31}. Now, each *char_info* value is defined as follows:

width index	16 bits
height index	8 bits
depth index	8 bits
italic index	14 bits
tag	2 bits
remainder	16 bits

Each *lig_kern_command* is of the form:

op byte	16 bits
skip byte	16 bits
next char	16 bits
remainder	16 bits

Finally, extensible recipes take double the room.

`.xvp` files. The `.xvp` files are simply `.vpl` files in which all restrictions to 8-bit characters have been removed. Otherwise, everything else is identical.

Minor changes. Since the changes above required carefully examining all of the code for TeX, we took advantage of the opportunity to remove all restrictions to a single octet. So, for example, more than 256 registers (of each kind) can be used. Similarly, more than 256 fonts can be active simultaneously.

Conclusions

The tranformation of TeX into Ω was a necessary step for the development of a typesetting tool that could be used for most (all?) of the world's languages. Scripts that, for various historical and political reasons, retained their calligraphic traditions, can now be printed with ease without sacrificing on aesthetics. In fact, as presented in Haralambous and Plaice (1994), it is now possible to use calligraphic-style fonts for Latin-alphabet languages, without any extra overhead: just change the font and the translation process, everything else is automatic.

Large fonts are definitely useful: all the interactions of characters in a font can be examined. However, it is not necessary to change all our device drivers. A large virtual font might still only reference small real fonts (unlikely to be the case in Eastern Asia, where all fonts are large).

Large fonts, with full interaction between the characters, mean that one can envisage variable-width Han characters. According to Lunde (1993), this topic has been mentioned in several Asian countries.

Finally, I should like to state that the change from TeX to Ω is really quite small. Apart from the idea of character cluster, everything is already there in TeX. It should be considered a tribute to Donald Knuth that so little time was required to make these changes.

Acknowledgements

The Ω project was devised by Yannis Haralambous and myself. It would never have gotten off the ground if it had not been for the numerous discussions that I had with Yannis. Many thanks as well for the discussions in the Technical Working Group on Multiple Language Coordination.

Bibliography

Haralambous, Yannis, "The Khmer script tamed by the lion (of TeX)", *TUGboat* **14**(3), pages 260–270, 1993.

Haralambous, Yannis, "Indic TeX preprocessor: Sinhalese TeX", TUG94 Proceedings, 1994.

Haralambous, Yannis, and John Plaice, "First applications of Ω: Adobe Poetica, Arabic, Greek, Khmer, Unicode", TUG94 Proceedings, 1994.

Lunde, Ken, *Understanding Japanese Information Processing*, O'Reilly and Associates, Sebastopol (CA), USA, 1993.

Pike, Rob, and Ken Thompson, `tcs` program, `ftp://research.att.com/dist/tcs.shar.Z`, 1993.

Plaice, John, "Language-dependent ligatures", *TUGboat* **14**(3), pages 271–274, 1993.

Object-Oriented Programming, Descriptive Markup, and TEX

Arthur Ogawa
TEX Consultants, P.O. Box 51, Kaweah, CA 93237-0051, U.S.A.
ogawa@orion.arc.nasa.gov

Abstract

I describe a synthesis within TEX of descriptive markup and object-oriented programming. An underlying formatting system may use a number of different collections of user-level markup, such as LATEX or SGML. I give an extension of LATEX's markup scheme that more effectively addresses the needs of a production environment. The implementation of such a system benefits from the use of the model of object-oriented programming. LATEX environments can be thought of as objects, and several environments may share functionality donated by a common, more general object.

This article is a companion to William Baxter's "An Object-Oriented Programming System in TEX."

I believe that the key to cost-effective production of TEX documents in a commercial setting is *descriptive markup*. That is, the document being processed contains *content* organized by *codes*, the latter describing the structure of the document, but not directly mandating the format.

The formatting of such a document is embodied in a separate module (usually a file of definitions of formatting procedures) which represents the implementation of a typographic specification (*typespec*). Thus, descriptive markup achieves the separation of document instance from formatting engine.

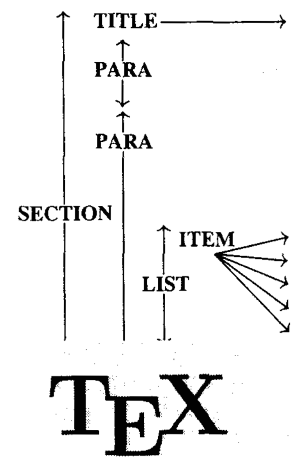
At the same time, the key to cost-effective generation of document formatters in TEX is found in the paradigms of object-oriented programming (OOP). Typographic elements are treated as *objects*, with data and methods. The formatter is a collection of code modules with well-defined boundaries and communication pathways. The programmer can take advantage of OOP techniques such as object encapsulation, data-hiding, and inheritance to create robust, easy-to-maintain, powerful formatters.

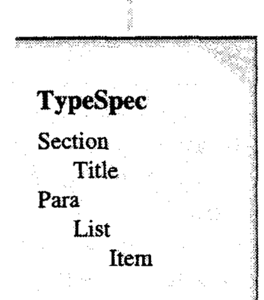
For the purposes of this article, LATEX and SGML will be used as specific instances of descriptive coding schemes, but other methods that cleave to the standards of descriptive markup are not excluded. In particular, databases are very descriptive in nature, and the processing engine described in this and the next paper will process such data well.

The present article discusses issues of descriptive markup and object-oriented programming as relate to TEX and document processing. The next article gives implementation details of the processing engine.

Commercial Typesetting with LATEX

Advantages of LATEX's Descriptive Markup. The descriptive markup of LATEX bestows numerous advantages on this document processing system, making it the predominant TEX macro package.





Organizing the OOP Formatter

To achieve a useful factoring of the code, we want a kernel of basic definitions, with appendages for defining

Insofar possible, we would like the system to allow these parts to be changed independently of each other, and as *late* as possible. So, for instance,

1. the class library, whose structure depends on that of our documents,
2. the user markup (the face), implementing LATEX2ε's \begin and \end, the alternative \open and \close, SGML notation, or other.
3. the element set (a list of element names),
4. the formatting procedures, appropriately parametrized, whose details depend on the typespec,
5. the values of the parameters of those formatting procedures, also dependent on the typespec.

a kernel of basic definitions, with appendages for defining

Insofar possible, we would like the system to allow these parts to be changed independently of each other, and as *late* as possible. So, for instance, we should be able to easily switch between LATEX2 markup syntax and that of SGML, say just before

Simple Syntax. LaTeX's environments and commands provide a simple system of user-level markup; there are only the *environments* (with content) and the *commands* (with argument).

Completeness. LaTeX's public styles are of sufficient richness to accommodate many of the structures required for a typical book. Modest extensions enable one to code fairly technical books.

Context-sensitive formatting. An enumerated list may contain yet another list: the latter is formatted differently than when it appears at the topmost level. The same environment can be used in numerous contexts, so there are fewer markup codes for the author or typesetter to remember.

Authoring versus formatting. Even though using the same set of markup codes as the author, the typesetter may employ a different set of formatting procedures, allowing the author to concentrate on content and structure while leaving the typesetter to deal with the thorny production problems (e.g., float placement, line- and pagebreaks).

Limitations of LaTeX's Markup. Despite the aforementioned advantages, LaTeX has a number of problems.

Inconsistencies. Some of LaTeX's codes introduce syntax beyond the environment and command mentioned above, e.g., the `\verb` command.

Architecture. LaTeX's *moving arguments* and *fragile commands* consititute annoying pitfalls. That the `\verb` command must not appear within the argument of another command has bitten numerous unwary users.

Debasement with procedural markup. When an author inevitably conceives of new markup elements, he or she will commonly be disinclined to simply define new environments to go with them. Instead the author is likely to introduce them in the document instance itself with explicit formatting codes.

The awkward optional argument. Even though many LaTeX commands and environments have a variant (*-form) or an optional argument (within brackets `[]`), not all do, and those that do not are unable to parse a * or optional argument if one does appear in the document. This increases LaTeX's syntactic complexity. Furthermore, the existing scheme is inadequate to accommodate much demand for options, because any one command may have at most one *-form and one optional argument.

User-interface software. Using TeX to code a document is problematic because only TeX can validate the document — and TeX does not perform well as a document validator (nor was it intended for such a use).

Software exists to help generate a valid LaTeX document; the emacs LaTeX mode and TCI's Scientific Word are two such. But neither can assert (as an SGML validator can) that the document has no markup errors.

Limitations of LaTeX Styles. Separating core processing functionality from design-specific formatting procedures is embodied in LaTeX's style (`.sty`) files. It is a useful idea, allowing the considerable investment in LaTeX's kernel to be amortized over a large body of documents, but it has limitations.

Excessive skill requirements for style writers. Because LaTeX exposes TeX's programming language within the style files, only someone skilled in programming TeX can create the style file for a new document typespec. Less daunting is the task of customizing an existing style, but this remains out of the reach of professional designers as a class. This situation stands in sharp contrast to commercial applications such as Frame Maker, which possess what I call a *designer interface.*

Designer-interface software. Some progress has been made to supply software that will generate the code of a LaTeX style, notably TCI's Scientific Word. One can think of a fill-in-the-blank approach that allows one to specify the values of dimensions that parametrize a typespec. But there is currently no method of extending an existing body of styles to accommodate new formatting procedures and parametrizations.

Incomplete Implementation. Much work remains to be done in separating style-specific code from kernel code: LaTeX2's core definitions as they now stand make numerous decisions about document structure and formatting metrics.

Commercial Typesetting with SGML

Because a Standard Generalized Markup Language (SGML) parser can verify the validity of the markup of a document, and because SGML markup is purely descriptive (to first order), it supplies an effective "front-end" to a TeX-based formatter. A number of commercial systems have implemented this idea. At the same time, SGML is not prey to LaTeX's limitations.

Documents in Classes. In an SGML system, a document instance belongs to a class defined by a Document Type Definition (DTD), which specifies the concretes of the markup scheme, the name of each *element*, or tag (in LaTeX: environment), its *attributes* (modifiers) and their allowable values, and the element's *content model.* The latter specifies

what elements may or must appear within a given element, and what order they must appear in. For example,

```
<!ELEMENT theorem - -
 (title, paragraph*)
 >
<!ATTLIST theorem
 id ID #REQUIRED
 kind (theorem|lemma|corollary) #IMPLIED
 >
```

defines the "theorem" element and specifies that it has to be given a key called "id" (like LATEX's `\label` command) and may carry an attribute, "kind", whose value, if specified, must be either "theorem", "lemma", or "corollary". Its content must have an element called "title", followed by any number of paragraphs. The DTD is thus the basis for SGML document validation.

Elements with attributes. SGML has just one syntax for its descriptive markup, namely the element. An element instance may specify the values of its attributes, or may accept a default; this allows the value to be determined effectively by the formatter, or by inheritance from some containing element (discussed in more detail below). A typical instance of an SGML element in a document might be:

```
<theorem ID="oops1" kind=Corollary>
 <title>OOPS, A Theorem</title>
 ⟨content of the theorem⟩
 </theorem>
```

Note that in SGML we really may not give the title as an attribute, because an SGML attribute can not, for instance, contain math. The practice is rather to put the text of the title in an element of its own.

General and consistent markup. The advantages of such a meager syntax cannot be overstated. An author may generate a relatively complex document with a fairly small set of markup. At the same time, SGML application software may assist in selecting and inserting the codes, thereby removing the onus of verbose markup.

The document as database. It is a common school of thought to treat an SGML document instance as rather a collection of structured data than a traditional book or article. This emphasizes the desirability of descriptive markup and the undesirability of procedural markup. Such a document can be published on numerous different media (paper, CD-ROM) and forms (demand publishing, custom publishing). The value of a document coded this way cannot be overstated.

Face-Independent Procedures

Separating Markup from Formatting Procedures. A core processor is something that will serve equally well as a formatter for SGML, LATEX2, LATEX3 and beyond. It must, in fact, be able to parse user markup defined by some external specification, what we call a *face*. At the same time, its style files must not at all determine the input syntax.

Here, I describe the span of user markup that must be parsed. Each one of these markup schemes constitutes a different face of the core processor.

Bestowing Attributes on LATEX Environments. An extension to the LATEX2 syntax which provides flexible SGML-like attributes is:

```
\begin{theorem
 \kind{Corollary}
 \number{2.1}
 \prime{}
 \title{OOPS, A Theorem}
 \label{oops1}
 }
 ⟨content of the theorem⟩
 \end{theorem}
```

This notation is such that current LATEX2 markup simply coincides with default values for all attributes.

SGML Markup. I gave an example of an SGML element instance above. What corresponds to a LATEX sectioning command might appear as:

```
<section ID="sgmlmarkup">
 <title>&SGML; Markup Syntax</title>
 <title-short>&SGML; Markup</title-short>
 <title-contents>&SGML; Markup</title-contents>
 ⟨content of the section⟩
 </section>
```

Here, the elements `<title-short>` and `<title-contents>` would be optional and would specify a short title for the running head and table of contents respectively. The syntax `&SGML;` is that of a text *entity*, an SGML shorthand. Interestingly enough, in a TEX-based processor for SGML markup, it suffices for the two characters < and & to have catcode active (13), with all others as letter (11) or other (12).

Markup for a Successor to LATEX2. For LATEX3 we propose the markup scheme:

```
\open\theorem{
 \number{2.1}
 \prime{}
 \label{oops1}
 }
 \open\title OOPS, A Theorem\close\title
 ⟨content of the section⟩
\close\theorem
```

The options appear in a brace-delimited argument,

while the command name is simply a token. This syntax replaces LaTeX2's environments and commands alike.

Note here that the implementation of the `\title` element could in principle parse its entire content into a TeX macro parameter using the tokens `\close\title` as a delimiter. The same observation also applies to SGML syntax (with `</title>` as the delimiter), but not to LaTeX2's syntax, where the end of the environment contains the brace characters. This observation was evidently not lost on the creators of AMS-TeX, who tend to close out their elements with a control sequence name, like `\endtitle`.

The Defining Word. A system that is able to encompass the above markup syntax may be readily extended to other syntax. More important, though, is that all commands defined by such a system share a single, consistent syntax. LaTeX2 would possess this attribute if all environments were defined by means of `\newenvironment`; anyone who has looked inside LaTeX's core macro file or its style files knows otherwise, though.

The `\newenvironment` command of LaTeX's style files is an instance of what we may call a *defining word*, to borrow a phrase from FORTH. We shall see later the relationship between defining words and the OOP concept of class creation.

Benefits in production. As the next talk will also emphasize, the mere existence of a convenient syntax for element attributes bears importantly on production needs. The need is so longstanding that the TeX Users Group-supplied macros for authoring papers submitted to this conference have a syntax for introducing multiple options, and LaTeX users from time immemorial have resorted to their own techniques, e.g.,

```
{\small
\begin{verbatim}
Your text
On these lines
\end{verbatim}
}
```

to reduce the typesize of an environment.

Object-Oriented Programming and TeX

In a rather happy conjunction of requirements and resources, we are now in a position to employ the 20-year old technology of Object-Oriented Programming (OOP) to advance the 16-year old TeX. Here, I introduce certain OOP concepts and show their relationship with the current work.

Object-Oriented Programming Basics.

Data and procedures are encapsulated into objects. To paraphrase a famous formula:

$$\text{Fields} + \text{Methods} = \text{Object}$$

That is, an object is a self-contained computing entity with its own data and procedures. For instance, we can have a object called "enumerated list", one of whose attributes tells whether it is an arabic, roman, or lettered list. Other instances of enumerated list have their own value for this attribute, determined by the context of the object, or specified in the instance.

The object is an instance of its class. A class abstracts an object. In the above example of enumerated list, all enumerated list objects are molded on the same form, the enumerated list class. When the formatter encounters an enumerated list within the document, it creates an instance of the class (say, object number 5):

$$\exists\,\text{list5} \Leftarrow \text{enumerated list}$$

We can look upon a document as a collection of elements, each being an instance of the related class. The paragraph you are reading falls within a section within a section within a section of an article. Three section objects exist simultaneously, yet distinctly. Each of these sections has a title, as a section must. The title of a section is an attribute which is always defined upon its appearance within a document; there is no (non-trivial) default value determined by the class.

An object's fields are private. Encapsulation refers to the practice of disallowing other objects from directly altering a class's fields; instead, objects pass each other messages. An object may alter one of its own fields in response to another object's message. In a numbered list, for example, the counter is "owned" by the list itself, not by the list item; when the latter is instantiated, it sends a message to the list object to increment the counter.

A derived class inherits from its base class. In what is possibly the most powerful paradigm of OOP, a new class of objects can be created (derived) from an existing (base) class by the addition of new fields and methods. The new, or child, class inherits all the fields and methods of the generating, or parent, class. Some of the added methods may supersede, or *override* those that would otherwise be inherited from the parent.

For instance, we may create an enumerated list class from a basic list class by appending a field which determines whether the list device is an number or letter and by overriding the procedure that formats the list device so that it uses this field

appropriately. All other aspects of the list format are determined by the parent class:

$\forall$ enumerated list $\Leftarrow$ list $\supset$ {counter + device}

A derived class may inherit from more than one parent. In a system with *multiple inheritance*, a new class can be created that inherits simultaneously from two or more exising classes. This is sometimes referred to as *mix-in classes.*

For instance, we may have created a class that numbers its instances, applying this to, say, equations and theorems, but the enumerated list class mentioned above should also be a child of this numbering class. In fact, the enumerated list class inherits from both the list class and the numbering class.

$\forall$ enumerated list $\Leftarrow$ list + counting $\supset$ {device}

The structure of the interrelated classes, including descendents is called the *class hierarchy.*

The object has a context in its document. Since the abovementioned sections are nested, each section has a different hierarchical position within the document. This affects their respective formatting (intentionally so, in order to *reveal* the document's structure). This nesting of elements in the document instance is called the *document hierarchy.*

Note that class hierarchy is independent of any particular document instance, while document hierarchy is not *a priori* related to the class hierarchy. Thus, any two enumerated list objects within a document are instantiated identically (they are "created equal"), regardless of where they might appear. Likewise, within a document, a list item must always appear within a list, but in the class hierarchy discussed in the next paper, the item class is a subclass of a run-in head.

Environments, Elements, and Objects. There seems a fairly straightforward connection between LATEX environments and SGML elements. But where do classes and objects fit in? We can think of a class as an abstract environment or element, and an object as a specific instance thereof within a document.

The distinction between class and object is important, because an instance of a class within a document is allowed to have instance options: these must not affect the fields' values in the class itself, which remain unaltered while the document is processed.

Advantages of the OOP approach. In other venues, OOP is said to have the advantages of good organization, robustness of code, reuse of code, ease of modification and extension, and ease of comprehension.

In descriptive markup, the OOP approach makes particular sense because of the close correspondence between element and class, and between element instance and object instance.

The modularity of objects implies a decoupling between them, allows the methods of one object to be maintained, changed, and extended without affecting other objects, and allows one to learn a particular class hierarchy by first understanding each of its elements separately, then in relation to each other.

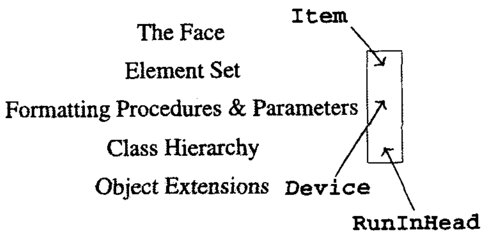
Organizing the OOP Formatter

To achieve a useful factoring of the code, we want a kernel of object extensions, with appendages defining

- the class library, whose structure depends on that of our documents,
- the formatting procedures, appropriately parametrized, whose details depend on the typespec,
- the values of the parameters of those formatting procedures, also determined by the typespec,
- the element set (a list of element names), each bound to a particular formatting procedure. In an SGML formatter, this could be derived automatically from the DTD or some other resource.
- the user markup (the face), implementing LATEX2's `\begin` and `\end`, the alternative `\open` and `\close`, SGML notation, or other syntax.

The figure shows these modules in relationship to each other. The last aspect to be applied, the face, is seen to be truly a very small module placed on top of the entire stack.

Modularity and Late Binding. Insofar as possible, we would like these parts to be independent of each other, and *late* changes should be permitted. So, for instance, we should be able to switch easily between the LATEX2 markup syntax and that of SGML, say just before the `\article` command starts the actual



document. Or, we would like to alter the name of an element; in principle, a `\chapter` command by any other name would still format a chapter opener. Equally well, we may wish to revise the details of a formatting procedure or the value of one of its parameters to reflect an alteration to the typespec. All of these changes are *incremental*. In fact, we shall be able to do all these things principally because TeX is an interpreter, not a compiler.

Maintaining the Formatter. There tends to be an additional relationship, an example of which is indicated, in which an element, a formatting procedure, and a class are connected. In this case, the abstract class `RunInHead` is subclassed to provide what will be known as the `Item` element. In the process, a procedure `Device` is donated, which takes care of the formatting of the list device.

This vertical connection is natural and, to the programmer, compelling. But when developing a document formatter, the distinctions between class, formatting procedure, and element name must nonetheless be preserved for ease of maintenance.

Extensive Use of Defining Words. In order to achieve the greatest of uniformity in the code, we will use defining words exclusively to create the class hierarchy, and to bind the user-level markup codes to their respective procedures. When a new class is derived from another, a defining word is invoked. A user-level code will invoke a different defining word to instantiate an object of a class.

Elsewhere, defining words are used to allocate counters and dimensions (as does LaTeX's `\newcounter` or Plain TeX's `\newdimen`), as well as other, more complex constructs.

Bibliography

Baxter, William E. "An Object-Oriented Programming System in TeX." These proceedings.

Lamport, Leslie. *LaTeX—A Document Preparation System—User's Guide and Reference Manual.* Reading, Mass.: Addison-Wesley, 1985.

Goldfarb, Charles F. *The SGML Handbook.* Clarendon Press, 1990.

Wang, Paul S. *C++ with Object Oriented Programming.* Boston: PWS Publishing, 1994.

An Object-Oriented Programming System in TEX

William Baxter
SuperScript, Box 20669, Oakland, CA 94620-0669, USA
web@superscript.com

Abstract

This paper describes the implementation of an object-oriented programming system in TEX. The system separates formatting procedures from the document markup. It offers design programmers the benefits of object-oriented programming techniques.

The inspiration for these macros comes from extensive book-production experience with LATEX.

This paper is a companion to Arthur Ogawa's "Object-Oriented Programming, Descriptive Markup, and TEX".

The macros presented here constitute the fruit of a struggle to produce sophisticated books in a commercial environment. They run under either plain TEX or LATEX, but owe their primary inspiration to LATEX, especially in the separation of logical and visual design. The author hopes that future TEX-based document production systems such as LATEX3 and NTS will incorporate these techniques and the experience they represent.

Throughout this paper we refer to book production with LATEX. Many of the comments apply equally well to other TEX-based document processing environments.

Design and Production Perspectives

Certain problems routinely crop up during book production with LATEX. The majority fall into two general categories: those related to the peculiarities of a particular job and those regarding the basic capabilities of the production system.

Peculiar documents. Strange, and sometimes even bizarre, element variants often occur within a single document. Without extremely thorough manuscript analysis these surprise everybody during composition, after the schedules have been set. The author received the following queries during production of a single book:

1. What is the proper way to set Theorem 2.1′ after Theorem 2.1?
2. Small icons indicating the field of application accompany certain exercise items. How do we accommodate these variations?
3. This book contains step lists numbered Step 1, Step 2, ..., and other lists numbered Rhubarb 1, Rhubarb 2, ... How do we code these?

Each variation requires the ability to override the default behavior of the element in question, or to create a new element. This is not difficult to accomplish ad hoc. The design programmer can implement prefix commands modifying the default behavior of a subsequent command or environment, add additional optional arguments, or create new commands and environments. But these solutions demand immediate intervention by the design programmer and also require that the user learn how to handle the special cases.

A markup scheme in which optional attributes accompany elements provides a simple, consistent, and extensible mechanism to handle this type of production difficulty. Instead of the standard LATEX environment markup

```
\begin{theorem}[OOPS, A Theorem]
...
\end{theorem}
```

we write

```
\open\theorem{
  \title{OOPS, A Theorem}
}
...
\close\theorem
```

Each attribute consists of a key-value pair, where the key is a single control sequence and the value is a group of tokens. The pair resemble a token register assignment or a simple definition.

The `\open` macro parses the attributes and makes them available to the procedures that actually typeset the element. Thus any element instantiated with `\open...\close` allows attributes. Furthermore, any such element may ignore (or simply complain about) attributes it doesn't understand. For example, if an exercise item coded as an element requires both application and difficulty attributes, it can be coded like this:

```
\open\item{
  \application{diving}
```

```
  \difficulty{3.4}
}
...
\close\item
```

Production can proceed, with the new attribute in place but unused. At some later time the design programmer can modify the procedures that actually typeset the element to make use of the new attribute.

In the case of the rhubarb list, we can use an attribute of the list element to modify the name of the items:

```
\open\steplist{
  \name{Rhubarb}
}
...
\close\steplist
```

Basic capabilities. Complex designs require macro packages far more capable than those of standard LaTeX. The designs require color separation, large numbers of typefaces, letterspacing, complicated page layouts, backdrop screens and changebars, interaction of neighboring elements, and other "interesting" aspects of design. A production system must address all of these aspects of design in order to remain viable in a commercial setting. And it must do so in a cost effective manner.

In their desire to reduce the total and initial costs of a formatting package, production houses ask:

1. How do we use these macros with another manuscript that employs different markup, say, SGML?
2. When will we be able to write our own macro packages?

The first question has a relatively simple answer. We define a generic markup scheme (very similar to `\open...\close`). Design programmers implement their formatting procedures assuming that all documents use this markup. A separate layer of parsing macros translates the markup that actually appears in the document into the generic markup. We call the generic markup scheme the *OOPS markup.* A design implemented behind the OOPS markup is a *formatter.* The markup scheme that actually appears in the document is the *document markup.* A set of macros that translate from a particular document markup into OOPS markup is a *face.*

The OOPS markup constitutes an interface between formatter and face, while the face bridges the gap between document markup and OOPS markup. The same formatter can be used with a different face, and the same face with a different formatter. This newfound ability to reuse a formatter code makes TeX far more attractive to commercial typesetters.

The second question poses a far greater puzzle. At present, implementing a complex book design in TeX or LaTeX requires too much skill for most production houses to maintain. The macros described in the remainder of this paper address this deficiency through the application of object-oriented programming techniques to the problem of design implementation.

The OOPS approach

Before delving too far into the actual workings of the system we deliver the propaganda.

The object-oriented programming paradigm fits the needs of document production extremely well. A document element is an object, and its type is a class. Thus a theorem element is an object of class `theorem`. Deriving one element type from another and overriding some behavior of the new element is a subclassing operation. For example, a lettered list class may be derived from a numbered list class. Attributes correspond to instance variables. The use of the `title` attribute in the theorem example above demonstrates this. The LaTeX notion of a document style resembles a class library.

After defining the OOPS markup, the remainder of this section describes a generic class library. A design programmer implements a particular document design using this standard set of element classes, possibly adding new classes as needed. The reader should consider a class library as an alternative to a LaTeX's document style or document class.

OOPS markup. The OOPS markup for this system works somewhat like the `\open` and `\close` markup scheme presented above. The `\@instantiate` command creates an element of a particular class. It takes two arguments: the name of the class (or parent class) and the list of instance attributes. The command `\@annihilate` destroys an element. It takes the element class to destroy as its single argument. So, reiterating the theorem example from above, the design programmer assumes the following style of markup:

```
\@instantiate\theorem{
  \title{OOPS, A Theorem}
  \number{2.1'}
}
...
\@annihilate\theorem
```

This sample code instantiates an element of class `theorem`, overriding the `title` and `number` attributes. After some other processing it then destroys the the `theorem` instance.

We pause here to provide some clues to the reader about how TeX sees commands, elements, classes, and attributes. A *command* is a control sequence destined for execution by TeX. In contrast, a *class, element,* or *attribute* is a string conveniently represented as a control sequence name. The OOPS

system neither defines nor executes these control sequence names. Instead, it derives a command name from an `element:attribute` or `class:attribute` pair. One can think of *attribute* as a shorthand for *command derived from the* `class:attribute` *or* `element:attribute` *pair.* Thus the phrase *execute an attribute* means *execute the command derived from an* `attribute:element` *or* `attribute:class` *pair.*

Returning to the example, the `\@instantiate` command creates an element of class `theorem`. It copies each attribute of the `theorem` class for the exclusive use of this particular element and overrides the meaning of the `title` and `number` attributes. After instantiation the hidden control sequence corresponding to the `element:title` attribute expands to "OOPS, A Theorem".

Adopting the OOPS markup as the interface between formatting procedures and document parsing routines is not terribly significant. But combining it with object-oriented programming techniques results in a powerful and flexible system for creating new element classes. Inheritance plays the key role. If one element class functions with the standard markup then new elements derived from it inherit this ability.

Subclassing. The `@element` class is the common ancestor of all classes giving rise to document elements. This class supports the OOPS markup described above. It also supports subclassing operations, allowing the design programmer to derive from it new element classes that support OOPS markup.

The `\@class` command derives one element class from another. In the following example we derive the rhubarb list from the steplist.

```
\@class\rhubarblist\steplist{
  \name{Rhubarb}
}
```

The arguments to `\@class` are the new class name (or child class), the name of the parent class, and the list of attributes that override or supplement those of the parent. The `\@class` command parses its arguments, stores them in standard places, and then executes the `@class` attribute from the parent class. The pseudo-code definition of the attribute `@class` in the `@element` class is:

```
@element:@class=
  <execute parent:@preclass>
  <create new class>
  <execute child:@subclass>
```

The `\@new` command carries out the low-level processing for subclassing. It takes the new class name, parent class name, and attribute list parsed by `\@class` out of storage and constructs the new class from them. Every `@class` attribute must execute `\@new` at some point, but can carry out other processing before and after executing `\@new`. The `@preclass` and `@subclass` attributes are subclassing hooks used, for example, to set up a default numbering scheme or allocate a class counter.

In the example above, `\@class` first clones the `steplist` class as `rhubarblist`. If the `name` attribute exists in the `rhubarblist` class then it is altered, otherwise it is added.

The `\@class` and `\@new` commands actually allow multiple inheritance. The parent argument to `\@class` may consist of one or more class names. The `@class` attribute is always executed from the *head parent*, the first parent in the list. With multiple inheritance two parents may contain conflicting definitions of the same attribute. Attributes are passed from parent to child on a first-come-first-served basis. The child inherits the meaning of an attribute from the first parent class containing that attribute.

The definition of the `@class` attribute in most classes is identical to that of `@element` because it is inherited without overriding. But this system permits overriding the `@class` attribute just like any other.

The design programmer uses `\@class` to create a class for each element. During document processing these classes are instantiated as document elements.

Instantiation. To instantiate an element the command `\@instantiate` parses the OOPS markup and squirrels away its arguments in special locations. It then executes the `@instantiate` attribute from the class (from `theorem` in the above example). A pseudo-code definition of the `@instantiate` attribute in the `@element` class is:

```
@element:@instantiate=
  <execute parent:@preinstantiate>
  <create new element instance>
  <execute child:@initialize>
  <execute child:@startgroup>
  <execute child:@start>
```

The `@preinstantiate` attribute is analogous to the `@preclass` attribute. It carries out processing that must precede the creation of the new element. The `@initialize` attribute performs processing required by the newly-created element. The `@startgroup` attribute determines whether the element is subject to TeX's grouping mechanism. It usually expands to `\begingroup`, but may also be left empty, resulting in an ungrouped element (like the LaTeX document environment). The `@start` attribute performs start processing for the particular element, in rough correspondence to the second required argument to the LaTeX command `\newenvironment`.

As with subclassing, the `\@new` command performs the low-level instantiation function. It constructs the new element from the material stored

away by \@instantiate. This brings to light one basic implementation decision: a class is simply an object created with \@class, while an element is an object instantiated with \@instantiate.

The \@annihilate command parses class name and stores it in a standard location. It then executes the @annihilate attribute from the most recent instantiation of that element. A pseudo-code definition of @annihilate in the @element class is:

```
@element:@annihilate=
  <execute element:@end>
  <execute element:@endgroup>
  <destroy element instance>
```

The @end attribute corresponds to the third required argument to the LaTeX \newenvironment command and carries out end processing. The @endgroup attribute matches @startgroup, and usually expands to \endgroup. Like @startgroup it can also be given an empty expansion to eliminate grouping.

The \@free command provides the low-level mechanism for destroying an element instance, complementing \@new. It operates on the element name parsed by \@annihilate, cleaning up the remains of the now-defunct object. At some point @annihilate must execute \@free, but other processing may precede or follow such execution.

In conclusion, the @element class contains the following attributes: @class, @preclass, @subclass, @instantiate, @preinstantiate, @initialize, @startgroup, @start, @annihilate, @endgroup, and @end. The \@class command derives new element classes from old. The \@instantiate command creates a new instance of a class as a document element, while \@annihilate destroys an element. Both \@class and \@instantiate share an underlying mechanism.

The class library. What good is the ability to derive one element class from another? It forms the basis for a class library that can replace a LaTeX document style. We now describe one such class library.

This class library is founded on the @element class described above. All classes are ultimately derived from @element using \@class to add new attributes as needed. We describe block elements, paragraphs, counting elements, listing elements, section elements, and independent elements.

A *block element* contributes to the vertical construction of the page. Examples include sections, theorems, tables, figures, display equations, and paragraphs. We interpret the common features of block elements as attributes in the class @block.

Vertical space separates each block from its surroundings. During book production, final page makeup inevitably requires manual adjustment of the space around certain blocks. Strictly speaking, this violates the principle of purely descriptive markup, but the need is inescapable. The difficulty of properly adjusting the vertical space around LaTeX environments routinely provokes severe consternation in production managers who care about quality.

The @block class unifies the various mechanisms for inserting space above and below any block element with four attributes: @abovespace, @belowspace, abovespace, and belowspace. The first two are for the design programmer, and constitute the default space above and below the element. The latter two are deviations added to the first two in the obvious fashion. They are for the convenience of the final page makeup artist.

Each block element class uses these attributes in the appropriate manner. An ordinary block element, such as a paragraph, may insert space above and below using \addvspace. But a display math element would instead use \abovedisplayskip and \belowdisplayskip. In either case the user is always presented with the same mechanism for adjusting this space: the attributes abovespace and belowspace. Furthermore, a class library can include the code to handle the most common block elements. Therefore the design programmer can work exclusively with the @abovespace and @belowspace attributes in the majority of cases.

A block element may also insert penalties into a vertical list above and below its content. It carries attributes @abovepenalty, @belowpenalty, abovepenalty, and belowpenalty that serve as penalty analogues to the above and below space attributes.

Block elements often require different margins than their surroundings. For example, a design may call for theorems, lemmas, and proofs to indent at each side. Furthermore, the justification may change from block to block. Block elements carry the attributes leftindent, rightindent, leftjustify, and rightjustify. The left and right indentation settings measure the relative offset from the prevailing margins, whereas the justification is an absolute setting. Again, each block element makes appropriate use of these attributes. Typically these attributes contribute to the values of \leftskip and \rightskip, with the fixed portion of the glue coming from the "indent" attribute and the stretch and shrink coming from the "justify" attribute. In this way they effectively decompose these \leftskip and \rightskip, demonstrating that we are not tied directly to the model that TeX provides.

Paragraphs are block elements. In the author's class library they are ungrouped in order to avoid placing unnecessary burdens on the underlying TeX system, and the save stack in particular. This class library also uses explicit paragraph instantiation as the p element, in its most radical departure from LaTeX. With this approach we can disable TeX's automatic insertion of \par at every blank line. This

permits greater liberty in the form of text in the source file. For example, using the \open...\close markup one can write

```
\open\p{}
This is a paragraph.

And this is another sentence
in the same paragraph.
\close\p
```

Explicit paragraph markup also eliminates subtleties that plague LaTeX neophytes everywhere, for example, the significance of a blank line following an environment.

Many elements count their own occurrences in a document. Sections, theorems, figures, and equations are common examples. We call these *counting elements*. These elements carry attributes that allow them to maintain and present the count. The author's class library accomplishes this by allocating count registers for such elements as part of the @subclass attribute processing, assigning the count register to the @count attribute in the new class. This type of register allocation is appropriate for elements, such as document sections, whose context is not restricted (unlike an item element, which always appears within a list). Each counting element also carries a number attribute to present the properly formatted element count in the document.

A *listing element* forms the context for item elements, and an item never appears outside of a list. The item functions somewhat like a counting element. It employs number and @count attributes to present its count in the document, but the count register for the item element is allocated when the containing list is instantiated. The count register is deallocated when the list element is destroyed. This dynamic allocation mechanism avoids placing any constraint (other than the number of available count registers) on the nesting depth of lists.

A *section element* is a counting block element. We leave it ungrouped, in deference to TeX's save stack. A generic section element could be created as follows:

```
\@class\@section{\@block\@counting}{
  \@startgroup{}
  \@endgroup{}
  <attribute overriding and addition>
}
```

The new attributes would include a @head attribute for typesetting the section head. This could immediately typeset a stand-alone head, or defer a run-in head to the subsequent paragraph.

The format of certain elements is independent of their immediate context. Footnotes are the classic example, but in LaTeX there are others: floating environments like figure and table, parboxes, minipage, and paragraph cells within the tabular environment. We call these *independent elements*. How does an independent element separate itself from its context?

Multiple hierarchies. The \@parboxrestore command constitutes the LaTeX mechanism for separating an element from its context in the document. It establishes "ground state" values for a set of TeX parameters, including \leftskip and \rightskip, \par, \everypar, and so on, from which further changes are made. The author's class library modifies this approach, offering the ability to created named "blockstates" consisting of settings for all block element parameters. Such a state may be established at any time, such as at the beginning of an independent element. A paragraph cell within a table would likely use one such blockstate and a floating figure another.

The problem of \everypar is more interesting. It typically carries out processing that one element defers to another. The information must be passed globally since the element deferring it may be grouped. The current LaTeX implementation subjects \everypar to horrible abuse, dramatically reducing its utility. The author's class library rectifies this with "parstates", analogous to blockstates but global in nature. These consist of parameters that hold information deferred to \everypar, including any pending run-in heads, flags indicating special indentation or suppression of the space above the next paragraph block, and the like. A parstate is an object whose attributes contain the parameter values. A corresponding stack tracks parstate nesting.

The definition of \everypar never changes during document processing. It always executes the attribute of its own name from the parstate object on top of the parstate stack. This attribute refers to its sibling attributes for any required parstate information. Push and pop operations on the stack effect "grouping" for parstates.

Perhaps this section should have been marked with a dangerous bend, for it opens a Pandora's box. The author has concluded that TeX's save stack provides insufficient flexibility. The \everypar treatment just described amounts to a separate "save stack" for parstates. We can replace sole reliance on TeX's save stack with different grouping mechanisms for different sets of parameters.

A *hierarchy* consists of objects and a stack to track their nesting apart from other hierarchies. Each object, either class or instance, is part of a single hierarchy. The @element hierarchy contains all element classes and instances, and the @element class forms its root, from which all other elements are ultimately derived. The author has implemented four different hierarchies for the class library: @element, @blockstate, @parstate, @rowstate. The last pertains to rows in tabular material. Rowstates make it easy to specify different default behavior for table

column heads and table body entries within the same column.

An independent element can push and pop the parstate at its own boundaries to insure that it does not pick up any stray material from its "parstate context". Or an element can, as part of its end processing, push the parstate to defer special treatment of the subsequent paragraph. After performing the necessary processing to start the paragraph, that parstate can pop itself from the parstate stack. This requires no cooperation from the other parstate objects on the stack, and therefore does not limit what one can accomplish within the \everypar processing.

Associating objects with hierarchies has another important function. Two or more instances of a particular element class can appear simultaneously within a single document, for example in a nested enumerated list. The user would see something like this:

```
\open\enumerate{}
  \open\enumerate{}
    \open\item{}
    ...
    \close\item
  \close\enumerate
\close\enumerate
```

In this example \enumerate and \item are the parent class names. What objects should be created? Where should the attribute values be hidden? The \@new derives a name for the new object from its hierarchy and stores the new attribute values under this name. To match this, \@free annihilates the most-recently-created object in the given hierarchy. For example, with the definitions

```
\def\open{\@instantiate}
\def\close{\@annihilate}
```

the first invocation of \open will create an object named @element0 and the next will create @element1. Any invocation of \close will annihilate the last such object. This convention suffices because objects within each hierarchy are well nested.

Implementing a Class Library

No single class library can anticipate the multifarious requirements of book publishing. When the inevitable occurs, and a new design moves beyond the capabilities of the available class libraries, the class library writer must extend the system for the design programmer. This section briefly describes the OOPS facilities for writing a class library.

We call one command that defines another a *defining word*. The commands \newcommand and \newenvironment are LaTeX defining words. The \@class and \@instantiate commands from above are defining words in the author's class library. A set of low-level defining words in the OOPS package form the basis for class library creation. Other macros assist in accessing the information stored in the attributes of classes and objects. We now present these programming facilities.

The construction of a new class library begins with the creation of a new hierarchy. The defining word \@hierarchy takes a hierarchy name and a list of attributes as its two arguments. It creates the hierarchy and a class of the same name, endowing the class with the attributes in the second argument. For example, the @element hierarchy described in the last section was created as follows:

```
\@hierarchy\@element{
  \@class{%
    \expandafter\@inherit
    \next@headparent\@preclass
    \global\let\this@element\next@object
    \@new
    \expandafter\@inherit
    \this@element\@subclass
  }
  \@preclass{}
  \@subclass{}
  \@instantiate{%
    \expandafter\@inherit
    \next@headparent\@preinstantiate
    \@new
    \@current\@element\@initialize
    \@current\@element\@startgroup
    \@current\@element\@start
  }
  \@preinstantiate{}
  \@initialize{}
  \@startgroup{\begingroup}
  \@start{}
  \@annihilate{
    \@current\@element\@end
    \@current\@element\@endgroup
    \@free
  }
  \@end{}
  \@endgroup{\endgroup}
}
```

This creates the @element hierarchy and the base class @element within that hierarchy.

Any hierarchy is assumed to support the three basic object manipulation commands \@class, \@instantiate, and \@annihilate. To this end, the \@hierarchy defining word provides functional default values for the corresponding attributes, thus insuring that they are always defined.

The definition of the @element:@class exposes the control sequences \next@headparent and \next@object as storage places for material parsed by \@class for use by \@new. The list of storage locations is: \next@object for the new object

name, `\next@parents` for the list of parent classes, `\next@hierarchy` for the hierarchy of the new object, and `\next@options` for the attribute additions and overriding. The `@class` demonstrates a simple use of these data.

The `\@inherit` command takes an `object: attribute` pair as arguments and executes the corresponding command. The `\this@element` macro is here purely for the sake of convenience. It could be replaced with a set of `\expandafter` commands.

The command `\@current` takes as arguments a `hierarchy:attribute` pair and executes the indicated attribute from the "current" (most recently created) instance within the hierarchy.

Several accessories complete the class library writer's toolkit. These allow setting the value of an attribute as with `\gdef`, `\xdef`, or `\global\let`. These are `\set@attribute`, `\compile@attribute`, and `\let@attribute`. For example, the following invocation defines the `bar` attribute of the `foo` object:

```
\set@attribute\foo\bar{Call me foobar.}
```

The "current" analogues to these commands take a hierarchy name in place of the object name. They are `\set@current`, `\compile@current`, and `\let@current`.

The `\acton@attribute` command looks forward past a single command, constructs the control sequence that represents an attribute, and then executes the command. In the example below, the command `\nonsense` uses as a first argument the control sequence representing the `bar` attribute in the `foo` object:

```
\acton@attribute\nonsense\foo\bar
```

The command `\expandafter@attribute` combines `\expandafter` with `\acton@attribute`. In the following example the `\gibberish` command could look forward to see the first level expansion of the `bar` attribute in the `foo` object.

```
\expandafter@attribute\gibberish\foo\bar
```

Of course, the "current" analogues also exist: `\acton@current`, `\expandafter@current`.[1]

These few tools suffice to support the construction of class libraries to handle a huge variety of elements. We still have not divulged how the information that constitutes an object appears to TEX. The class library writer need not know.

[1] The imaginative reader will note that these last four control sequences reveal the storage mechanism used for object attributes. It is considered bad form to use such information. Moreover, doing so can cause macros to depend on the underlying implementation, and thereby break them when OOPS support primitives are added to TEX.

Conclusions

Compatibility with Plain TEX and LATEX. This entire OOPS package and the markup it employs is fully, but trivially, compatible with the current plain and LATEX macro packages. It does make use of a small number of low-level LATEX macros, but these can easily be provided separately. Because the element instantiation

```
\open\theorem{
  \title{OOPS, There It Is}
}
...
\close\theorem
```

executes neither `\theorem` nor `\endtheorem` it can safely coexist with the standard LATEX invocation

```
\begin{theorem}[OOPS, There It Is]
...
\end{theorem}
```

The standard LATEX implementation makes `\everypar` particularly difficult to use. The mechanism of `\@parboxrestore` can interfere with the `@parstate` mechanism in the author's class library. Even the meaning of `\par` is not constant in LATEX. A class library written with these constraints in mind can work around them. Alternatively, the LATEX macros can be redefined to avoid the interference.

Marking each paragraph as an element constitutes the greatest departure from present-day LATEX. It is possible to create a somewhat less capable system allowing implicit paragraph instantiation. In the author's opinion, hiding verbose markup behind an authoring tool is preferable to dealing with markup ambiguity.

Shortcomings. This system's Achilles heel is its execution speed. Comparisons with ordinary LATEX mean little, however, since the functionality is so divergent. Perhaps the ongoing PC price wars will provide inexpensive relief. A more blasphemous solution consists of adding OOPS support primitives to TEX. The author achieved a twenty percent speed increase through the addition of a single primitive to TEX.

People familiar with standard LATEX do not always easily accept the advantage of a different markup scheme. Document authoring tools that enforce complete markup while hiding the details behind a convenient user interface promise to remove this obstacle. Design programmers who work with LATEX only reluctantly change their approach to programming. Questions like "Will this be compatible with LATEX3" cannot yet be answered.

Experiences in production. This new system has seen use in book production with encouraging results. The author has used the OOPS approach to typeset tabular material with a colored background screen spanning the column heads, to handle bizarre

numbering schemes for theorems described above, and also to handle cases of neighboring elements interacting (successive definitions sharing a single backdrop screen). The verbose markup caused some initial concern to users, but automatic formatting of the element was ultimately considered more important.

Futures. The combination of authoring tools, design editors, and object-oriented macros constitutes a complete, powerful, and cost-effective document production system. Authoring tools can hide the verbose OOPS markup, making it simple for users to work with. Rich markup in turn allows the automation of most design element features. Design editing tools can manipulate classes derived from a particular class library, or even create new class libraries, thus reducing the time and skill presently required to implement a design.

In a commercial book production setting, where tens of thousands of pages are processed in a single year, the prospect of a twenty percent increase in OOPS code execution speed compels the addition of a single primitive. Extensions supporting complex design features are also appropriate, as are more capable class libraries. An author may not require an extended TeX or a sophisticated class library to produce a magnum opus. Adherence to systematic descriptive markup, supported by the OOPS package, allows a production bureau to apply their more capable system to final production without destroying an author's ability to work with the same source files.

Future TeX-based typesetting systems and authoring tools can benefit from the techniques presented here. The advantages the OOPS approach offers are too great to ignore. Everybody should enjoy them.

A World Wide Web Interface to CTAN

Norman Walsh
O'Reilly and Associates, 90 Sherman Street, Cambridge, MA 02140, U.S.A.
norm@ora.com

Abstract

There are a lot of different software packages, style files, fonts, etc., in the CTAN archives. Finding the things you need in a timely fashion can be difficult, as I found out while writing *Making TEX Work*. The ability to combine descriptions of packages with the directory listings from CTAN could help alleviate some of the difficulty. The HyperText Markup Language (HTML) is the document structuring language of the World Wide Web and it provides one possible means of combining different views of the archive into a single vision. The CTAN-Web project is my attempt to provide this vision.

Introduction

A functioning TEX system is really a large collection of programs that interact in subtle ways. Processing even a relatively simple document like this one requires several programs (TEX, a previewer, and a printer driver at the very least), most of which read input files or can be configured in other ways. It was this complexity that lead me to start writing *Making TEX Work* (Walsh 1994), a book I hoped would unravel many of these intricacies (end of plug ;-).

In the process of writing *Making TEX Work*, I looked at a lot of the software packages, style files, fonts, etc., in the CTAN archives. It really made me appreciate how much stuff the TEX user community has made freely available. By my estimates there are more than 31,000 files in more than 2,300 directories in /tex-archive on ftp.shsu.edu.

My first challenge was to find the things that I wanted to write about. This was a long process that involved coordinating (at least mentally) the lists of files in the upper-level CTAN directories, entries from David Jones' TeX-index, descriptions maintained by the CTAN archivists, my own intuitions about what was available, and the tidbits that I had collected over the years from Info-TeX postings. It was occasionally tedious, but it was never really difficult (at least technically).

When the book was beginning to fall into place and I was starting to try to track down all the loose ends, I came to a realization: in the early days, finding things had been an end as well as a means. Now, with pressure mounting on an almost daily basis to finish, I discovered just how hard it was to find things on CTAN. This is not a criticism of the CTAN archivists in *any* way. Without their foresight and diligent efforts, the task could easily become impossible. It's just a fact: there's a *lot* of stuff out there.

One tool became invaluable in my daily efforts: ange-ftp for GNU emacs. GNU emacs, if you aren't familiar with it, is an extremely flexible and powerful editor (it's most common on UNIX workstations, but versions exist for MS-DOS, Windows, OS/2, VMS, and a few other platforms). One of the editing modes of emacs, called *dired*, allows you to "edit" directories (a directory listing appears in a window on the screen). In dired mode, the editing keys let you rename, copy, delete, view, and edit files, among other things. Ange-ftp is an extension for emacs that lets you edit *remote* file systems via ftp in dired-mode. This lets me load the /tex-archive/macros directory from ftp.shsu.edu into an emacs buffer and view files simply by pointing to them and pressing "v". Ange-ftp handles all of the transactions with the ftp client in the background. Ange-ftp made gathering information from README files *much* easier.

Inspiration

What I really wanted wasn't an easier way to browse directories, no matter how grateful I was to have that, but a way of combining the TeX-index and other descriptions with a directory listing in some coherent way. A typical interaction with CTAN, in my experience, goes something like this: I need a *widget*, that's under the *something* directory. Oh! There are several things like that. This one looks interesting. Nope that's not it. How about this one. Yeah, that's better. Still, is this other one better? Nope. Ok, I'll try the second one.

I find this sort of interaction tedious via ftp.

As it happens, I was also beginning to explore the World Wide Web (WWW) at the same time, motivated, in part, by experimentation with LATEX2HTML and other tools that translate TEX documents into HTML for online documentation projects. Might this be the answer, I wondered ...? After several days of hacking, the first incarnation of CTAN-Web was born; the CTAN-Web home page is shown in Color Example 16.

What is the World Wide Web?

The WWW is a vast collection of network-accessible information. In an effort to make this information manageable, protocols have been developed for cross-referencing the Web and software written to browse documents in the Web. One of the most popular browsers is Mosaic, a browser from the NCSA.[1] WWW documents use hypertext to make traversing between documents transparent, allowing the user to follow a stream of ideas without regard to where the embodiment of the ideas exists in the Web.

Hypertext links allow you to build dynamic relationships between documents. For example, selecting a marked word or phrase in the current document can display more information about the topic, or a list of related topics.

Naturally, WWW documents can contain hypertext links to other WWW documents, but they can also contain links to documents available through other servers. For example, *Gopher* servers and anonymous ftp servers. Documents in the WWW are addressed by a "universal resource locator" (URL) that identifies the site from which they are available and the protocol that should be used to retrieve them. The general format of a URL is *protocol://site/pathname*. For example, the URL for the LaTeX help file that I maintain is:

`http://jasper.ora.com/texhelp/LaTeX.html`

In other words, it is available via the *http* protocol at `jasper.ora.com` in the file `/texhelp/LaTeX.html`.

Once retrieved, it is up to the browser to determine how they should be displayed. In addition to displaying HTML documents directly, many browsers can automatically spawn external viewers to view PostScript documents and image files in a variety of formats.

What is HTML?

WWW documents are plain ASCII files coded in HTML (Flynn 1994). HTML provides a convenient way to describe documents in terms of their structure (headings, paragraphs, lists, etc.). HTML is really a particular instance of an SGML document. SGML is the Standard Generalized Markup Language and it is defined by the ISO 8879 specification.

The relationship between SGML and HTML can be a little confusing. SGML provides a general mechanism for creating structured documents. HTML documents are SGML documents that conform to a single, fixed structure. (The HTML specification is available at `http://info.cern.ch/hypertext/WWW/MarkUp/MarkUp.html`.)

[1] The figures in this paper are of the X11 version of Mosiac.

A detailed exploration of structured documentation principles is beyond the scope of this article, however, a few words may help clarify the picture; users familiar with LaTeX are already familiar with structured documentation.

The key notion is that structures (characters, words, phrases, sentences, paragraphs, lists, chapters, etc.) in a document should be identified by *meaning* rather than appearance. For example, here is a sentence that you might find in an installation guide (this sentence is coded in TeX):

```
Use the {\bf cd} command to change to the
{\it /usr/tmp/install} directory.
```

The same sentence might be coded in a structured way like this:

```
Use the <command>cd</command> command to
change to the <directory>/usr/tmp/install
</directory> directory.
```

The advantage of the structured document is that it is possible to answer questions about the *content* of the document. For example, you might check to see if all of the commands that are mentioned in the installation guide are explained in an appendix. Since commands are explicitly identified, it is easy to make a list of all of them. In the unstructured case, it would be very difficult to identify all the commands accurately.

You can achieve structured documentation in TeX with macros, but you are never forbidden from using lower-level commands. The advantage of using a formal structured documentation system, like SGML, is that the document can be validated. You can be sure that the document obeys precisely the structure that you intended. The disadvantage of a formal system is that it must be translated into another form (or processed by a specialized application) before it can be printed, but that is becoming easier. In the case of HTML, many browsers already exist.

Since an HTML document is described in terms of its structure and not its appearance, most HTML documents can be effectively displayed by browsers in non-graphical environments. There is a browser for Emacs called W3 and a browser called Lynx for plain text presentation, for example.

What is CTAN-Web?

CTAN-Web is a collection of WWW documents that combines descriptions of many packages available from CTAN with pointers to each of the files in the archive. At present, the descriptions come from an early draft of my book, David Jones' `TeX-index`, and the `00Description` files in the archives. Over time, additional descriptions will be added. Figure 3 shows the top of the `/tex-archive/macros` directory.

The CTAN-Web also has the following features:

- Links are made directly to other online references in the Web. For example, the online help files provided in the `info/htmlhelp` directory are also available as WWW documents on the net. This fact is exploited in the descriptions of these files by creating a hypertext link directly to the online help.

  In addition, font samples can be displayed for several METAFONT fonts (viewing font samples requires a browser that understands GIF files).[2]
- The CTAN-Web documents are indexed. Users can perform online queries for material based upon any word that appears as a filename or in the online description of any file. Simple conditional searches can also be performed (for example, "x or y" or "x and y").

  A query for "verbatim and plain" finds 5 files and 9 directories.[3]
- Each instance of a file that appears in more than one place in the archive is identified. For example, any reference to the file `verbatim.sty` identifies all 7 instances of it in the archive.
- Want to know which files were modified within the last 12 days? Or between 1 Jan and 31 Jan of 1993? Information about the age of each file is maintained in a separate database, accessible via a script run by the CTAN-Web server. This allows you to perform online queries of the archive by age.
- A "permuted index" is constructed each time the Web is built. This allows you to quickly locate files by name.
- A list of files added or modified in the last 7 or 30 days is also constructed each time the Web is built.
- A tree (hierarchical) view of the archive is also available. The tree view provides a fast means of "walking" down into the lower levels of the archive.

Reaching CTAN-Web

You can reach the CTAN-Web pages by using the URL: `http://jasper.ora.com/ctan.html`

Behind the Scenes

For those who are curious, this section provides a brief description of how the CTAN-Web is constructed. The Web is now rebuilt on a daily basis using the most recent information from the `ftp.shsu.edu` server.

[2] Samples for all the METAFONT fonts will be generated shortly.

[3] In the Web built on 20 May 1994.

Handling the descriptions. In order to quickly locate descriptions for the various packages, I maintain the collection of descriptions in a directory structure that parallels the CTAN archives. Each description file is written in a mixture of TEX and HTML (a mixture is used so that it may one day be possible to produce a printed version of the Web). For example, the current description of `latex-help-html.zip` is shown in Figure 1.

Retrieving files from the archives. One of the first problems that had to be solved was how files would be retrieved from the archives. While it's easy to create a link to a file at an ftp site, in the case of CTAN-Web that isn't sufficient because CTAN exists at several sites. The link really needs to be made to the *closest* ftp site.

Although I suppose it is possible to identify the closest ftp site from the user's host id, that seemed impractical. The following compromise was selected instead: rather than linking files directly to an ftp site, they are linked to a script. The document server (`httpd`) provides a facility for making links that cause a program to be executed; the output produced by this program is then displayed as a WWW document. By passing the name of the file requested by the user as an argument to the script, it was possible to write a retrieval script that dynamically constructs a "retrieval document." The retrieval document contains links to the requested file at each of the CTAN hosts. It is then possible for the user to select the closest host. An example of the retrieval document created for `README.archive-features` is shown in Figure 2.

Selecting a link within the retrieval document causes the browser to actually retrieve the file via anonymous ftp from the selected site.

Documents in the Web. There are three kinds of documents in the CTAN-Web and within each document there are several kinds of links.

Directory documents. There is one directory document in the Web for each directory in the archive. Each directory document lists all of the files in the directory it represents along with their associated descriptions.

Directory names in each document are linked to the corresponding directory documents. File names are linked to filename documents (described below) or to the retrieval script, depending on whether the file occurs multiple times in the archive.

The directory document for the `tex-archive/macros` directory is shown in Figure 3.

Tree documents. There is one tree document in the Web for each directory in the archive that contains subdirectories. The tree document displays three levels of hierarchy starting at the directory it represents.

```
<!-- tex-archive/info/htmlhelp/latex-help-html.zip -->
An HTML version of the LaTeX help file created by George Greenwade.
This is the version provided online at <tt>jasper.ora.com</tt>.
It is also available in VMS format (formatted ASCII),
TeXinfo format, HTML format, and as a Microsoft Windows help file.
<!--ONLINE-->
<P>
The LaTeX help file is also
<A HREF="http://jasper.ora.com/texhelp/LaTeX.html">available online</a>.
<!--/ONLINE-->
```

Figure 1: The description of `latex-help-html.zip` in the Web sources.

Directory names in each document are linked to the corresponding tree document. If a directory in the tree does not have subdirectories, it is linked to its directory document instead.

The tree document for the `tex-archive/macros` directory is shown in Figure 4.

Filename documents. There is one filename document for each file that occurs in more than one place in the hierarchy. The filename document lists all of the instances of the filename.

Each instance of the filename in the document is a link to the directory document where that file resides in the archive.

The filename document for the `verbatim.sty` file is shown in Figure 5.

Building the Web document. Early versions of the Web document were constructed from the `FILES.byname` list from the server `ftp.shsu.edu`. Several *Perl* scripts manipulated the listing and constructed the Web document.

After a few weeks, it became clear that the `FILES.byname` listing was insufficient for constructing the Web document because the list contains no indication of symbolic links, for example. It is also poorly organized for my purposes (the necessity of making multiple passes was causing memory problems). George Greenwade kindly agreed to run a script on the archive that extracts more information and stores it in a form that can be translated into the CTAN-Web document in a single pass. (This information is provided in `/pub/fornorm.gz`, if you're interested).

Room for Improvement

I plan to improve CTAN-Web in a number of ways.

- One of the most important improvements is getting the Web off the node `jasper.ora.com` and distributed amongst all the CTAN hosts. The Internet connection from `jasper` to the outside world is actually quite slow and many users find that the performance is poor.
- Assign fixed URLs to each CTAN directory. At present, most of the URLs are assigned more-or-less sequentially when the Web is constructed. This means that the URL for the `tex-archive/macros/latex2e/contrib` directory, for example, changes over time. This prevents people from saving the URLs of frequently visited regions of CTAN-Web. The top level directories already have fixed names.
- Clean up the descriptions. Using an automatic tool to extract the descriptions from several sources in the archives was a fast way to get a large number of descriptions, but the process was not error free. A small, but significant, number of files in the Web have incorrect descriptions.
- Potentially add a report-generating function that can return an annotated list of the files that match a particular query.

Conclusion

I'm quite pleased with the CTAN-Web. There is room for improvement, but I already find it a faster and more flexible way to search the archives. If only I'd had it before I wrote the book. Ah well, there's always the next edition...

References

Flynn, Peter. "How to Write HTML Files." Hypertext electronic document available using the URL `http://www.ucc.ie/info/net/htmldoc.html`. 1994.

Walsh, Norman. *Making TEX Work.* O'Reilly & Associates, 1994.

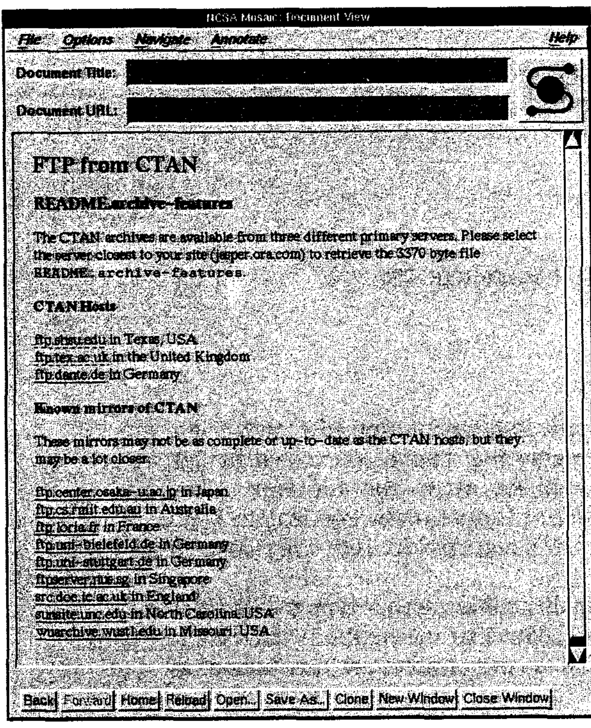


Figure 2: Example of a retrieval document.

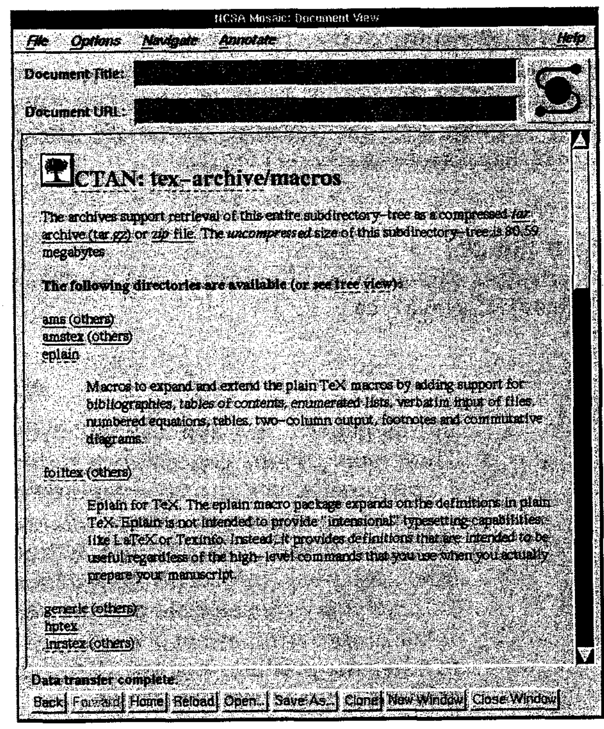


Figure 3: The `CTAN:/macros` directory.

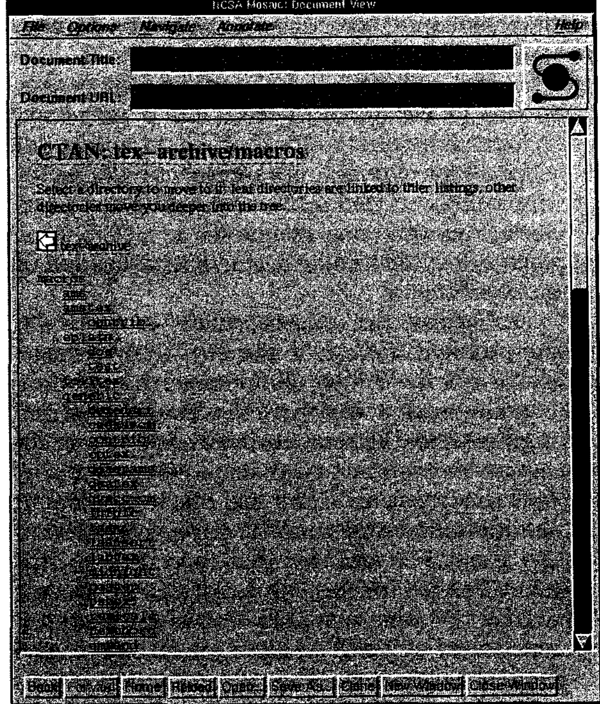


Figure 4: The `tex-archive/macros` tree document

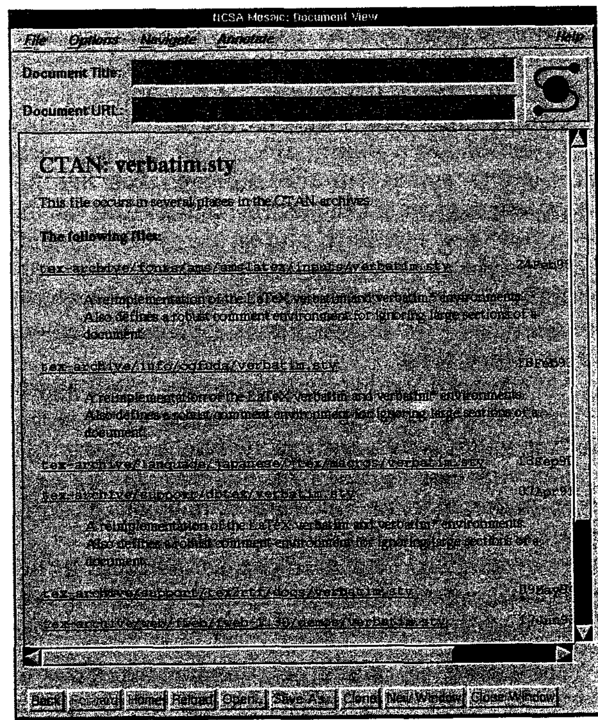


Figure 5: The `verbatim.sty` filename document

First applications of Ω: Greek, Arabic, Khmer, Poetica, ISO 10646/UNICODE, etc.

Yannis Haralambous
Centre d'Études et de Recherche sur le Traitement Automatique des Langues
Institut National des Langues et Civilisations Orientales, Paris.
Private address: 187, rue Nationale, 59800 Lille, France.
Yannis.Haralambous@univ-lille1.fr

John Plaice
Département d'informatique, Université Laval, Ste-Foy, Québec, Canada G1K 7P4
plaice@ift.ulaval.ca

Abstract

In this paper a few applications of the current implementation of Ω are given. These applications concern typesetting problems that cannot be solved by TeX (consequently, by no other typesetting system known to the authors). They cover a wide range, going from calligraphic script fonts (Adobe Poetica), to plain Dutch/Portuguese/Turkish typesetting, to vowelized Arabic, fully diacriticized scholarly Greek, or decently kerned Khmer.

For every chosen example, the particular difficulties, either typographical or TeXnical (or both), are explained, and a short glance to the methods used by Ω to solve the problem is given. A few problems Ω *cannot* solve are mentioned, as challenges for future Ω versions.

On Greek, ancient and modern (but rather ancient than modern)

Diacritics against kerning. It is in general expected of educated men and women to know Greek letters. Already in college, having used θ for angles, γ for acceleration, and π to calculate the area of a round apple pie of given radius, we are all familiar with these letters, just as with the Latin alphabet. But the Greek language, in particular the ancient one, needs more than just letters to be written. Two kinds of diacritics are used, namely accents (acute, grave and circumflex), and breathings (smooth and raw) which are placed on vowels; breathings are also placed on the consonant rho.

Every word has at most one accent[1], and 99.9% of Greek words have exactly *one* accent. Every word starting with a vowel has exactly one breathing[2]. It follows that writing in Greek involves much more accentuation than any Latin-alphabet language, with the obvious exception of Vietnamese.

How does TeX deal with Greek diacritics? If the traditional approach of the \accent primitive had been taken, then we would have practically *no* hyphenation (which in turn would result in disastrous over/underfulls, since Greek can easily have long words like ὠτορινολαρυγγολογικός), *no* kerning, and a cumbersome input, involving one or two macros for every word.

The first approach, originated by Silvio Levy (1988), was to use TeX's ligatures ('dumb' ones first, 'smart' ones later on) to obtain accented letters out of combinations of codes representing breathings (> and <), accents (', ' and ~ or =) and the letters themselves. In this way, one writes >'h to get ᾒ. This approach solved the problem of hyphenation and of cumbersome input.

Nevertheless, this approach fails to solve the *kerning* problem. Let's take the very common case of the article τὸ (letter tau followed by the letter omicron); in almost all fonts there is a kerning instruction between these letters, obviously because of invariant characteristics of their shapes. Suppose now that omicron is accented, and that one writes t'o to get tau followed by omicron with grave accent. What TeX sees is a 't' followed by a grave accent. No kerning can be defined for these two characters, because we have no idea what may follow after the grave accent (it can very well be a iota, and usually there is no kerning between tau and iota). When the letter omicron arrives, it is too late; TeX has already forgotten that there was a tau before the grave accent.

A solution to this problem would be to write diacritics *after* vowels ("post-positive notation"). But

[1] Sometimes an accent is transported from a word to the preceding one: ἄνθρωπός τις instead of ἄνθρωπος, τίς, so that typographically a word has more than one accent.

[2] With one exception: the letters ρρ are often written ῤῥ, when inside a word: πoῤῥῶ.

this contradicts the visual characteristics of diacritics in the upper case, since these are placed on the left of uppercase letters: Ἔαρ could hardly be transliterated `E>'ar`. And, after all, TEX should be able to do proper Greek typesetting, however the letters and diacritics are input.

Ω solves this problem by using an appropriate chain of Ω Translation Processes (ΩTPs), a notion explained in Plaice (1994): as example, consider the word ἔαρ:

1. Suppose the user wishes to input his/her text in 7-bit ASCII; he/she will type `>'ear`, and this is already ISO-646, so that no particular input translation is needed. Another choice would be to use some Greek input encoding, such as ISO-8859-7 or ΕΛΟΤ; then he/she may as well type >' εαρ or >έαρ (the reason for the absurd complication of being forced to type either >' ε or >έ to obtain ἔ, is that "modern Greek" encodings have taken the easy way out and feature only one accent, as if the Greek language was born in 1982, year of the hasty and politically motivated spelling reform). The first ΩTP will send these codes to the appropriate 16-bit codes in ISO 10646/UNICODE: `0x1f14` for ἔ, `0x03b1` for α, and `0x03c1` for ρ.
2. Once Ω knows what characters it is dealing with (Ω's default internal encoding is precisely ISO-10646), it will hyphenate using 16-bit patterns.
3. Finally, an appropriate ΩTP will send Greek ISO 10646/UNICODE codes to a 16-bit virtual font (see next section to find out why we need 16 bits), built up from one or more 8-bit fonts. This font contains kerning instructions, applied in a straightforward manner, since we are now dealing with only three codes: <ἔ>, <α> and <ρ>. No auxiliary codes interfere anymore.
4. `xdvicopy`[3] will de-virtualize the `dvi` file and return a new `dvi` file using only 8-bit fonts, compatible with every decent `dvi` driver.

By separating tasks, hyphenation becomes more natural (for TEX one has to use patterns including auxiliary codes `'`, `‘`, `=` etc.). Furthermore, an additional problem has been solved *en passant*: the primitives `\lefthyphenmin` and `\righthyphenmin` apply to characters of `\catcode` 12. To obtain hyphenation between clusters involving auxiliary codes, we have to declare these codes as 'letter-like characters'. For example, the word ἔαρ, written as `>'ear`: the codes `>` and `'` must be considered as letter-like (non-trivial `\lccode`) to allow hyphenation; but this means that for TEX, ἔαρ *has 5 letters* instead of 3, and hence even if we ask `\lefthyphenmin=3`, we will still get the word hyphenated as ἔ-αρ!! Ω solves this problem by hyphenating *after* the translation has been done (in this case >'e or >' ε or >έ → ἔ).

Dactyls, spondees and 16-bit fonts. Scholarly editions of Greek texts are slightly more complicated than plain ones[4], one of the add-ons being a *third* level of diacritization: syllable lengths.

One reads in Betts and Henry (1989, p. 254): *"Greek poetry was composed on an entirely different principle from that employed in English. It was not constructed by arranging stressed syllables in patterns, nor with a system of rhymes. Greek poets employed a number of different metres, all of which consist of a certain fixed arrangement of* ***long and short syllables****".* Long and short syllables are denoted by the diacritics *macron* and *breve*. The diacritics are placed *between* the letter and the regular diacritic, if any (except in the case of uppercase letters, where they are placed over the letter while regular diacritics are placed to its left).[5]

The famous first two verses of the Odyssey

Ἄνδρα μοι ἔννεπε, Μοῦσα πολύτροπον, ὃς μάλα πολλὰ
πλάγχθη ἐπεὶ Τροίης ἱερὸν πτολίεθρον ἔπερσε

form *hexameters.* These consists of six feets: four dactyls or spondees, a dactyl and a spondee or trochee (see again Betts and Henry 1989 for more details). One could write the text without accents or breathings to make the metre more apparent:

Ᾱνδρᾰ μοῐ ε̄ννε̆πε̆, Μοῦσᾰ πο̆λῡτρο̆πο̆ν, ο̄ς μᾰλᾰ πο̄λλᾰ
πλᾱγχθη̆ ε̆πει̅ Τροῑ̄ης ῐε̆ρο̄ν πτο̆λῐε̄θρο̆ν ε̆πε̄ρσε̆

or one could decide to typeset all types of diacritics:

Ἄ̄νδρᾰ μοῐ ἔ̄ννε̆πε̆, Μοῦ̄σᾰ πο̆λῡ́τρο̆πο̆ν, ὃ̄ς μᾰ́λᾰ πο̄λλᾰ̀
πλᾱ́γχθη̆ ἐ̆πει̅̀ Τροῑ́ης ἱ̆ε̆ρο̄̀ν πτο̆λῐ́ε̄θρο̆ν ἔ̆πε̄ρσε̆

Having paid a fortune to acquire the machine that sits between the keyboard and the screen, one could expect that hyphenation of text and kerning between letters remains the same, despite the constantly growing number of diacritics. Actually this isn't possible for TEX: there are exactly 345 combinations of Greek letters with accents, breathings, syllable lengths and subscript iota; TEX can handle at most 256 characters in a font. Therefore Ω is necessary for hyphenation of Greek text, whenever syllable lengths are typeset.

In this case, things do not run as smoothly as in the previous section: although 345 is a small number compared to 65,536 (= 2^{16}), ISO-people decided that

---

[3] In the name of this program, which is an extended version of Peter Breitenlohner's `dvicopy`, 'x' stands for 'extended', not for 'X-Window'.

[4] After all, scholars have been studying Greek text for more than 2,000 years now.

[5] These additional diacritics are also used for a different purpose: in prose, placed on letters alpha, iota or upsilon, they indicate if they are pronounced long or short (this time we are talking about *letters* and not about *syllables*).

there is not room enough for all combinations of accents, breathings and syllable lengths.[6]

Whenever ISO 10646/UNICODE comes too short for our needs, we use the *private zone*. Just as in the TV serial *The Twilight Zone*, in the private zone *anything* can happen. In the case of Ω all operations remain internal, so that we have absolute freedom of defining characters: according to the 1993-05-01 version of ISO-10646-1, the private zone consists of characters `0xe000` to `0xfffd` (of group 0, that is the 16-bit part of ISO-10646-1), a total of 8,190 positions.

Ω will treat the input like in the previous section, but letters with macron and breve diacritics will occupy internal positions in the private zone. The rest of the treatment will be exactly the same. As for the transliterated input one could take ^ and _ to denote macron and breve (after having changed their catcodes so that they do not interfere with math operators), or any other combination of 7-bit or 8-bit codes.

A dream that *may* come true. As stated by the first author in his Cork talk, back in 1990, his dream was—and still is—to draw a Greek font based on the famous « Grecs du Roi » typeface by Claude Garamont, graved in 1544–46 for the king François I. This typeface was designed after a manuscript of Ἄγγελος Βεργήκιος, a Cretan, calligrapher and reader of Greek at the French Court, in the beginning of the XVI century. There are 1327 different types, most of them ligatures of two or more letters (sometimes entire words). One can read in Nationale (1990) that *"this typeface is the most precious piece of the collection [of the French National Printing House]"*, certainly not the least of honors! Ω is the ideal platform for typesetting with this font, since it would need only an additional ΩTP to convert plain Greek input into ligatures. The author hopes to have finished (or, on a more realistic basis, to have brought to a decent level) this project in time for the International Symposium "Greek Letters, From Linear B to Gutenberg and from G to Ψ", which will take place in Athens in late Spring 1995, organized by the Greek Font Society[7].

[6] Nevertheless, they included lower and upper alpha, iota and upsilon with macron and breve, probably for the reasons explained in the previous footnote. However, combining diacritics must be used to code letters with macron/breve *and* additional diacritics.

[7] For additional information on the Symposium, contact Ἑταιρεία Ἑλληνικῶν Τυπογραφικῶν Στοιχείων, Ἑλλανίκου 39-41, 116 35 Ἀθήνα, Greece, or Michael S. Macrakis, 24 Fieldmont Road, Belmont, MA 02178-2607, USA.

Arabic, or "The Art of separating tasks"

Plain Arabic, quick and clean and elegant. Arabic typesetting is a beautiful compromise between Western typesetting techniques (finite number of types, repeated *ad infinitum*) and Arabic calligraphy (infinite number of arbitrarily complex ligatures). We can subdivide Arabic ligatures into two categories: (a) mandatory ones: letter connections (ه + م → هم) and the special ligature lām-alīf (ل + ا → لا), and (b) optional ones, used for esthetic reasons.

The second category of ligatures corresponds to our good old 'fi' ,'fl' ,etc. They depend on the font design and on the degree of artistic quality of a document. The first author has made a thorough classification of esthetic ligatures of the Egyptian typecase (see Haralambous 1992, reprinted in Haralambous 1993c). Here is an example of the ligaturing process of the word تحمل, following Egyptian typographical traditions:

- ت ح م ل (letters not connected);
- تحمل (only mandatory ligatures, connecting letters);
- تحمل (esthetic ligature between the first two letters);
- تحمل (esthetic ligature between the first three letters).

To produce more than 1,500 possible ligatures of two, three or four letters, three 256-character tables were necessary. Each ligature is constructed by superposition of small pieces. Once TeX knows which characters to take, and from which font, it only needs to superpose them (no moving around is necessary). The problem is to recognize the existence of a ligature and to find out which characters are needed. This process is highly font-dependent. A different font—for example in Kuffic or Nastaliq style—may have an entirely different set of ligatures, or none at all (like the plain font, in which the two words تاريخ العربي are written, that is widely used in computer typesetting because of its readability); nevertheless, the mandatory ligatures remain strictly the same, whatever font is used.

Up to now, there are three solutions to the problem of mandatory Arabic ligatures:

- First, by K. Lagally (1992), is to use TeX macros for detecting and applying mandatory ligatures (we say: "to do the contextual analysis"). This process is cumbersome and long. It is highly dependent on the font encoding and the macros used can interfere with other TeX macro packages. All in all, it is not the natural way to treat a phenomenon which is a low-level fundamental characteristic of the Arabic script.

- Second, by the first author (Haralambous 1993b), is to use "smart" TeX font ligatures (together with TeX--XeT, the bi-directional version of TeX); this process is philosophically more natural, since contextual analysis is done behind the scenes, on the very lowest level, namely the one of the font itself. It does not depend on the font encoding, since every font may use its own set of ligatures. The disadvantage lies in the number of ligatures needed to accomplish the task: about 7,000! The situation becomes tragic when one wants to use a dozen Arabic fonts on the same page: TeX will load 7,000 probably strictly identical ligatures for each font. You need more than BigTeX to do that.
- Third, also by the first author (Haralambous 1993b), is to use a preprocessor. The advantage is that the contextual analysis is done by a utility dedicated to this task, with all possible bells and whistles (for example adding variable length connecting strokes, also known as 'keshideh'); it is quick and uses only a very small amount of memory. Unfortunately there are the classical disadvantages of having a preprocessor treat a document before TeX: one file may `\input` another file, from any location of your net, and there is no way to know in advance which files will be read, and hence have to be preprocessed; preprocessor directives can interfere with TeX macros; there is no nesting between them and this can easily produce errors with respect to TeX grouping operations, etc.

None of these methods can be applied for large scale real-life Arabic production: in all cases, the Arabic script is treated as a 'puzzle to solve' and, inevitably, TeX's performance suffers.

We use ΩTPs to give a natural solution to the problem; consider once again the example of the word تحمل:

- First, تحمل is read by Ω, either in Latin transliteration (`tHml`) or in ISO-8859-6, or ASMO, or Macintosh Arabic, or any decent Arabic script input encoding.
- The first ΩTP converts this input into ISO 10646/UNICODE codes for Arabic letters: `0x062a` (ت), `0x062d` (ح), `0x0645` (م), `0x0644` (ل);
- ISO 10646/UNICODE being a *logical* way of codifying Arabic letters, and not a graphical one, there is no information on their contextual forms (isolated, initial, medial, final). The second ΩTP sends these codes to the private zone, where we have (internally) reserved positions for the combinations of Arabic characters and contextual forms. Once this is done, Ω knows the form of each character.
- The third ΩTP simply translates these codes to a 16-bit standard Arabic TeX font encoding (this is a minor operation: the private zone being located at the end of the 16-bit table, we move the whole block near to the beginning of the table).
- If the font has no esthetic ligatures, we are done: Ω will send the results of the last ΩTP to the DVI file, and produce تحمل. On the other hand, if there are still esthetic ligatures—as in تحمل—then these will be included in the font, as 'smart' ligatures. Since the font table can have as many as 65,536 characters, there is plenty of room for small character parts to be combined.[8]

What we have achieved is that the fundamental process of contextual analysis is done by background machinery (just like TeX hyphenates and breaks paragraphs into lines), and that the optional esthetic refinements are handled exclusively by the font (in analogy to Roman fonts having more ligatures than typewriter ones, etc.).

Vowelized Arabic (things get harder). In plain contemporary Arabic, only consonants and long vowels are written; short vowels have to be guessed by the reader, using the context (the same consonants with different short vowels, can be understood as a verb, a noun, an adjective etc.). When it is essential to specify short vowels, small diacritics are added over or under the letters. Besides short vowels, there are also diacritics for doubling consonants, for indicating the absence of vowel, and for the glottal stop (like in "Oh-oh"): counting all possible combinations, we obtain 14 signs. These diacritics can give TeX a hard time, since they have to be coded between consonants, and hence interfere in the contextual analysis algorithm: for example, suppose that TeX is about to typeset letter x, which is the last letter of a word, followed by a period. Having read the period, TeX knows that the letter has to be of final form (one of the 7,000 ligatures has to be <x in medial form> + <.> → <x in final form><.>). Now suppose that the letter is immediately followed by a short vowel, which in our case is necessarily placed between the letter x and the period. TeX's smart ligatures cannot go two positions backwards; when TeX discovers the period after the short vowel it is too late to convert the medial x into a final one.

Fortunately, ΩTPs are clever enough to be able to calculate letter forms, whatever the diacritics surrounding them (which is exactly the attitude of the

[8] If these esthetic ligatures are used in several fonts, it might be possible that we have the same problem of overloading Ω's font memory; in this case, we can always write a fourth ΩTP, which would systematically make the 'esthetic analysis' out of the contextually analyzed codes.

human typesetter, who first typesets the letters, and then adds the corresponding diacritics).

Nevertheless, ΩTPs are not perfect, and there are problems which cannot be solved, even by the most judicious ΩTP: for example, the positioning of diacritics. We all know that TEX (and hence also Ω, which is no more than an humble extension of TEX) places all elements on a page by using boxes. Unfortunately, the placement of diacritics requires more information than just the height, width, depth and italic correction of a character; in some cases, a real insight into the shape of the character itself and the surrounding characters is necessary (think of ligatures constructed vertically out of four letters, each one having its own diacritic).

This problem can easily be solved for an (esthetically) non-ligatured font: counting all possible letters (not to forget Farsi, Urdu, Pashto, Sindhi, Kirghiz, Uigur and other languages using variants of Arabic letters), in all possible forms one may end up with a figure of no more than a thousand glyphs. Combining these glyphs with the 14 diacritical signs, combinations would result in no more than 14,000 font positions, a figure well under the 65,536 character limit. Since the private ISO 10646/UNICODE zone is not big enough to handle so many characters, we would use an additional ΩTP to send combinations of <contextually analyzed consonant or long vowel> and <diacritic> to codes in the output font encoding. The advantage of this method is that every diacritic can be placed individually (assuming a minute to find the ideal position for a diacritic is needed, the font can be completed in four weeks of steady work), or one can use QDTEXVPL methods to place the diacritics automatically, and then make the necessary corrections.

Unfortunately the number of necessary font positions grows astronomically when we consider 2- or 3-letter ligatures, where each letter can have a different diacritic. One of the future challenges of the Ω project will be to analyze Arabic script characters and find the necessary parameters to determine diacritic positioning, just like D.E. Knuth did for mathematical typesetting. It should be noted that despite the huge complexity of this task, we remain in the strict domain of Arabic *typography*, which is after all no more than a *reduced and simplified version of Arabic calligraphy*.

Multiscript languages: "do you *read* Vietnamese?" Both Westerners and Arabs had the—not so democratic—habit of imposing their alphabet on nations they conquered (either militarily, religiously, culturally or technologically). So it happens that we can all read Vietnamese (but not understand it) just as any Arab can read Malay and Sindhi, and not understand a word (except perhaps for some Arabic words which accompanied the alphabet in its journey).

In some cases, more than one script remained in use for the same language and attempts are made to clarify and standardize the equivalences between letters of these scripts, in order to provide an efficient transliteration algorithm.

The first author has worked on two cases involving the Arabic script: Berber and Comorian.

Berber: a language with three scripts and two writing directions. The Berber language (Berbers call it "Tamazight") can be written in Arabic, Latin or native ("Tifinagh") script. The first author has developed, under the guidance of Salem Chaker,[9] a Tifinagh font in METAFONT. Here is a small sample of Berber text, written in left-to-right Tifinagh:

[illegible]

Follows the same text in right-to-left Tifinagh:

[illegible]

and in Arabic script:

تيفيناغ، دتيرا تيمزوّرا ن يِمازيغن. لّانت دي
تمورتنّغ دات تيرا ن تاعرابت دتلاطينيت.
نّولفانتدّ دات يِمير ن وقّلّيد ماسينيسن.
يِمازيغن ن يِميرن، تّارونتنت غف يِزرا، دقّ
يِفران، غف يِقّدورن، ماشا تيقّتّي غف
يِزكوان : تّارون فلّاسن يِسم ن ومتّين، د
ويِتيلان، د واين يخدم دي تودرتيس اكّن
ور ت تّتّون يِناطفارن.

The encoding of these fonts is such that one can output the same TEX input in Latin transliteration, left-to-right Tifinagh, right-to-left Tifinagh or Arabic.

[9] Director of the Berber Studies Department of the National Institute of Oriental Languages of Paris

All one needs to change is a macro at the beginning. Accomplishing this feature for Latin and Tifinagh was more-or-less straightforward; not so for Arabic. Unfortunately, that font has all the problems of plain Arabic fonts: it needs more than 7,500 ligatures to do the contextual analysis, and is overloaded: there is no longer room to add a single character, an annoyance for a language that is still under the process of standardization.

Another source of difficulties is the fact that the equivalences between Latin, Tifinagh and Arabic are not immediate. Some short vowels are written in the Latin text, but not in the Arabic and Tifinagh ones. Moreover, double consonants are written explicitly in Latin and Tifinagh, while they are written as a single consonant with a special diacritic in Arabic. And perhaps the most difficult problem is to make every Berber writer feel "at home", regardless of the script he/she uses: one should not have the impression that one script is privileged over the others!

Finally, the last problem (not a minor one when it comes to real-life production) is that we need a special Arabic font for Berber, because of the different input transliteration: for example, while in plain Arabic transliteration we use 'v' for ڨ and 'sh' for ش, in Berber we are forced to use 'g' for the former and 'c' for the latter. There are two supplementary letters used for Berber in Arabic script: چ and ژ; these letters are also used in Sindhi and Pashto, so that the glyphs are already covered by the general Arabic TEX system; but in Berber, they have to be transliterated as 'j' and 'z', because of the equivalences with the Latin alphabet. This forces us to use a different transliteration scheme than the one for plain Arabic, and hence—because of TEX's inability to clearly separate input and output encoding—to use a differently encoded TEX output font. Imagine you are typesetting a book in both Berber and Arabic; you will need two graphically identical fonts for every style, point size, weight and font family, each one with more than 7,000 ligatures. And we are just talking of (esthetically) non-ligatured fonts!

Ω solves this problem by using the same output fonts for Berber and plain Arabic. We just need to replace the first ΩTP of the translation chain: the one that converts raw input into ISO 10646/UNICODE codes. Berber linguists can feel free to invent/introduce new characters or diacritics; as long as they are included in the ISO 10646/UNICODE table we will simply have to make a slight change to the first ΩTP (and if these signs are not yet in ISO 10646/UNICODE we will use the private zone).

Comorian: African Latin versus Arabic. A similar situation has occurred in the small islands of the Comores, between Madagascar and the African continent. Both the Latin alphabet (with a few additions taken from African languages) and the Arabic one are used. Because of the many sounds that must be distinguished, one has to use diacritics together with Arabic letters. These diacritics look like Arabic diacritics (for practical reasons) but are not used in the same way; in fact, they are part of the letters, just like the dots are part of plain Arabic letters.

Once again, the situation can easily be handled by an ΩTP. While it is still not clear what should be proposed for insertion into ISO 10646/UNICODE (this proposal, made by Ahmed-Chamanga, a member of the Institute of Oriental Languages in Paris, is now circulating from Ministries to Educational and Religious Institutions, and is being tested on natives of all educational levels), Comorians can already use Ω for typesetting, and upgrade the transliteration scheme on the fly.

Khmer

As pointed out in Haralambous (1993a), the Khmer script uses clusters of consonants, subscript consonants, vowels and diacritics. Inside a cluster, these parts have to be moved around by TEX to be positioned correctly. It follows that TEX *must* use `\kern` instructions between individual parts of a cluster. Because of these, there is no kerning anymore: suppose that characters ខ and ក need to be kerned; and suppose that the consonant ខ is (logically) followed by the subscript consonant ្ក, which is (graphically) placed under this letter: ខ្កក. For TEX, ខ is not immediately followed by ក anymore, and so no kerning between these letters will be applied; nevertheless, graphically they still *are* adjacent, and hence need eventually to be kerned.

Ω uses the sledgehammer solution to solve this problem: we define a 'big' (virtual) Khmer font, containing *all currently known* clusters. As already mentioned in Haralambous (1993a), approximately 4,000 codes would be sufficient for this purpose. And of course one could always use the traditional TEX methods to form exceptional clusters, not contained in that font.

As in Arabic, we deal with Khmer's complexity by separating tasks. A first ΩTP will send the input method the use has chosen, to ISO 10646/UNICODE Khmer codes (actually there aren't any ISO 10646/UNICODE Khmer codes yet, but the first author has submitted a Khmer encoding proposal to the appropriate ISO committees, and expects that soon some step will be taken in that direction—for the moment we will use once again the private zone). A second ΩTP will analyze these codes contextually and will send groups of them to the appropriate cluster codes. The separation of tasks is essential for allowing multiple input methods, without redefining each time the contextual analysis—which is after all a basic characteritics of the Khmer writing system. Ω will

send the result of the second ΩTP to the dvi file, using kerning information stored in the (virtual) font. Finally, xdvicopy will de-virtualize the dvi file and create a new file using exclusively characters from the original 8-bit Khmer font, described in Haralambous (1993a).

Adobe Poetica

Poetica is a chancery script font, designed by Robert Slimbach and released by Adobe Systems Inc. Stating Adobe's promotional material, *"The Poetica typeface is modeled on chancery handwriting scripts developed during the Italian Renaissance. Elegant and straightforward, chancery writing is recognizable as the basis for italic typefaces and as the foundation of modern calligraphy. Robert Slimbach has captured the vitality and grace of this writing style in Poetica. Characteristic of the chancery hand is the common use of flourished letterforms, ligatures and variant characters to embellish an otherwise formal script. To capture the variety of form and richness of this hand, Slimbach has created alternate alphabets and character sets in his virtuoso Poetica design, which includes a diverse collection of these letterforms."*

Technically, the Poetica package consists of 21 PostScript fonts: Chancery I-IV, Expert, Small Caps, Small Caps Alternate, Lowercase Alternates I-II, Lowercase Beginnings I-II, Lowercase Endings I-II, Ligatures, Swash Caps I-IV, Initial Swash Caps, Ampersands, Ornaments. The ones of particular interest for us are Alternate, Beginnings, Endings and Ligatures, since characters from these can be chosen automatically by Ω. The user just types plain text, possibly using a symbol to indicate degrees of alternation. An ΩTP converts the input into characters of a (virtual) 16-bit font, including the characters of all Poetica components. Using several ΩTPs and changing them on the fly will allow the user to choose the number of ligatures he/she will obtain in the output. It will also allow us to go farther than Adobe, and define kerning pairs between characters from different Poetica components.

See Fig. 1 for a sample of text typeset in Poetica.

ISO 10646/UNICODE and beyond

It is certainly not a trivial task to fit together characters from different scripts and to obtain an optically homogeneous result. Often the esthetics inherent to different writing systems do not allow sufficient manipulation to make them 'look alike'; it is not even trivial if this should be tried in the first place: suppose you take Hebrew and Armenian, and modify the letter shapes until they resemble one another sufficiently, to our Western eyes. It is not clear if Armenian would still look like Armenian, or Hebrew like Hebrew; furthermore one should not neglect the psychological effects of switching between scripts (and all other changes the switch of scripts implies: language, culture, state of mind, idiosyncracy, background): the more the scripts differ, the easier this transition can be made.

The only safe thing we can do with types from different origins is to balance stroke widths so that the global gray density of the page is homogeneous (no "holes" inside the text, whenever we change scripts).

These remarks concern primarily scripts that have significantly different esthetics. There is one case, though, where one can apply all possible means of uniformization, and letters can be immediately recognized as belonging to the same font family: the LGCAI group (LGCAI stands for "Latin, Greek, West/East Cyrillic, African, Vietnamese and IPA"). Very few types cover the entire group: Computer Modern is one of them (not precisely the most beautiful), Unicode Lucida another (a nice Latin font but rather a failure in lowercase Greek); there are Times fonts for all members of this group, but there is no guarantee that they belong to the same Times style, similarly for Helvetica and Courier. Other adaptations have been tried as well and it is to be expected that the success (?) of Windows NT will lead other foundries into "extending" their typefaces to the whole group[10].

Fortunately, TEX/Ω users can already now typeset in the whole LGCAI range, in Computer Modern[11] (by eventually adding a few characters and correcting some others). The 16-bit font tables of Ω allow:

1. hyphenation patterns using arbitrary characters of the group;
2. the possibility of avoiding frequent font changes, for example when switching from Turkish to Welsh, to Vietnamese, to Ukrainian, to Hawsa;
3. potential kerning between all characters.

But Ω can go even beyond that: one can include different styles in the same (virtual) font; looking at the ISO 10646/UNICODE table, one sees that LGCAI characters, together with all possible style-dependent dingbats and punctuation, fit in 6

[10] As a native African pointed out to the first author, this will also mean that Africans will find themselves in the sad and paradoxical situation of having (a) fonts for their languages, (b) computers, since Western universities send all their old equipment to the Third World, but (c) no electricity for running them and using the fonts...

[11] *Definitely, sooner or later some institution or company will take the highly praisable initiative of sponsoring the development of other METAFONT typefaces*; the authors would like to encourage this idea.

La Nature est un temple où de vivants piliers
Laissent parfois sortir de confuses paroles ;
L'homme y passe à travers des forêts de symboles
Qui l'observent avec des regards familiers.
Baudelaire, LES FLEURS DU MAL

Figure 1: Sample of the Poetica typeface.

rows (1,536 characters). This means, that—at least theoretically—a virtual Ω font can contain up to 42 (!) style variations[12] of the whole LGCAI group, for example Italic, Bold, Small Caps, Sans Serif, Typewriter and all possible combinations [a total of 24 = (Roman or Italic) × (normal or bold) × (plain or small caps[13]) × (Serif or Sans Serif or Typewriter)]. Defining kerning pairs between all different styles will avoid the use and misuse of \/ (italic correction) and give a better appearance to mixed-style text.

It will be quite an experience to make such a font, since many African and IPA characters have no uppercase or italic or small caps style defined yet; see Jörg Knappen's paper on African fonts (Knappen 1993) for the description of a few problematic examples and the solutions he proposes.

Dutch, Portuguese, Turkish: the easy way

These three languages (and maybe others?) have at least one thing in common: they need fonts with a slightly different ligature table than the one in the Cork encoding. Dutch typesetting uses the notorious 'ij' ligature (think of the names of people very well known to us: Dijkstra, Eijkhout, van Dijk, or the name of the town Nijmegen or the lake IJselmeer); this ligature appears in the Cork encoding (as well as in ISO 10646/UNICODE), but until now there was no user-transparent means of obtaining it. In Ω you just need to place the macro `\inputtranslation{dutchij}` into the expansion of the Dutch-switching macro; according to Ω syntax (described in Plaice 1994) this ΩTP can be written as simply as

```
in: 1
out: 2
expressions:
'I''J' => @"0132;
'i''j' => @"0133;
.      => \1;
```

where @"0132 and @"0133 are characters 'IJ' and 'ij' in ISO 10646/UNICODE.

'f' + 'i' or 'f' + 'ı'"

Portuguese and Turkish typesetting do not use ligatures 'fi',... 'ffl' (in Turkish there is an obvious reason for doing this: the Turkish alphabet uses both letters 'i' and 'ı', and hence it would be impossible to know if 'fi' stands for 'f' + 'i' or 'f' + 'ı'). This is a major problem for TEX, since the only solution that would retain a natural input would be to use new fonts; and defining a complete new set of fonts (either virtual or real), just to avoid 5 ligatures, is more trouble than benefit. Ω solves that problem easily; of course, it is not possible to 'disable' a ligature, since the latter arrives at the very last step, namely inside the font. It follows that we must cheat in some way; the natural way is to place an invisible character between 'f' and 'i'; in ISO 10646/UNICODE there is precisely such a character, namely 0x200b (ZERO WIDTH SPACE); this operation could be done by an ΩTP line of the type `'f''i' => "f" @"200b "i"` for each ligature. This character would then be sent to the Cork table's 'compound mark' character, which was defined for that very reason.

A still better way to do this would be to define a second 'f' in the output font table, which would *not* form a ligature with 'f', 'i', or 'l'. This would give the font the possibility of applying a kern between

[12] The authors would like to emphasize the fact that it is by no means necessary to produce all styles out of the same METAFONT code, as is done for Computer Modern. As a matter of fact the font we are talking about can very well be a mixture of Times, Helvetica, Courier; the advantage of having them inside the same structure is that we can define kerning pairs between characters of different styles.

[13] The figure is actually less than 18, since only lowercase small caps are needed...

the two letters, and counterbalance the effect of the missing ligature (after all, if a font is designed to use a ligature between 'f' and 'i', a non-ligatured 'fi' pair would look rather strange and could need some correction).

Other applications: *Ars Longa, Vita Brevis*

In this paper we have chosen only a few applications of Ω, out of personal and highly subjective criteria. Almost every script/language can take advantage in one or another way of the possibilities of internal translation processes and 16-bit tables. For example, the first author presents in this same conference his pre-processor *Indica*, for Indic languages (languages of the Indian subcontinent (except Urdu), Tibetan and Sanskrit). *Indica* will be rewritten as a set of ΩTPs; in this way we will eliminate all problems of preprocessing TEX code.

All in all, the development of 16-bit typesetting will be a fascinating challenge in the next decade, and TEX/Ω can play an important rôle, because of its academic support, openness, portability, and non-commercial spirit.

References

Betts, G. and A. Henry. *Teach yourself Ancient Greek.* Hodder and Stoughton, Kent, 1989.

Haralambous, Y. "Typesetting the Holy Qur'ān with TEX". In *Proceedings of the 2nd International Conference on Multilingual Computing—Arabic and Latin script (Durham).* 1992.

Haralambous, Y. "The Khmer Script tamed by the Lion (of TEX)". In *Proceedings of the 14th TEX Users Groups Annual Meeting (Aston, Birmingham).* 1993a.

Haralambous, Y. "Nouveaux systèmes arabes TEX du domaine public". In *Comptes-Rendus de la Conférence « TEX et l'écriture arabe » (Paris).* 1993b.

Haralambous, Y. "Typesetting the Holy Qur'ān with TEX". In *Comptes-Rendus de la Conférence « TEX et l'écriture arabe » (Paris).* 1993c.

Knappen, J. "Fonts for Africa". *TUGboat* **14** (2), 104–106, 1993.

Lagally, K. "ArabTEX". In *Proceedings of the 7th European TEX Conference (Prague).* 1992.

Levy, S. "Using Greek Fonts with TEX". *TUGboat* **9** (1), 20–24, 1988.

Imprimerie Nationale. *Les caractères de l'Imprimerie Nationale.* Imprimerie Nationale, Éditions, Paris, France, 1990.

Plaice, J. "Progress in the Ω Project". Presented at the 15th TUG Annual Conference, Santa Barbara, 1994, and published in these Proceedings.

ε-TEX and NTS: A Status Report

Philip Taylor,
Technical Director, NTS project
The Computer Centre, RHBNC
University of London, U.K.
P.Taylor@Vax.Rhbnc.Ac.Uk

Abstract

The NTS project was created under the ægis of DANTE during a meeting held at Hamburg in 1992; its brief was to investigate the possibility of perpetuating all that is best in TEX whilst being free from the constraints which TEX's author, Prof. Knuth, has placed on its evolution. The group is now investigating both conservative and radical evolutionary paths for TEX-derived typesetting systems, these being respectively ε-TEX (extended TEX) and NTS (a New Typesetting System). The group is also concerned that whilst TEX itself is completely stable and uniform across all platforms, the adjuncts which accompany it vary from implementation to implementation and from site to site, and has therefore proposed that a 'canonical TEX kit' be specified, which once adopted could safely be assumed to form a part of *every* TEX installation. Work is now well advanced on the ε-TEX project, whilst the group are concurrently involved in identifying the key components of a complete portable TEX system and in investigating sources of funding which will allow the NTS project to become a reality.

Background

The NTS project first saw the light of day at the Hamburg meeting of DANTE during 1992; prior to this meeting, Joachim Lammarsch had sent e-mail messages asking those interested in the future of TEX to register their interest with him, and invitations to attend the DANTE meeting were sent to those who had registered their interest. At the meeting, Joachim proposed the formation of a working group, under the technical direction of Rainer Schöpf and with membership drawn from DANTE and UK-TUG, whose brief would encompass investigating, and possibly implementing, a successor or successors to TEX: that is, typesetting systems which would embody all that was best in TEX whilst being free from the constraints which had been placed on the evolution of TEX itself. These constraints had been imposed when Knuth announced that his work on TEX was complete, and that he now desired it to remain unchanged, apart from any essential bug fixes, in perpetuity.

Because of Rainer's very heavy commitments on other projects (and in particular on the LATEX3 project), not a lot was accomplished during the first year, and at the meeting of DANTE one year later Rainer announced that he was standing down as technical director; he and Joachim had already asked if I would be prepared to undertake that rôle, to which I had agreed, and therefore with effect from the 1993 Chemnitz meeting of DANTE the original NTS group was stood down and a new group formed.

The Group Meets

The initial membership of the re-formed group was just Rainer, Joachim and myself, and the first task was to identify others who would be willing to devote quite considerable amounts of time to ensuring the success of the NTS project. In the end Peter Breitenlohner, Bogusław Jackowski, Mariusz Olko, Bernd Raichle, Marek Ryćko, Joachim Schrod, Friedhelm Sowa and Chris Thompson were identified as likely candidates and invited to an inaugural meeting to be held in Kaiserslautern. (Barbara Beeton was also invited to join the group, but indicated in her acceptance that she would prefer to be a 'corresponding member', participating by e-mail but not in person.) Not all of those invited could attend, and the inaugural meeting was eventually attended by Peter B., Mariusz O., Bernd R., Joachim L., Joachim S., Friedhelm S. and myself, with Marion Neubauer acting as minutes secretary. The meeting was opened by Joachim L., who explained how NTS had come into existence, and the discussion was then opened for suggestions from those present.

Joachim S. was the first to speak, and he presented some quite radical ideas which he had clearly spent considerable time in preparing. The essence of his proposal was that TEX *per se* was no longer capable of evolution: quite apart from the constraints placed on its evolution by DEK, the complexity of the code and the interdependencies which existed within it militated against any further significant development; indeed, it was Joachim's contention that even Don no longer found it easy to modify the source of TEX, and that if any real evolution was to be achieved, the first task would involve a complete re-implementation. Joachim proposed that this be accomplished in two stages: (1) a re-implementation in a modern rapid-prototyping system such as CLOS ('Common Lisp Object System'), and (2) a further re-implementation in a modern efficient mainstream language such as 'C++'. Both of these re-implementations would be tackled using literate programming techniques.

Joachim explained the rationale behind the two-phase approach: in the first phase, the primary objective would be to identify the true modular nature of the TEX typesetting system, and to factor out into separate modules each of the fundamental components. This would allow, in the future, any component of the typesetting system to be replaced by an experimental version, with minimal impact on the design and interaction of the other modules; through this mechanism, alternatives to the current TEX algorithms could be evaluated and tested. For the initial implementation, however, this flexibility would not be fully exploited; instead, the system would simply be a fundamental re-implementation of TEX using modern object-oriented techniques, within which the modular nature of TEX would be properly represented. Although not expected to be efficient, this re-implementation would provide full TEX functionality, and once working the next step would be to demonstrate the functional equivalence of the re-implementation and TEX; this would be accomplished by means of the standard 'Trip' test for TEX, together with as much additional testing as was felt necessary to demonstrate that it was indeed a true 'TEX'.[1]

Once a faithful re-implementation of TEX had been achieved using the prototyping system, the second phase would involve a further re-implementation, this time using a widely available language such as 'C++'. The modular structure identified during the prototyping phase would be accurately mirrored in the 'production' phase, and each module would be functionally equivalent in both systems. Again a rigorous programme of testing would be required to demonstrate that the production system was also a true 'TEX'; once this was accomplished, the production system would be made available to TEX implementors world-wide, with a request that they attempt a port to the operating system(s) which they supported. Obviously the group would need to be responsive to problem reports from the implementors, and the ported implementations themselves would need to be tested for complete compliance with the TEX standard, but once these steps were accomplished, the project would be ready to move on to the next phase, which would be to release the re-implementation in its production form to the TEX world.

If all of this could be achieved within a reasonable timescale (measured in months rather than years), it was hoped that the TEX world could be encouraged to standardise on the re-implemented production TEX rather than TEX *per se*; for this to be accomplished, the re-implemented system would need to be at least as efficient as present TEX implementations, and equally bug-free. But if these criteria could be met, and if 're-implemented TEX' achieved the universal acceptance which was hoped for, then the group could turn its attention to the next and most exciting phase, which would be to start work on $\mathcal{N_T S}$ itself, using the prototyping system to evaluate alternative typesetting paradigms and algorithms, and subsequently re-implementing the most successful of these and incorporating them into the production system.

Needless to say, the group were impressed by Joachim's proposal: clearly thought out, and quite radical in its approach, it would require considerable resources to be brought to fruition, yet the end results would almost certainly justify the means. However, the practical aspects could not be ignored, and the group agreed that without adequate financial backing they lacked the resources necessary to accomplish even phase-1, let alone subsequent phases. A small group of competent programmers, working full time for two to four months, would probably be needed to accomplish the first re-implementation, and perhaps only slightly fewer resources would be needed to accomplish phase-2. Bearing this caveat in mind, it was decided to put Joachim's plan on ice while seeking funding which would allow its commencement, and to identify fall-back plans which could be accomplished within the present resources of the group.

[1] The group are aware, of course, that only the *presence* of bugs can be demonstrated by testing, never their absence...

Two basic ideas were proposed: (1) despite Joachim's warning that TeX in its present form was essentially incapable of significant further development, Peter Breitenlohner felt that his experience in implementing TeX-XeT and TeX--XeT would allow him to make further changes to TeX within the framework of its present implementation; a number of good ideas had been proposed on the NTS-L list for incremental improvements to TeX, and both Peter and Bernd had ideas of their own which they would like to see implemented. (2) Marek and Bogusław had proposed during a conversation at Aston that before the NTS project sought to extend TeX in any way, it would be a very good idea to ensure that TeX's present capabilities were capable of being exploited to the full; in particular, they felt that TeX was frequently under-exploited because the additional software which was sometimes necessary to fully exploit TeX was not universally available. They therefore proposed that the group specify a minimum TeX kit which should be available at *every* site; once this was known to be universally available, TeX documents could be written to assume the existence of this kit, rather than simply assuming at most TeX + LaTeX + the 75 Computer Modern fonts, which assumption tended to form the basis for portability at the moment.

Both of these proposals were well received: it was agreed that Peter Breitenlohner should take primary responsibility for extending TeX in its present form, whilst Marek and Bogusław would be asked if they were prepared to oversee specification of 'the canonical TeX kit' (it will be remembered that M&B were not present at the meeting, and therefore no assumptions could be made about their involvement in the project). All members of the group would be invited to contribute to all three projects, and members were also asked to investigate possible locations where a small team of programmers could work on NTS. Such a location would clearly need good Internet connectivity (particularly as it was envisaged that not all members of the NTS programming team would necessarily work in the same place), and for reasons of economy it was considered desirable to site them at locations where their day-to-day expenses would not be too great.

Finally it was agreed that the membership of the group could be usefully enhanced by inviting some eminent members of currently unrepresented groups to participate.

The Interregnum

There then followed a period of several months during which members of the group returned to their normal place(s) of work; communication between members of the group was conducted by e-mail, and a report of the inaugural meeting published to encourage outside participation. Invitations to join the NTS project were extended to the nominated national representatives (sadly, not all had the courtesy to reply), and Bernd and Peter attempted to focus discussion on the NTS-L list by proposing some concrete ideas for 'extended TeX' (which was by now referred to simply as e-TeX, following the nomenclature separately proposed by Frank Mittelbach and myself in our *TUGboat* papers on the future of TeX.

During this interval, Joachim Schrod tendered his resignation from the group; all were sad to see him depart, particularly since his proposals for NTS still formed a central element of the group's plans, but the group respected his decision and since then he has pursued his own independent research into literate programming, typesetting systems, and the many other fields which are of interest to him professionally. Jiří Zlatuška was nominated by the Czech TeX User Group (CSTUG) to represent them on NTS, and Klaus Lagally and Richard Palais accepted an invitation to join, the latter electing to follow Barbara's example and be a 'corresponding member'.

The Second Meeting

In February of 1994, DANTE held a further meeting, this time at Münster in Westfälische, and for a third time the NTS group were invited to be DANTE's guests; on this occasion none of the Polish delegates were able to be present, but Jiří represented the Czechs and Volker Schaa very kindly stepped into the breach to act as minutes secretary, Marion being unable to attend. Others present at the meeting included Peter Breitenlohner, Joachim Lammarsch, Bernd Raichle, Rainer Schöpf, Friedhelm Sowa and myself.

The group had made some promising contacts concerning possible funding for the project, but nothing definite had been agreed and commercial confidentiality regrettably dictates that no further detail can be recorded here. Following a very useful contact made at SOFSEM '93 (the Czech and Slovak annual computer science conference), some discussion had taken place concerning the possible use of

a semi-automatic system for reverse-engineering TeX, but again nothing had been agreed at the time of the meeting. Marek and Bogusław had been unable to offer a committment to leading the 'canonical TeX project', and no further progress had been made in that area, despite (or perhaps because of) a counter-proposal by TUG who had somewhat belatedly decided to undertake an almost identical scheme.

By far the most promising news was that Peter Breitenlohner had made enormous progress in the implementation of ε-TeX, and indeed had a preliminary version already working. In view of this, and in view of the fact that the membership of the group had undergone some very significant changes, it was decided to adopt a slightly more formal structure within the group: Joachim Lammarsch would continue to take overall financial and political responsibility, whilst I would continue as Technical Director; Peter Breitenlohner would head the ε-TeX sub-group, backed up by Bernd Raichle, whilst Jiří Zlatuška would head the NTS sub-group, again backed up by Bernd; Rainer Schöpf would take responsibility for the 'canonical TeX kit' project, backed up by Friedhelm Sowa, and in addition Friedhelm would continue to act as treasurer. No formal responsibilities were laid on members who were not present at the meeting, although it was hoped that Mariusz Olko would continue to look after the multi-lingual areas of both ε-TeX and NTS by liaising with the TWG-MLC.

As progress on the 'canonical TeX kit' project had been almost non-existent, the group felt it worthwhile to devote some time to attempting to identify the elements of a TeX system which were truly fundamental. Rather interestingly, members seemed to hold stronger (and sometimes more divergent) views on this subject than on almost anything proposed for either ε-TeX or NTS! The net result was that only the most basic elements were agreed, and considerable further work will be needed in this area in conjunction not only with TUG but also with the entire TeX community.

Much of the discussion which took place during the NTS meeting at Münster concerned specific details of proposals for ε-TeX, and whilst these formed the basis for progress, later discussions (both in less formal meetings at Münster and subsequently via e-mail) caused considerable revision of the ideas which emerged; what follows is therefore a synthesis of ideas which were first mooted at the NTS meeting in Münster, together with ideas which were mooted later, either in less formal meetings or via e-mail.

ε-TeX: Some Specific Proposals

Perhaps the most important of the proposed extensions is the mechanism by which the extensions themselves are activated, either individually or as a group; an absolutely fundamental requirement is that ε-TeX be capable of processing all existing TeX documents in a manner identical to TeX. The typeset results must be identical to those produced by TeX, as must all 'reasonable' side effects (for example, information written to ancillary files). Thus it is intended that there be no possible reason for the non-adoption of ε-TeX as a replacement for TeX. The mechanism by which divergent behaviour is enabled will be under user control: a user may elect to use ε-TeX in a totally compatible manner, or may elect to use only that set of extensions which do not compromise the semantics of TeX, or may elect to use one or more of the most radical extensions (e.g. TeX--XeT, the left-right/right-left extension for non-European languages) which by their very nature require a fundamental modification to the behaviour of the typesetting system.

Once such a mechanism is in place, users (or more precisely user documents) will need to be able to investigate their environment; since user-X may habitually use extension-A, yet may send his/her document in source form to user-Y who habitually disables extension-A, a document must be able to check which extensions are available, and to either adopt a fall-back position if a preferred extension is not available, or to issue an error message and abort tidily (a user document may, of course, attempt to enable an extension which it needs, but the result may be 'not available within this environment'). A mechanism for environmental enquiries is therefore proposed.

Other proposals for ε-TeX include: (a) improved control over tracing (TeX can be very verbose, and interpreting the trace output is distinctly non-trivial); (b) an additional class of maths delimiters (middle, as well as left and right); (c) improved access to the current interaction mode (to allow code to ascertain as well as change the current mode of interaction); (d) improved mechanisms for checking the existence of a control sequence without necessarily creating such a sequence; (e) improved avoidance of internal integer overflow; (f) an alternative ligature/kerning implementation; (g) extensions to the set of valid prefixes for macro definitions, such as `\protect` (to inhibit subsequent undesired expansion) and `\bind` (to render a definition independent of the environment within which it is expanded); (h) support for colour.

NTS

The complete re-implementation of TeX ('NTS') is a far more ambitious project, the success of which is crucially dependent on obtaining adequate funding. The proposals which follow should therefore be regarded as being preliminary ideas, rather than absolute decisions which have been cast in stone. Any ideas which the TeX community in general would like to contribute to the project will be very gratefully received!

In phase-1, a rapid prototyping language such as CLOS ('Common Lisp Object System') or PROLOG will be used to develop a primary re-implementation, within which it will be possible to experiment with various possible internal modular representations of the present TeX typesetting engine; these alternative representations will be evaluated to attempt to determine an 'ideal' model of the TeX engine, within which the various functional elements are as independent as possible. The purpose of this analysis is to allow each of these functional elements to be enhanced or replaced at will, with minimal effect on the other modules; such flexibility is demonstrably lacking in the present TeX implementation, and is the primary reason for proposing a complete re-implementation. Once an ideal representation has been found, the work will progress to the second phase, although the work invested in developing the first phase will not be wasted: there are many reasons why the phase-1 implementation will continue to be both needed and useful.

Firstly, prototyping languages based on term data structures, used in conjunction with a disciplined programming style free from the exploitation of side-effects, allow the generation of code which would provide a specification of TeX independent of the procedural Pascal source code, and potentially amenable to the use of automated transformation techniques; such code could also be partially interpreted or meta-interpreted. This would produce an environment within which the development of enhancements is much facilitated, reaching beyond the particular structure of the code of the program.

Secondly, for some of these prototyping languages at least, rather powerful enhancements have been developed, in particular the employment of constraint solving in constraint logic programming languages. This could allow certain parts of the typesetting task to be formulated as a set of constraints to be fulfilled, within which constraints an optimal solution must be found. Certain classes of problem, particularly those occurring in connection with chapter layout optimisation, float/insertion placement, and conformance to the grid, all appear to require a search of the space state and the employment of backtracking within certain parts of the typeset text.

Thirdly, the kind of test-bed implementation proposed for phase-1 should provide a suitable basis for the independent exploration of possible enhanced designs and/or implementations, including alternative attempts to provide solutions based on different approaches and paradigms, and also including investigations into the trade-offs involved when selecting from a set of mutually incompatible features.

In phase-2, the modular structure which has been identified in phase-1 will be once again re-implemented, this time in a widely available compiled language such as C++. This re-implementation is aimed entirely at efficiency: the test-bed which is developed during phase-1 is fully expected to be unacceptably inefficient as a production typesetting system, and the flexibility which will characterise it is unlikely to be of any use in a production environment. Thus in phase-2 attention will be paid to the efficiency of the algorithms used, and particular attention will be paid to ensuring that the resulting system is able to run efficiently on a very wide range of hardware and software platforms. Whilst ε-TeX is expected to run on the same range of hardware and software as the present TeX system (i.e., everything from an AT-class PC to a Cyber or Cray mainframe), it is accepted that NTS may have slightly reduced availability: none the less, the minimum configuration on which NTS will be expected to run efficiently is an 80386-class PC (or the equivalent in other architectures); everything above should pose no problems.

Both the phase-1 and phase-2 implementations, in their initial release, will be purely and simply re-implementations of TeX (or, more likely, of ε-TeX, since it is hoped that by the time phase-1 is complete, ε-TeX will have gained widespread adoption amongst the TeX community; the intellectual effort invested in extending and enhancing TeX to create ε-TeX will therefore not be wasted). These re-implementations will be rigorously tested to ensure that they behave identically to (ε-)TeX in all circumstances, and only once this testing is complete will they be offered to the community at large.

Once the project has reached this stage, and the phase-2 re-implementation has been made available to, and used by, a significant proportion of the existing TeX user community (and any other interested parties), the real research work will commence: using the phase-1 re-implementation (the 'test bed', as

it is generally termed), work will commence on investigating alternatives to the existing TeX paradigms. For example, one of the frequent criticisms levelled against TeX is that its command line oriented user interface seems unbelievably anachronistic to those whose experience of computers is almost entirely limited to graphical user interfaces (GUIs); whilst various implementors (and most notably Blue Sky Research, with their 'Lightning TeXtures' implementation) have endeavoured to conceal this interface beneath a more graphical front-end, all of these attempts have been restricted to one particular hardware and software platform. A primary longer-term objective of the $\mathcal{NTS}$ project would therefore be to provide an integral graphical interface to $\mathcal{NTS}$ in an entirely portable manner; this interface would not be layered on top of the existing command line interface, but would operate at the same hierarchical level, thereby allowing the more GUI-oriented to use $\mathcal{NTS}$ in their preferred manner with no loss of efficiency (the existing command line interface would not necessarily disappear: the group are well aware that one reason for the widespread adoption of TeX may well lie in the appeal of its more traditional interface to large numbers of people who pre-date the GUI era, as well as in the re-creatability and reproducibility of documents which are represented as human-readable text rather than as ephemeral mouse movements on a screen).

There are many other areas in which TeX is felt to be deficient by a significant group of well-informed users; various papers have been published in *TUGboat* in which some of these deficiencies have been discussed. It is therefore proposed that NTS would attempt to rectify as many of these deficiencies as possible by providing some or all of the following features: (a) the ability to typeset material on a grid; (b) the ability to flow text around regular (and irregular) insertions; (c) the treatment of 'the spread' (two facing pages) as the basic unit of makeup; (d) the treatment of the chapter (in book-like material) as the minimum unit over which page optimisation should be performed; (e) provision of pattern recognition within the paragraph-building algorithm, which could enable both the avoidance of 'rivers' (accidental contiguous regions of regular white space spanning several lines) and also (at a less graphical level) the avoidance of subtly different hyphenations on consecutive lines; (f) improved interaction between the line- and page-breaking algorithms, to permit co-optimisation; (g) an improved model of lines within a paragraph, to enable languages which make regular use of diacritical marks to be set on a tighter leading (interline spacing) without increasing the risk of conflict between superior and inferior diacritics; (h) intrinsic support for hanging punctuation (whereby leading and trailing punctuation on a line are allowed to hang out into the margin); (i) improved interaction between the page makeup module and the paragraph building module, to allow the shape of a paragraph to be influenced by its final position on the page; (j) greater awareness within the typesetting engine of the shape of a glyph (character), which could allow spacing to be better optimised; (k) improved parameterisation of fonts; (l) improved access to the ligature and kerning information from within the typesetting system. There are many other areas, primarily concerned with rather technical aspects of TeX-the-language and TeX-the-typesetting-system, which have also been convincingly argued are less than perfect, and which it is intended will be addressed by $\mathcal{NTS}$.

The long-term objective of $\mathcal{NTS}$ is therefore to make maximum use of the test bed to investigate and evaluate possible approaches to overcoming the various perceived deficiencies of TeX, and to incrementally produce an ever-better typesetting system, capable of taking maximum advantage of current technology. This typesetting system will undoubtedly become ever less TeX-like, yet the group believe that there are so many good ideas enshrined in TeX that the day when $\mathcal{NTS}$ owes nothing to TeX lies several decades in the future.

The author would like to express his sincere thanks to Peter Breitenlohner and Jiří Zlatuška for their most helpful comments on the first draft of this article, and to Michel Goossens for his willingness to allow late and unrefereed submission of the copy; any errors remaining are, of course, entirely the responsibility of the author.

TEX innovations at the Louis-Jean printing house

Maurice Laugier
General Director of ILJ, Imprimerie Louis-Jean, B.P. 87, 05003 Gap Cédex, France
`louijean@cicg.grenet.fr`

Yannis Haralambous
Centre d'Études et de Recherche sur le Traitement Automatique des Langues
Institut National des Langues et Civilisations Orientales, Paris.
Private address: 187, rue Nationale, 59800 Lille, France.
`Yannis.Haralambous@univ-lille1.fr`

Abstract

In this paper we will present several TEX innovations, conceived, or currently under development, at ILJ (the Louis-Jean printing house). ILJ was founded in 1804, at Gap (Southern French Alps) and employs 130 people. Due to its specialization in typesetting and printing of scientific books, ILJ has been using TEX since the late eighties. Currently about 30% of ILJ's book production is done with TEX. New developments in the TEX area sponsorized or financed by ILJ are described in this paper.

This paper will be published in the next issue of *TUGboat*.

Design by Template in a Production Macro Package

Michael Downes
American Mathematical Society, 201 Charles Street, Providence, Rhode Island 02904 USA
mjd@math.ams.org

Abstract

The American Mathematical Society has been involved in the development of TeX from the beginning and began using it for journal production ten years ago; we now produce nearly all of our publications (a couple of dozen journals and book series) with TeX, using AMS-developed macro packages. One of the goals set for a major overhaul of the primary in-house macro package, begun in 1992, was to make revisions to the visual design of a given publication easier. In the new version of the macro package the design specifications for a particular document element (such as an article title) are not embedded in TeX code, but are entered into an element specs template that is comparatively easy to read and modify, and that corresponds more directly to traditional book design specs (e.g., vertical spacing is expected to be given in base-to-base terms).

Introduction

Some of the terms used herein have a specialized meaning in the publishing industry; two that should be mentioned in particular are *composition*, meaning the general process of composing characters into words, paragraphs, and pages (historically done by setting lead characters in type frames, nowadays done with software), and *design*, meaning the visual style and physical layout of a book or other publication, including choice of fonts, dimensions of the type block, and arrangement of document elements. Publishers employ freelance or in-house *publication designers* (more commonly known as *book designers*) to analyze authors' manuscripts and devise appropriate designs.

The transition of the publishing industry in the last few decades to working with electronic documents was impelled initially by the desire for more efficient production of traditional printed forms. It has become clear by now, however, that the electronic document should be the *primary* goal of authors and publishers; and moreover, that documents in information-rich formats such as SGML are many times more valuable than documents limited to a single medium and visual format. The challenge now for compositors is to make composition software that enables the composition process to be driven more directly by an electronic document, when the content and structure of the document are adequately marked. Given a visual design in suitable form, the typesetting operations to be applied can in principle be deduced from the information present in the document. If software can do this task, it will render unnecessary some of the expensive rote work traditionally done by human copyeditors when marking up manuscripts with instructions for the compositor. The copyeditor served as a sort of interpreter between the author and the compositor, who were not expected to understand each other's language. And the book designer may be thought of as the linguist who wrote the bilingual dictionary used by the copyeditor in doing the interpreting.

Book designers use a high-level, rather informal language that has evolved over the last few centuries together with printing technology and methods. By 'informal' I mean that traditional book design specifications aren't sufficiently detailed and well-structured to be directly interpreted by typesetting software, even after doing the obvious streamlining of vocabulary and syntax. In the past the work of translating book designs into suitable typesetting operations was done by skilled compositors who brought to the translation process a great deal of enriching knowledge and craftsmanship. In recent years, the computerization of typesetting has shifted more and more of that knowledge and craftsmanship into typesetting software.

But the state of the TeX world today is that not enough of the knowledge and craftsmanship has been transferred to the software. There is a wide gap between the customary design specs that publishers pay designers for, and the actual application of a design to the pure information content of a document by a TeX macro package. Outside the sphere of TeX, this gap is usually closed by idiosyncratic stylesheet capabilities of commercial publishing software ranging from word-processors to high-end publishing systems. The internal format used for holding style information, the internal typesetting operations used to apply the style information, and the underlying analysis of document design are proprietary information, and I imagine that not too many software companies are rushing to make theirs public.

The language used in FOSIs[1] is the only major, publicly accessible attempt I know of to formalize visual document design into a standard language suitable for automated typesetting. And as it is, FOSIs only deal with abstract specifications; you still have to make your typesetting software understand the elements of a FOSI and do the right thing with them. An extended discussion of using TEX and FOSIs together appeared not long ago in *TUGboat* (Dobrowolski 1991).

In the TEX world the gap between design and actual typesetting is usually bridged only by the brow-sweat of a skilled TEXnician. Consider these design specifications for an article abstract:

> **AB:** Abstract. Body text 8/10 Times Roman justified × 27 picas, indented 1 1/2 picas, para indent 1em. Heading "Abstract" and period set 8/10 Times Roman bold, flush left × 27 picas, followed by N space, run in body text. 18 points base-to-base above and below.

As written in typical TEX macro code, the AB element described above might look roughly like this:

```
\def\abstract{\par
  \ifdim\lastskip<\medskipamount
    \removelastskip \medskip
  \fi
  \begingroup \parindent 1em
    \leftskip=1.5pc \rightskip=\leftskip
    \typesize{8}{10}\justify
    \noindent {\bf Abstract.\enspace}}

\def\endabstract{\par \endgroup
  \ifdim\lastskip<\medskipamount
    \removelastskip \medskip
  \fi }
```

In LATEX you could avail yourself of the `list` environment to simplify the task. But it remains clearly a TEX macro writing task, needing to be done by someone familiar with TEX.

Given that the limits of book design are scarcely less wide than the limits of human visual imagination, it's unlikely that custom programming will ever disappear from the picture; unprecedented demands will always require new solutions. Nevertheless, the majority of scientific and academic publications have scholarly communication rather than visual design innovation as their *raison d'être*, and hence are characterized by relatively sober designs with many routine aspects suitable for automation. Most typesetting software doesn't do as much as users would like to make the routine aspects easy to deal with:

> The frequent need for new or differently formatted entities presents a serious problem. It is typical, that either there are too many limitations on the kind of formatting that can be prescribed, or else the formatting prescriptions are very difficult to write and comprehend. — Bo Stig Hansen (1990)

TUGboat articles about the Lollipop macro package by Victor Eijkhout (1992), and the ZzTEX macro package by Paul Anagnostopoulos (1992), show something of what is possible with TEX.

The design-to-typesetting gap needs to be bridged from two directions: from the designer end, by a careful analysis of document design, leading to a formalized (to the extent possible!) language for prescribing design elements and rules, independent of any particular typesetting software;[2] and from the TEX end, by more powerful and flexible macros that match up with the design analysis (along the lines of LATEX's `\@startsection` and `\@hangfrom`). If these two pieces are done well, then the process of transferring a document design from the designer's mind to the typesetting software can be made very easy.

In this paper I discuss some methods being employed to bridge the design-to-typesetting gap in a large TEX macro package, developed over the past two years for use at the American Mathematical Society as the in-house, production-oriented version of the publicly available AMS-TEX package. As the new package has no established name (we've been just calling it 'the new production system') I suppose I'd better define a name of convenience for use hereinafter. Let's call it DBT: design-by-template system. Although there's more to it than the visual design-related features, that should serve well enough.

The goals of DBT, as compared to its predecessor, include:

1. improvements in document markup to enrich and regularize the information content of documents passing through the AMS production system;
2. more sophisticated formatting routines to solve fundamental TEX typesetting problems;
3. change management;
4. a 'fill-in-the-blank' system for specifying the visual design of a given publication through simple variable assignments rather than through TEX macro programming.

As an example of the second item, the page-breaking routines in the new macro package automatically allow a page to run half a line long or short to accommodate the deviation from the text grid that occurs more often than not in pages containing many

[1] FOSI: Formatted Output Specification Instance. Part of the U.S. Department of Defense CALS initiative, see Mil. Std. MIL-M-28001B.

[2] Here I echo the call for a 'front end programming language for style design' of Victor Eijkhout and Andries Lenstra (Eijkhout and Lenstra 1991).

```
\thm{9} The elegance of a mathematical
result is equal to the quotient of power
and length: $E = P/L$.\endthm
```

Figure 1a: Document source text for a theorem.

Theorem 9. *The elegance of a mathematical result is equal to the quotient of power and length:* $E = P/L$.

Figure 1b: Output of the given theorem.

```
[THM]description:{theorem}
[THM]typesize:{10}
[THM]linespacing:{12}
[THM]font:{it}
[THM]wordspacing:{\normalwordspacing@}
[THM]case:{\normalcase@}
[THM]justification:{%
  \fulljustification@{\colwidth}}
[THM]paragraphshape:{\indented@{18pt}}
[THM]breakbefore:{\badbreak@{1}}
[THM]spacebefore:{%
  1.5\linespacing plus.25\linespacing}
[THM]breakafter:{\goodbreak@{1}}
[THM]arrangement:{R head * . {\enskip} body *}
%    Subcomponent 'THM head':
[THM head]font:{bf}
```

Figure 2: Representative specs (pretending no defaults or inheritance) for a theorem element.

math formulas. We have also gone through some rather extensive experiments with nonzero mathsurround.

The focus of this paper is on the fourth item: How to make the process of creating and changing publication designs easier. It seems best to begin with an example, to serve as a point of reference for later discussion. Figure 2 exhibits a representative set of DBT specs for a theorem element. In practice many of the style variables would be given default values (such as 'inherit from context') by omitting the corresponding lines.

For comparison Figure 1 shows a portion of a document file and the output that would be produced given the specs in Figure 2. The main point is that the document file has only information content, not visual formatting instructions, and all the formatting specified in Figure 2 is applied automatically by the software.

Style variables. Here are descriptions of the style variables currently recognized by DBT for major text elements. It should be fairly obvious that this is *not* a universally sufficient set; rather, it is a set that seems to be more or less sufficient for the relatively modest design needs of most AMS journals and books.

description mandatory; used in printing documentation
placement floating or nonfloating
otherbefore catch-all hook (historically this has been used for making things work where the implementation hasn't caught up yet; thus tending to disappear later when the implementation does catch up)
typesize Self-explanatory
linespacing Self-explanatory
typeface Typeface name such as Garamond or Palatino
font One of: rm, bf, it, sc, bit, ...; for simplicity each style/weight/width combination is addressed by a single distinct name.
wordspacing name of a function that sets all relevant parameters
case upper, lower, normal
justification full, raggedright, raggedleft, centered, ...
paragraphshape indented, hangindented, ...
interparspace extra amount, usually 0
breakbefore bad break or good break; the TeX range 0–10000 is collapsed to 0–10
spacebefore base-to-base
actionbefore inner hook
arrangement how subcomponents are combined
actionafter inner ending hook
breakafter cf breakbefore
spaceafter cf spacebefore
otherafter catch-all ending hook

The set of variables is extensible in the sense that a new variable can be freely added for any element, and the assignment will be stored in proper form with the other specs for the element; it's just that you would also have to add some internal TeX processing to do the right thing with the new variable, in order for it to have any effect.

In addition to the above variables that are applied for elements within the text stream, there are a number of global variables that address page layout and other aspects. These are simply handled as TeX integer or dimension variables. Some examples:

```
\typeblockwidth    \linesperpage
\typeblockheight   \trimwidth
\runheadspace      \trimheight
\runheadheight     \headmargin
\textwidth         \guttermargin
\textheight        \dropfoliodepth
```

Features and Limitations of DBT

Concentration of visual design information in a single location. All the visual design and layout aspects of a given publication are kept in a single file

with an extension of `.pds` (publication design specifications). This file is editable ASCII text, like TeX documents, with the same advantages and disadvantages. An added interface layer with menus, context-sensitive help, and *tsk-tsk* sound effects for bad design decisions would be nice but we haven't done anything of that sort yet.

Minimized redundancy. Publications with similar designs can share all the style variable settings that coincide by putting them in a common file to be 'inherited' via TeX `\input` statements. This is important for AMS use because we have several designs that differ only in a few aspects. If each design were kept as a separate full copy, maintenance would be more difficult.

Separation of visual style concerns from information transfer. A large part of some FOSIs that I've seen is taken up with specifying how certain information should be moved around (such as running heads or table-of-contents information). In DBT such information transfer is kept separate from visual style specs. In a `.pds` file there is nothing to say what information should go into the running head; only 'if such-and-such information happens to turn up, here is how to format it'.

Style template. The style of a given element is almost entirely specified in the element 'style template' — nothing more than a list of assignment statements for style variables — and is applied by a generic element-printing function. Little remains to be done by TeX macro writing.

A small number of generic element-printing functions. Essentially four generic element-printing functions are required — one for major elements (slices of the vertical text column), one for floating elements, one for displayed equations, and one for minor elements (distinct logical entities within paragraph text).

Predefined functions for subordinate typesetting tasks. Some of the variables in the element style templates are intended to take on function values: justification (ragged right, ragged center, full justification, etc.), paragraph shape (indented, non-indented, hang-indented, etc.), word spacing (default, 'french', loose). The idea is that a new function should be created whenever existing functions fail to provide the desired style in a form that can be easily called in an element template.

Compact notation for subcomponent handling. The 'arrangement' variable in the element style template is a special variable whose value is a description of the subcomponents (optional or mandatory) that an element may have, and how they should be combined. The notation is effective for concisely describing what is to happen when optional subcomponents are absent.

For the THM element in Figure 2, the arrangement is

```
R head * . {\enskip} body *
```

Each arrangement has seven parts. The first part is the arrangement name. `R` in this case is shorthand for 'run-in'; there are also `H` for horizontal and `V` for vertical. Less common arrangements have fully spelled out names. The second and sixth parts are the names of the subcomponents that are to be combined — here, `head` and `body`. The fourth and fifth parts are in-between material, loosely speaking punctuation and space, respectively. If one of the two components in an arrangement is optional, and absent in a particular instance, the in-between material is omitted.

Arrangements can be combined recursively. The third and seventh parts of an arrangement are slots for fitting in subordinate arrangements. A subordinate arrangement is enclosed in braces so that it can be read as a single macro argument by various arrangement-scanning functions. For example, suppose that we wished to allow an optional note component in the THM head, like the `[note]` option of LaTeX's theorem environments. The arrangement for THM would be expanded by replacing the * after `head` with a second-level arrangement:

```
R head {H mainhead * - {\ } note *}
  . {\enskip} body *
```

A hyphen for part four or five means 'null', in this case 'no in-between punctuation'.

Thus an arrangement is a sort of binary tree of subcomponents. Although restricting to binary combinations requires thinking up more component names than might otherwise be the case (viz the introduction of 'mainhead' above), higher-order combinations are unable to handle an optional middle component without ambiguity. Consider rewriting the above example as a three-way combination:

```
head R - {\ } note H . {\enskip} body
```

If the note is omitted, it's not clear which pieces of the in-between material should be used between the head and the body. Simple strategies such as 'use the first set of in-between material and ignore the second' (or vice versa) proved to be unreliable when I tried them with a range of real examples.

One current limitation of the arrangement notation has to do with list-like arrangements, where an element consists of an indeterminate number of identical subcomponents. An example might be a list of author names and addresses. Such arrangements are sufficiently binary in nature to have no ambiguity problems, but I'm not sure how to extend the current notation to handle them.

Arbitrary number of subcomponents for elements. Elements can be broken down as far as necessary to yield the desired level of independent control over fonts and other aspects of style for each component. A typical THM arrangement is slightly more elaborate than the one in the example given earlier, treating the word 'Theorem' and the number as separate components. The most complex element I've had to deal with so far consisted of four arrangements eight components.

By suitable combination of simpler arrangements, the design for a given element can become an arbitrarily complex two-dimensional structure.

Inheritance. If one element has nearly the same style as another, there is a way in a DBT template to specify that the element is 'based on' the other, and reset only the style variables that have differing values.

Variable clustering. Instead of having separate entries in the element specs template for every variable provided by TEX, and every variable added by the macro package, some of the entries reflect clusters of related variables. To get ragged-right you need just one line:

```
[XX]justification:{\raggedright{30pc}}
```

instead of many lines:

```
[XX]hsize:{30pc}
[XX]leftskip:{0pt}
[XX]rightskip:{0pt plus3em}
[XX]parfillskip:{0pt plus1fil}
[XX]exhyphenpenalty:{3000}
[XX]hyphenpenalty:{9000}
[XX]pretolerance:{200}
[XX]tolerance:{400}
```

I've vacillated about leaving `\hsize` as a separate parameter, perhaps under a different name. The reasons for folding it into the justification parameter are: (a) this corresponds well to the way justification is specified in traditional book designs; and (b) there is a check in the internal processing of the generic element-printing functions to see if a new value for justification is the same as the previous value; if so, the resetting operation can be skipped. If `\hsize` were a separate parameter, the check would have to test two variables instead of one to decide whether the skipping can be done.

Some aspects of style templates remain TEXnical. Although the syntax of element style templates has been intentionally deTEXified, towards the goal of making them accessible to nonTEXnicians, the example in Figure 2 exhibits some backslashes, curly braces, and (gasp) even @ characters. The main reason for this was convenience during the development phase. Further improvements to the syntax would not be that difficult but have not yet reached high enough priority.

There were two reasons for allowing private control sequences: first, it leaves open the door to enter arbitrary TEX code in the value of a variable (though in our experience so far this has not been needed as much as I expected); and second, it avoids name clashes for things like `\goodbreak` and `\uppercase` that seemed natural for certain values.

Another significant practical constraint was parsability. Currently all design specs are parsed and assimilated at run-time. Though it may not be obvious at first sight, behind the syntax shown in Figure 2 lie many rejected variations that would have made it much more difficult to write the routines that scan component names and variable values. For example, use of anything other than curly braces to delimit variable values would lead to various problems: if end-of-line is used to mark the end of the value, then multiline values become difficult to deal with; if parentheses are used as delimiters, then it becomes difficult to use parentheses in a variable's value; and so on.

Printing out written specs from the `.pds` file. The organized structure of element arrangements allows them to be traversed by a suitable function in order to print out a transcription (which comes closer than you might expect to traditional specs written by a human designer). Vertical spacing values are not only entered in base-to-base form, but also stored that way, so printing out the values does not require backward conversion. The application of the specified vertical spacing is highly accurate by virtue of complicated internal code, which is beyond the scope of this paper.

Black-box math. Math style is dealt with in a more-or-less black-box manner: Here is a whole math setup, take it or leave it. As a matter of fact, the knowledge necessary for high-quality math typesetting has traditionally resided more in the hands of compositors than in the hands of book designers, so the current sketchy treatment of math style in `.pds` files is wholly typical. Although it would be good to open up math style for easy access, the necessary work hasn't been attempted yet. Note that the variables needed to control math style are almost completely different from the variables needed for normal text elements.

Memory hogging. Compared to other macro packages DBT is a memory hog, easily going over 64K in TEX's main memory category before even starting the first page, not to mention loading any hefty extension modules like a table package. And this is after moving error message and help message texts out of macro storage onto disk. In general I followed the philosophy of the authors of the X window system: Write the software the way you think it ought to be done, ignoring the fact that available computer

resources are strained to support it, and hope that the resources will catch up by the time the software reaches maturity. Some further routine optimizations can still be done but it would be premature to do them now.[3]

Slowness. And it runs slowly, because it's trying to be so clever and do everything the right way, rather than the expedient way. For example, the reason that `.pds` files are scanned at run-time rather than precompiled, is that this puts a part of the maintenance burden where it ought to be — on the computer, rather than on the persons who set up design specs; an extra compilation step would make development and testing more onerous. Rumors that we would get a super-fast new machine on which to run in-house TeX production have not yet been substantiated by putting it in our hands. At the moment, running on a not-too-shabby two-year-old Unix workstation (circa 20 Specmarks, 30 MIPS), a typical book run of around 300 pages may take more than two hours, as opposed to thirty minutes or so with the macro package that preceded DBT. (That includes some non-TeX overhead such as `dvips` processing, and with the workstation simultaneously serving other processes.)

Various Complications

In this section I want to describe some of the technical complications and problems that have crossed my path. A few of them are TeX-specific, while others are relevant for any system that seeks to automate the application of document designs.

(1) Intent: Style variable values should be carefully specified to reflect the designer's intent, as much as possible, rather than the results of that intent. For example, suppose that the designer calculates the point values for space around a section head to make the head occupy exactly two lines of space at 12pt linespacing, setting the `spacebefore` and `spaceafter` variables to 21pt and 15pt respectively. If the linespacing subsequently changes to 13pt, the values of `spacebefore` and `spaceafter` need to be updated by hand, whereas if they had been set to `1.75\linespacing` and `1.25\linespacing` then the change from 12pt to 13pt could automatically propagate as desired.

(2) When to evaluate: It's not always desirable to fully evaluate the value of a variable at the time of assignment. Immediate full evaluation makes it impossible for a dependent variable to keep in sync with another variable as it changes. For example, if the space above a section head is specified in relation to linespacing, and the normal linespacing value is overridden later to produce a double-spaced proof copy of a document, you would probably want the section head spacing to change proportionally. This won't happen if the spacing value was fully evaluated at the time of first assignment. Compare the well-known distinction in computer science between passing function arguments by value or by reference.

(3) Inheritance: When specifying the style of two similar elements, the one with a more complex subcomponent structure should be based on the one with a simpler structure, rather than vice versa. This is almost self-evident but I once had to set up four or five unnumbered footnote-type elements, and in my first attempt I thoughtlessly spec'd the normal numbered footnote first and based the unnumbered elements on that, before realizing it ought to be done the other way around.

(4) Inheritance: The question of immediate versus delayed evaluation for individual style variables applies also to collections of variables, when a major element is specified to be based on another. If the similarity of style is coincidental rather than due to a logical relationship between the two elements, then immediate evaluation would probably be desired.

(5) Documents that are nominally supposed to have the same documentstyle, in practice often don't. For a book series, the primary design for the series is typically subject to minor modifications in individual volumes. For example, in one volume the style of the footnote marks was changed because they too closely resembled some elements of the math formulas in the volume.

Similarly certain kinds of style variations are routinely permitted between different articles within a journal issue. In an article where a bullet • is used as a math symbol, it would probably be a good idea to override a standard list style that marks unnumbered list items with bullets.

Different authors prefer different numbering schemes, and because the numbering scheme of a document is closely bound up with the logical structure of the document, we routinely allow style for theorem heads and section heads to vary (within certain standard limits) to better suit the numbering scheme, rather than rigidly enforce a single numbering scheme. Here are three of the most common variations in theorem numbering:

Theorem 1.1. *Normal.*

1.2. **Theorem.** *Swapped numbers.*

1.3. *Numbers only.*

Numbers are typically swapped to the front when there are many numbered elements and different element types share the same numbering sequence. The font and intervening space then change for design reasons. All of the variations can be produced from identical markup by style override statements at the beginning of the document.

[3] "Premature optimization is the root of all evil." Donald Knuth, `tex.web` (version 3.14, 1991, §986).

(6) In fact, different articles in a journal frequently differ not only in style aspects but in the very elements that they contain. In a recent example, an article contained two different kinds of proofs instead of the normal one kind: proofs given in full, and brief sketches that left the details to be filled in by the reader. The two kinds were marked by two different QED symbols.

As anticipating all possible elements and forbidding unknown elements are equally impractical, the solution seems to be to make it easy to declare a new element and its associated style, and put those declarations into the individual document where needed.

(7) Arrangements in DBT are applied only to subcomponents within a single element. But documents may also contain sequences of major elements where some of the elements themselves are optional, which leads to the same sort of potential ambiguities as with subcomponent arrangements. Suppose the opening of an article consists of title, author, dedication, key words, subject classification, abstract, table of contents, and finally, main text. And suppose that the dedication, key words, abstract and table of contents are optional. It gets to be rather tricky to specify the vertical spacing to be used in all possible combinations. In DBT this is handled mainly by application of a single vertical spacing rule: when adding a major element to the page, compare the space after the preceding element and the space before the new element and use whichever is larger. Practically speaking this seems to suffice most of the time, if care is taken in choosing where the various space values are specified (e.g., for a given set of space values it may work out better to leave the `spaceafter` variable for the author element at zero and rely only on the `spacebefore` values of all the components that could possibly follow after the author). There is a rudimentary mechanism in DBT for specifying interelement space depending on the type of the two elements, but it hasn't been needed much.

(8) Communication between information-handling and design-handling functions: In DBT a TEX command such as `\thm` is used in a document to collect the contents of a theorem. `\thm` sends this data to a generic element-printing function that takes the given pieces of information and feeds them into the declared arrangement for theorem elements. Thus the definition of `\thm` is interdependent with the arrangement. Ideally the definition of `\thm` could be derived in some semi-automatic way from the arrangement structure but there are many complications, so at present DBT doesn't attempt to be too clever; someone has to explicitly define all the components that `\thm` should look for, and the document syntax to expect. There are high-level syntax-related functions that make this task fairly easy, but the person doing the defining has to actually look at the declared arrangement when setting up the parallel structure in the `\thm` command, rather than having any of the transfer done automatically.

(9) Information content for the components of an element can be provided either explicitly in the document or by giving a default value in the element template. For example, a proof head will normally be given a default value of 'Proof' in this manner:

```
[PRF]arrangement:{%
  R head * . {\enskip} body *}
[PRF]head:{Proof}
```

In DBT there is a mechanism for optional substitution of alternate text for such a component, for selected instances of the parent element in a document. This is needed occasionally to get an alternative proof head such as 'Sketch of the Proof'.

It's not clear whether default information content specifications like this should be lumped together with style specifications. Is the material content or style? I would say content, but then consider the following ways of marking the beginning of a proof:

1. 'Proof' heading:

   ... preceding text.
   Proof. Text of the proof ....

2. Dingbat:

   ... preceding text.
   ¶ Text of the proof ....

3. Inline horizontal rule:

   ... preceding text.
   —— Text of the proof ....

4. Full-measure rule and whitespace:

   ... preceding text.

   Text of the proof ....

5. White space only:

   ... preceding text.

   Text of the proof ....

At what point does the material that marks the beginning of the proof stop being content and start being style?

(10) In math formulas font changes are mathematically significant. Suppose that an author uses bold and non-bold versions of Θ (Greek Theta) in a paper for different mathematical entities. Consider what happens in a bold section head:

4. Let's talk about Θ

The reader can't tell which version of Θ was intended. In more complicated examples serious garbling of the formula's meaning can occur. To prevent such garbling the AMS policy is to never vary the fonts in a math formula for purely stylistic reasons. (As a tangential benefit, this prevents practical problems with availability of bold or sans serif math symbols.)

(11) In traditional composition, certain minor points of style cut across the logical structure of the text and are therefore hard to handle by automated typesetting. For example, if the declared font for a theorem head is bold, then the logical thing to do for all in-between material *within* the theorem head, such as punctuation, would be to use bold. But it may be that one subcomponent within a theorem head (e.g. the number in the swapped-numbers example above) is not to be printed in bold. The result would be a bold period after a nonbold number, which tends to look bad:

1.2. **Theorem.** *Swapped numbers.*

(Look closely at the periods.) A bold period also looks odd following a math formula at the end of a run-in section head, since math formulas are not printed in bold in accordance with the AMS policy mentioned above. To prevent such distracting oddities, most manuals of composition style say that punctuation should take on the weight (and sometimes slant, if applicable) of the preceding text. The DBT system takes care of this, but not without effort.[4]

There is also the question, should the period be considered in-between material, or an interior part of the preceding component? I tend to believe treating it as in-between material is better, but the weight mismatch problem is one example of the complications lurking behind that approach.

(12) It's unclear how best to specify vertical spacing around displayed equations. Specifying it in base-to-base terms leaves open, for many equations, a question 'The base of what??'. And the reasonably satisfactory answer 'the base of the highest and lowest full-size entities in the equation, disregarding delimiters' is rather difficult to implement. Not to mention the fact that egregiously large sub or superscripts can still make a mockery of the intended spacing. The alternative of specifying the space around equations as visual space to the topmost and bottom-most points of the equation doesn't always work perfectly either. The mechanism built into TeX is a sort of composite method that uses base-to-base spacing until the equation contents get too tall, then switch to visual spacing. Although this works pretty well, it often results in a discrepancy of two or three points in the spacing above and below equations on the same page.

(13) In the presence of running heads, accurate calculation of type block height and text block height is not easy. Doing it properly requires knowing the alphabet height of the fonts used for the running head and the main text. Then placing the main text and the running head properly within the defined heights requires some careful work.

[4] A numeric code for the most recently used font is recorded in `\spacefactor` for later reference.

(14) 'Above display short space'. There is a mechanism built into TeX for automatically reducing the vertical space around a displayed equation if the horizontal distance from the end of the preceding text line is large enough.

Here we are testing a short skip.

$$\frac{B_1}{A}$$

Note the space below.

Contrasting to this test of the not-short skip.

$$A = B + \frac{C - D}{E}$$

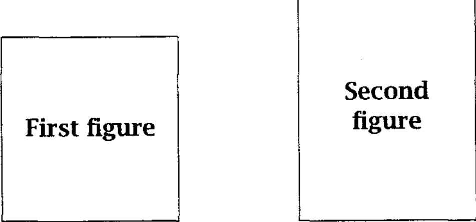
Note the space below.

How should this design requirement be specified in a style template?

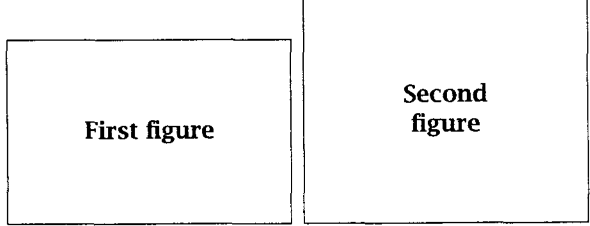
(15) Side-by-side figures: Proper specification of the surrounding space is a little tricky.

First try: put all the available space between the figures.

Second try: Divide the available space into four parts, put two parts in the middle, and one on each side.



But when the figures are close to half of the available width, the results can be poor:



Third try: Specify a minimum space between the figures, then divide up the space remaining into four equally distributed parts.

First figure

Second figure

(16) QED symbol placement: The end of a mathematical proof is frequently marked by the letters 'Q.E.D.' or some sort of dingbat, as an aid to the reader. As the QED symbol is consistently applied to all proofs in a document, it seems best to treat it as an aspect of the proof style and add it automatically, rather than requiring it to be included in the content portion of the proof. To avoid wasting vertical space, the QED symbol is usually fitted into the bottom right corner of the last paragraph. But then if the proof ends with a displayed equation, or anything other than a plain paragraph, the formatting of the last text in the proof will probably be finished before the `\endproof` command kicks into action; retrofitting the QED symbol into the preceding text becomes a real problem.

(17) Omitting redundant punctuation: e.g., a period after a run-in section head that ends with a question mark; compare the omission of the sentence-ending period when a sentence ends with *etc.* (This is a problem in documents where sentences are marked up with `\sentence` ... `\endsentence`.)

(18) Running two different elements together. If a numbered list falls at the very beginning of a headed element such as a proof, it is common practice to run the first list item in on the same line as the proof head (saves vertical space, thus tends to save paper = cost = final cost to the reader). But the rule for deciding when to allow such running in is difficult to express in a way that can be automated.

(19) How to evaluate: For some style variables it is occasionally desirable to give a value in terms of other variables. But evaluation in TEX is problematic for any arithmetic expression more complicated than a factor multiplying a register. You cannot write

```
[THM]linespacing:{\currtypesize + 2pt}
```

The TEX way of writing such an assignment is

```
\linespacing = \currtypesize
\advance\linespacing 2pt
```

This can't easily be jammed into the value slot of a template entry. It would be possible to apply some sort of arithmetic processing when reading the value of a variable, but designing a syntax suitable for TEX would not be particularly easy.

(20) Case changing. TEX's `\uppercase` or `\lowercase` cannot be applied indiscriminately to an element component if the component might contain (for example) a math formula. Also, in AMS editorial practice the uppercase form of McLeod is McLEOD, not MCLEOD. For related TEXnical reasons, case changes in DBT cannot be applied to a composite component, only to 'atomic' components.

It would be better if case changes were combined with font changes — in other words, if uppercasing were done by switching to an uppercase font instead of by applying `\uppercase`. This would automatically avoid problems with embedded math formulas, for example. But the implementation details would be a bit thorny. You can make a virtual font that substitutes capital letters for the lowercase letters, but it seems sort of silly to create a separate font to access characters that already exist in the current font; so perhaps some sort of output encoding change would be better. But TEX 3.x doesn't provide that capability.

Conclusion

Most of the analysis in DBT for breaking down style specifications into suitable variables is straightforward. A few aspects, however — notably the 'arrangement' concept — have little precedent that I know of and have not had sufficient testing and analysis to raise their status beyond 'experimental'. The implementation in TEX of DBT approaches or surpasses some of the current typical limits on TEX memory resources and processing speed. Nonetheless, the system is currently in use in-house at the AMS and is doing a pretty good job of delivering the desired easy maintenance of our publication designs. The possibility of wider release at some point is not out of the question, but the amount of work necessary to polish it up for such release (including some optimization to decrease the strain on TEX memory capacities) would be rather large.

References

Anagnostopoulos, Paul. "ZzTEX: A macro package for books." *TUGboat* **13** (4), pages 497-504, 1992.

Bringhurst, Robert. *Elements of Typographic Style.* Hartley & Marks, Point Roberts, Washington, USA, 1992.

Brown, P.J. "Using logical objects to control hypertext appearance." *Electronic Publishing — Origination, Dissemination and Design* **4** (2), pages 109-118, 1991.

Dobrowolski, Andrew. "Typesetting SGML documents using TEX." *TUGboat* **12** (3), pages 409-414, 1991.

Eijkhout, Victor. "Just give me a Lollipop (It makes my heart go giddy-up)." *TUGboat* **13** (3), pages 341-346, 1992.

Eijkhout, Victor and Andries Lenstra. "The document style designer as a separate entity." *TUGboat* **12** (1), pages 31-34, 1991.

Hansen, Bo Stig. "A function-based formatting model." *Electronic Publishing — Origination, Dissemination and Design* **3** (1), pages 3-28, 1990.

Livengood, William P. *The Maple Press Company Style Book.* Maple Press Co., York, Pennsylvania, USA, 1931.

Less is More: Complex Page Layouts and Shapes with TEX

Alan Hoenig
John Jay College / CUNY, 17 Bay Avenue, Huntington, NY 11743, USA
ajhjj@cunyvm.cuny.edu

Abstract

This presentation discusses by means of several examples the ability of TEX to generate complex page layouts in a special case of text. In this special case of restricted text, no tall characters or stretchable vertical glue is allowed and all lines are assumed to have the same height and depth. To anyone accustomed to TEX's generality, this may seem overwhelmingly restrictive, but probably 99% of all typesetting—letters, novels, non-technical material, etc.—conforms to this model. Examples I'll discuss include line numbering of text, changebars, flowing text entirely around a special shape, and fine control over inter-column cutouts (*à la The New Yorker* magazine). I'll also how to include material which does *not* conform to this model (such as section heads, display material, and so on), so this model is more flexible than it appears.

Introduction

An imaginary conversation between the author and a 'conventional' desktop publisher inspired this article. The discussion involves the ability to typeset fancy column shapes. Consider:

DTPUB: My software can create fancy column shapes. I can 'synchronize' the shapes so that pairs of columns can perfectly enclose an odd-shaped figure.

AU: Sounds impressive. But how do you account for displayed equations, tall characters, stretchable white space, and other items that might louse up the alignment of the lines on the right to the lines on the left? And what happens if the column break leads to a lonely widow or club line at the top or bottom of a column?

DTPUB: What are you talking about? I only set material so that each line is exactly the same height as any other. As far as math goes, why would it appear in this kind of context? I don't care about 'lonely' lines because horizontal alignment of lines across columns takes precedence. And what the heck's stretchable glue?

Designers of macro packages bend over backwards to make their macros as general purpose as possible, but there are still typographic effects that remain difficult if not outright impossible in the general case. This mini-conversation made me realize that complete generality may not be a virtue, at least not 100% of the time. I decided to imagine that my typesetting was restricted to the same universe as that of the desktop publisher to see if giving up some flexibility led to the ability to do new things with TEX.

In what follows, I shall use the term 'restricted text' to refer to text conforming to that of the desktop publisher above. Specifically, restricted text consists only of prose such that the total height of each line—the sum of the depth and height—does not exceed some certain amount. Tall characters and stretchable vertical glue make no appearance in this text, and nor does a reluctance to leave widows and club lines dangling fore or aft of a column. Although this is quite restrictive, much (if not most) printed matter does conform to this model. Furthermore, it turns out that section heads, extended quotations, and so on—elements which do not conform to this restricted model—can be incorporated into a restricted document, so restricted text is not quite so restrictive after all.

I will present some examples whereby complex and unusual page layouts can be done with TEX providing that the text conforms to the restrictions we mentioned above. Using this model, I am cautiously optimistic that that anything any other desktop publishing program can do, TEX can also.

To no one's surprise, several of the crucial ideas have been lifted from work done by Don Knuth. Several additional ideas are identical or similar to ones discussed by David Salomon in his ongoing series of tutorials on the `\output` routine which can be found in *TUGboat*.

Purpose of This Presentation

It is not my purpose here to present a finished set of macros to accomplish the tasks I will discuss. Rather, I hope to suggest a philosophy—that reining in TEX can sometimes be beneficial—and to present examples showing some advantages of this thinking. Readers may decide for themselves whether the benefits really do outweigh the disadvantages.

Nevertheless, readers who agree with the author that it sometimes pays to restrict certain of TEX's abilities may wish to recreate these effects. The examples of the first portion of the paper can be integrated into personal style files using the macro snippets that appear. The final example (magazine layout) is a different matter, and here the author has an additional agenda. I seek feedback that the user interface—the way that macro names and macro arguments have been organized—is reasonable. After incorporating comments, I hope to make these macros available shortly thereafter.

A New Output Routine

It seemed likely to me that TEX would have to be re-configured a bit to make dealing with restricted text easier. A new output routine, suggested by Don Knuth (Knuth 1987) in another context, seemed to fit the bill. The height of a strut, `\strutht`, is usually the maximum height of a line of text in the restricted text under discussion. If we set the `\vsize` of the document to be `\strutht`, then TEX's output routine will obligingly slice up the text into line-sized morsels. It will be up to this restructured `\output` routine to collect these lines, stack them together, and actually ship out a page only when the line count equals the capacity of a single page.

Of course, as TEX passes each line to the collecting area, perhaps using code in the output routine something like

```
\ifnum\lineno < \linesperpage
  \global\setbox\partialpage=
   \vbox{\unvbox\partialpage \box255}
\else ...
```

it's possible to include a macro to do something special to each line, much as the token list `\everypar` can at the start of each paragraph. (In this way, we can mimic the structure of an `\everyline` token list, something that features high on many people's TEX wish list. *Mimic* is the key word, for source file tokens have long been processed by TEX.) That is, the above `\output` fragment should better look like

```
\ifnum\lineno < \linesperpage
  \setbox\partialpage=\vbox{
  \unvbox\partialpage\processline}
\else ...
```

Let's pause to consider some of the ways we can exploit `\processline` (which can be thought of as a mock-`\everyline`).

Numbering lines of text. Numbering lines in an `\obeylines` environment has always been straightforward. Now it's simple enough for regular text, at least in our restricted case. Simply invoke definitions like

```
\newcount\linesdone
\newif\ifdivisiblebyfive
\newcount\scr
\def\ModFive{% Is \linesdone div by 5?
  \global\divisiblebyfivefalse
  \scr=\linesdone \divide\scr by 5
  \multiply\scr by-5
  \advance\scr by\linesdone
  \ifnum\scr=0
  \global\divisiblebyfivetrue \fi}
\def\processline{%
  \global\advance\linesdone by 1
  \ModFive \ifdivisiblebyfive
    \hbox{\llap{%
    \oldstyle\the\linesdone\ }\box255}
  \else\box255 \fi}
```

which will print line numbers every five lines across paragraph and page boundaries. Figure 1 displays an example of typography showing line numbering.

Change bars. Most change bar styles use PostScript. Here's one way that we can implement it independently of PostScript.

Even with our restrictions in effect, this problem highlights an important set of problems. Because of TEX's asynchronous mode of processing, output occurs at very different times then when the source file is chewed by TEX's mouth. So a `\changebar` macro has to contain instructions to the output routine which `\processline` will then execute.

Of the several methods available for this communication, I chose the following. The command `\changebar` inserts a strut whose depth is ever so slightly greater than the normal depth of a line. The depth is so slight that no reader will ever be able to see it, but it is great enough so that TEX can perceive it. The slight amount we use is two scaled points; we recall that one printer's point contains 64K scaled points.

If we code `\changebar` to act like a font change, so that a group must enclose the change'd text, then a simplified coding might look as follows. First, we define deep struts.

```
\newbox\varstrutbox  \newdimen\lostrutdp
\def\varstrut{\relax
 \ifmmode\copy\varstrutbox\else
 \unhcopy\varstrutbox\fi}
\def\lostrut#1{%
  \global\lostrutdp=\strutdp
  \global\advance\lostrutdp by#1 sp
  \global\setbox\varstrutbox=
  \hbox{\vrule width0pt height\strutht
   depth\lostrutdp}\varstrut}
```

And now, here is TEX code for `\changebar`. The `\aftergroup` hack ensures that the increase to the height of a line happens after the `\changebar` group has been concluded. The output routine will check the depth of the line. A surplus depth of two scaled points signals the beginning of a change bar, and a surplus of three scaled points signals its end.

HERMAN MELVILLE *MOBY-DICK* CHAPTER I

head. True, they rather order me about some, and make me jump from spar to spar, like a grasshopper in a May meadow. And at first, this sort of thing is unpleasant enough. It touches one's sense of honor, particularly if you come of an old established family in
115 the land, the van Rensselaers, or Randolphs, or Hardicanutes. And more than all, if just previous to putting your hand into the tar-pot, you have been lording it as a country schoolmaster, making the tallest boys stand in awe of you. The transition is a keen one, I assure you, from the schoolmaster to a sailor, and requires a strong
120 decoction of Seneca and the Stoics to enable you to grin and bear it. But even this wears off in time.

What of it, if some old hunks of a sea-captain orders me to get a broom and sweep down the decks? What does that indignity amount to, weighed, I mean, in the scales of the New Testament?
125 Do you think the archangel Gabriel thinks anything the less of me, because I promptly and respectfully obey that old hunks in that particular instance? Who aint a slave? Tell me that. Well, then, however the old sea-captains may order me about—however they may thump and punch me about, I have the satisfaction of knowing that
130 it is all right; that everybody else is one way or other served in much the same way—either in a physical or metaphysical point of view, that is; and so the universal thump is passed round, and all hands should rub each other's shoulder-blades, and be content.

Again, I always go to sea as a sailor, because they make a point
135 of paying me for my trouble, whereas they never pay passengers a single penny that I ever heard of. On the contrary, passengers themselves must pay. And there is all the difference in the world between paying and being paid. The act of paying is perhaps the most uncomfortable infliction that the two orchard thieves entailed upon
140 us. But being paid,—what will compare with it? The urbane activity with which a man receives money is really marvellous, considering that we so earnestly believe money to be the root of all earthly ills, and that on no account can a monied man enter heaven. Ah! how cheerfully we consign ourselves to perdition!

145 Finally, I always go to sea as a sailor, because of the wholesome exercise and pure air of the forecastle deck. For as in this world, head winds are far more prevalent than winds from astern (that is,

Figure 1: Typography similar to John Baskerville's 1757 edition of the *Bucolics and Georgics of Virgil*. This sample is set in Monotype Baskerville.

```
\def\changebar{\aftergroup\endchangebar
 \lostrut2}
\def\endchangebar{\lostrut3}
```

The `\processline` macro must examine the depth of the current line and take action accordingly. It's the responsibility of a macro to add the actual changebar segment to \box255 which contains a single line of text.

```
\def\addchangebar{\hbox{\llap{\vrule
  width4pt height\strutht
  depth\strutdp\quad}%
 \box255}}
```

Our routine will examine the register `\DeltaDP` which stores the surplus depth and sets flags as appropriate. We will also assume that the last line of the changebar group needs a changebar segment, and therefore another switch `\ifflushing` is necessary for that purpose. (Without this switch, the last typeset line of changed text would appear without the changebar segment.)

```
\newif\ifchanging \newif\ifflushing
\def\processline{%
  \ifnum\DeltaDP=2
    \global\changingtrue \fi
  \ifnum\DeltaDP=3
    \global\changingfalse
    \global\flushingtrue \fi
  \ifchanging \addchangebar \else
    \ifflushing \addchangebar
     \global\flushingfalse
    \else\box255
 \fi\fi}
```

The `\output` routine needs a hook to check on surplus depth. Macro `\identity` simply typesets \box255 without doing anything to it.

```
\def\checkline{%
  \dimen0=\dp255
  \advance\dimen0 by-\strutdp
  \DeltaDP=\dimen0
  \ifnum\DeltaDP=1
  \global\let\processline=\identity \fi
}
```

We reserve a surplus depth of 1 as a signal to return to normal, standard typesetting. A redefined `\bye` command will automatically cancel out any special effects.

```
\outer\def\bye{\finish
 \vfill\supereject\end}
\def\finish{\endgraf
 \leavevmode\lostrut1 \endgraf}
```

Source marked up similar to

```
 ... end, {\changebar for ...
 ... is wrong}.  It follows that ...
```

generated the changebars (and the text) displayed in figure 2.

Section heads; escaping the restrictions. A section head is an example of a document component that would escape the restrictions we have imposed upon ourselves. This is a good time to explore ways to include these elements in our document.

The previous example suggests ways that signals to the output routine can allow us to build up the partial page out of non-restricted components. These components should have a total height equal to a whole number of single line heights.

Let us suppose that we want to leave a total vertical separation of two lines between the sections. In this white space we insert the section head, which should be 12-point bold type. Furthermore, we want to leave a little extra space between the section head and the following line, and we don't want to print the section head unless there is room for at least one additional line following the section head. It would appear to be straightforward to write a `\section` macro to create a `\vbox` to create the vertically spaced heading and to send a signal to `\output` which can determine whether there is room enough for the heading and act accordingly.

But a problem could arise. What if there were several headings to be very close to one another in the document? In that case, because TeX's mouth often gets ahead of the processing done by `\output`, new contents of a `\sectbox` would over-write previous contents before TeX would have typeset them. We can be sure, though, that `\output` will process these special boxes in the order in which they appear in the document, and so we adopt a subtler strategy. We will use a single special `\vbox` to hold these insertions, and we will take care to add new boxes to it from the bottom. Whenever `\output` needs to take a special box for typesetting, we will take off the top of this box. This special `\insertbox` acts like a "first in first out" (FIFO) queue containing insertions. In this way, we can be sure that no headings get lost in the asynchronous maze which is TeX.

Incidentally, we can use this method to include displayed equations, tables, extended quotations, and other non-restricted text in the body of a 'restricted' document, so that our restricted document is not so restricted after all. (Note that the term 'insertion' that I've used refers to text that does not conform to a restricted format but which I wish to include. It is quite different in spirit and scope from the usual TeX insertion whose eventual appearance on the page is unpredictable.)

```
\def\section#1{\setbox\sectbox=
 \vbox to2\baselineskip{%
 \vss \leftline{\strut\bigbf #1}%
 \vskip1pt\hrule height0pt}%
 \global\setbox\insertbox=
 \vbox{\unvbox\insertbox
 \goodbreak \box\sectbox}%
```

~~TYPOGRAPHY~~ Typography may be defined as the art of rightly disposing printing material in accordance with specific purpose; of so arranging the letters, distributing the space and controlling the type as to aid to the maximum the reader's comprehension of the text. Typography is the efficient means to an essentially utilitarian and only accidentaly aesthetic end, ~~for enjoynemt of paterns is rarely the readers' cheif aim. There4, any disposition of pritning materail which, whtaever the intentoin, has the affect of coming betwixt author & reader is rong~~ for enjoyment of patterns is rarely the reader's chief aim. Therefore, any disposition of printing material which, whatever the intention, has the effect of coming between author and reader is wrong. It follows that in the printing of books meant to be read there is little room for 'bright' typography. Even dullness and monotony in the typesetting are far less vicious to a reader than typographical eccentricity or pleasantry. Cunning of this sort is desirable, even essential in the typography of propaganda, whether for commerce, politics, or religion, because in such printing only the freshest survive inattention. ~~But the typography of books, apart from the category of narrowly limited editions, requires an obedience to convention which is almost absolute—and with reason.~~

~~The laws governing the typography of books intended for general circulation are based first upon the essential nature of alphabetical writing, and secondly upon the traditions, explicit, or implicit prevailing in the society for which the printer is working.~~ But the typography of books, apart from the category of narrowly limited editions, requires an obedience to convention which is almost absolute—and with reason.

The laws governing the typography of books intended for general circulation are based first upon the essential nature of alphabetical writing, and secondly upon the traditions, explicit, or implicit prevailing in the society for which the printer is working. While a universal character or typography applicable to all books produced in a given national area is practicable, to impose a universal detailed formula upon all books printed in roman types is not. National tradition expresses itself in the varying separation of the book into ~~preliminaries~~ prelims, chapters, etc., no less than in the design of the type. But at least there are physical rules of linear composition which are obeyed by all printers who know their job.

Figure 2: Text decorated with changebars. Strike outs indicate the revised material. (This text is set in Adobe Garamond.)

```
%  build \sectbox from bottom
 \signal % message to output routine
}
\def\signal{\line{\hss\lostrut2}%
 \endgraf}
```

A box called `\sectbox` holds the current heading.

Back in the output routine, we use a definition like this for `\processline`.

```
\def\processline{\ifnum\scr=2
 \writesection \else
 \hbox{\strut\box255}\fi}
```

Here, `\scr` is a scratch counter which will hold the excess depth of the most recent `\box255`. Any value of `\scr` other than zero is a signal to be acted upon in some way. Macro `\writesection` is actually responsible for printing the section head or saving it for the top of the next page.

```
\def\writesection{%
% \box255 has a deep strut--discard it.
  \setbox\nullbox=\box255
  \setbox255=
    \vsplit\sectbox to 2\baselineskip
%% see is there room?
  \global\scr=\linesperpage
  \global\advance\scr by-\lineno
  \ifnum\scr>3 % there is room
    \box255 \global\advance\lineno by 1
  \else %
% spit out empty lines, go to next page
    \global\lineno=\linesperpage
      \loop \copy\emptyline
          \advance\scr by-1 \ifnum\scr>0
      \repeat
      \global\setbox\sectbox=
      \vbox{\box255
      \goodbreak\unvbox\sectbox}%
      \aftergroup\signal
    \fi \global\scr=0 }
```

Complex Page Shapes

Setting `\vsize` to `\baselineskip` is a simple but powerful tool, but we need more help to have extensive control over the appearance of the page. TEX's basic `\parshape` command is useful for single paragraphs but appears to be useless in case a shape should extend across paragraph boundaries. We can extend `\parshape` but to do so, we need to recall some useful facts: at the conclusion of each paragraph, the register `\prevgraf` has been increased by the number of lines in the paragraph; TEX uses the value of `\prevgraf` at the start of a paragraph as the index into the list of line dimensions which accompany any `\parshape`; and all paragraph shaping commands—`\parshape` and `\hangafter`—are reset to their standard values. Extending this paragraph shaping capability involves making TEX ignore this usual resetting procedure.

The following lines provide a `\pageshape` command, whose syntax mirrors that of `\parshape`.

```
\newcount\totallines
```

TYPOGRAPHY may be defined as the art of rightly disposing printing material in accordance with specific purpose; of so arranging the letters, distributing the space and controlling the type as to aid to the maximum the reader's comprehension of the text. Typography is the efficient means to an essentially utilitarian & only accidentaly aesthetic end, for enjoyment of patterns is rarely the reader's chief aim. Therefore, any disposition of printing material which, whatever the intention, has the effect of coming between author & reader is wrong. It follows that in the printing of books meant to be read there is little room for 'bright' typography. Even dullness and monotony in the typesetting are far less vicious to a reader than typographical eccentricity or pleasantry. Cunning of this sort is desirable, even essential in the typography of propaganda, whether for commerce, politics, or religion, because in such printing only the freshest survive inattention. But the typography of books, apart from the category of narrowly limited editions, requires an obedience to convention which is almost absolute—and with reason. § The laws governing the typography of books intended for general circulation are based first upon the essential nature of alphabetical writing, and secondly upon the traditions, explicit, or implicit prevailing in the society for which the printer is working. While a universal character or tyupography applicable to all books produced in a given national area is practicable, to impose a universal detailed formula upon all books printed in roman types is not. National tradition expresses itself in the varying separation of the book into prelims, chapters, etc.,no less than in the design of the type. But at least there are physical rules of linear composition which are obeyed by all printers who know their job.

FIRST PRINCIPLES OF TYPOGRAPHY *by* STANLEY MORISON

abcdefghijklm ABCDEFGHIJKLMNOPQRSTUVWXYZ *nopqrstuvwxyz*

Figure 3: Hermann Zapf designed this page before 1954. It originally appeared in his *Manuale Typographicum.* Zapf's original specimen appeared in some flavor of Baskerville; this is set in Monotype Baskerville.

```
\def\pageshape{\afterassignment
 \dopageshape \scr }
\def\dopageshape{%
 \ifnum\scr=0\def\par{\endgraf}
 \else\def\par{{\endgraf
  \global\totallines=\prevgraf}}%
 \fi
 \everypar={\prevgraf=\totallines}%
 \parshape \scr }
```

The `\afterassignment` hack in `\pageshape` allows us to obtain the numeric value for `\pageshape`. In case it is zero, standard paragraph shaping is invoked. Interesting things occur otherwise. We use `\totallines` to remember the final value of `\prevgraf`, and then use `\everypar` to set `\prevgraf` as each paragraph commences.

`\par` has been redefined to end the paragraph, but to do so within a group (which is why the `\totallines` equation needs a `\global` prefix). What is the purpose of this additional level of grouping? As the paragraph concludes, TeX restores the standard value for `\parshape`. But this restoration occurs within a group—which means that the former value prevails when the group is exited. This former value is precisely the `\parshape` specified within the "`\pageshape`" command. It's as if we included a `\parshape` specification within `\everypar`.

(As far as the current author knows, this method was first elucidated by the author of TeX and communicated in a private letter to Elizabeth Barnhart of *TV Guide* in about 1987. I am grateful to her for having made this letter available to me.)

An immediate application is toward the creation of windows within paragraphs. How so? Suppose we wanted text with a window like this:

We simply use `\pageshape` to create text which looks as follows.

(The dashed lines emphasize the the position of the interline glue.) Then, macros in `\output` have to be smart enough to backspace up (via a command like `\vskip-\baselineskip`) after adding the left side of the window to the partial page but before adding the right side.

Figure 3 shows how I used TeX to typeset a page originally typeset by Hermann Zapf some years ago. This method can be generalized to form windows of arbitrary shape; see, for example, (Hoenig 1992) and (Hoenig 1987).

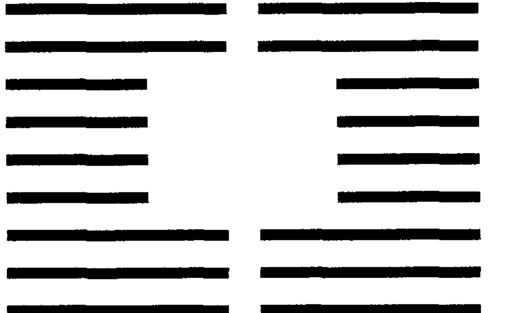


Figure 4: A simple symmetric straddle between two columns.

Magazine layout. My last example was inspired by layouts of magazines like *The New Yorker* and *Scientific American.* These layouts are deceptively simple. Both magazines fit into our model of restricted text. Both magazines employ a multiple-column format, but both also depend on sophisticated methods of leaving space for figures, author biographies, ads, cute drawings, and the like. Part of a two-column spread for *The New Yorker*, for example, might schematically appear as in figure 4. Can we get TEX to do it, and, if yes, using a reasonable set of mark-up conventions? Based on my investigations, we can be optimistic that such macros are possible.

It has proven possible to design macros with a reasonable interface. Before the text begins, all such commands are sandwiched between two commands, `\layout` and `\endlayout`. The layout commands recognize two kinds of layout 'events', namely skipping lines and straddles, which are spaces straddling an intercolumn boundary and requiring indentations in the columns which must match across columns. Miscellaneous other commands can be included, such as `\nextcolumn` or `\nextpage`, which make the task of anchoring the layout to the page, easier.

How may we specify a layout event? All such events have a vertical extent—how many lines are they supposed to last. In addition, for straddles, we need to specify a horizontal extent plus information as to how to position the straddle to the left or right of the column boundary. Finally, TEX needs to know how far down from the top of the page (or up from the bottom of the page) to begin the skip or straddle.

Commands controlling layout. Here follows a brief glossary of the major layout commands that I have (so far) been able to implement. All of the commands controlling layout events occur in two varieties. The first specifies that the beginning of the event (straddle or skip) should occur so many lines down from the first line of the page; these have `...FromTop` in their names. The second says that the bottom of the

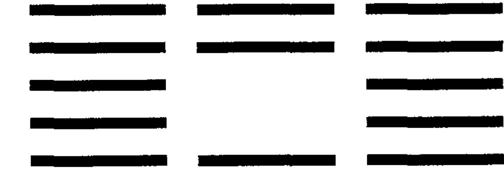


Figure 5: A small gap in a short page containing three columns.

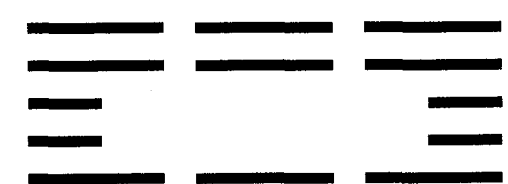


Figure 6: A straddle across one column in a page containing three columns.

event begins so many lines up from the bottom of the page and have `...FromBottom` in their names.

`\n` tells the macros how many columns to set the text in. No attempt is made to balance columns on the last page (although such a capability could be added); we assume that editors need to know by how much an article exceeds or falls short of its allotted space.

`\StraddleFromTop#1#2#3` creates a straddle similar to that of figure 4. Here, #1 is the total width of the straddle, #2 is total height of the straddle in lines, and #3 is the number of lines from the top of the page at which point the straddle begins. The straddle of figure 4 could have been specified with a command like `\StraddleFromTop{6pc}{4}{2}`.

`\StraddleFromBottom#1#2#3` Here, parameter #3 is the number of lines of the *bottom* of the indent from the bottom of the page. Using this command, the straddle of figure 4 would be `\StraddleFromBottom{6pc}{4}{3}`.

`\AStraddleFromBottom#1#2#3#4` This command yields an asymmetric straddle. Here, #4 is the length of the left portion of the indent. There is also an `\AStraddleFromTop` command.

`\SkipFromTop#1#2` will skip #1 lines (leave a vertical gap of that many lines) starting #2 lines from the top. There is a companion command `\SkipFromBottom#1#2` where the second parameter #2 specifies the number of lines below the bottom of the gap.

The vertical gap in figure 5 comes from either of the commands `\SkipFromTop{2}{2}` or `\SkipFromBottom{2}{1}`.

`\StraddlesFromTop#1#2#3#4` This and the following commands control layout events that span

several columns. Here, parameter #1 is the total width of cutout; #2 should be the number of full columns straddled; #3 is the vertical height in lines; and #4 is the number of lines from the top of the page. Figure 6 displays a straddle across a single column; either of `\StraddlesFromTop{1.3\C}{1}{2}{2}` or `\StraddlesFromBottom{1.3\C}{1}{2}{1}` (where `\colwd` is some hypothetical register containing the width of a column) could have generated it.

`\SpanColumnsFromTop#1#2#3` requests TEX to create a vertical gap, but a gap which spans several columns horizontally. Here, #1 is the number of columns spanned, #2 is the number of vertical lines to be skipped, and #3 is the number of lines from the top of the page at which to begin the span. A companion command, `\SpanColumnsFromBottom` also exists in case it is easier to relate gaps to the page bottom. It also requires three parameters, the third one being the number of lines from the page bottom.

`\TwoEyesFromTop#1#2#3` generates the peculiar formation shown in figure 9. The three parameters refer to the total length of the eye, the vertical duration of the eye in lines, and the number of lines down from the page. Its companion is `\TwoEyesFromBottom}`.

`\nextcolumn` and `\nextpage` tells the macros to resume its line counting on the next column or page. If several columns are set plain with no layout events, then several `\nextcolumn` commands will need to follow each other in the `\layout` section of the document.

In addition to these high level commands, there are three others which lurk behind the scenes: `\AtLeftFromBottom`, `\AtRightFromBottom`, and `\LeftRightFromBottom` (plus their `Top` companions). These control indentations at the sides of a column. These together with the `\SkipFrom...` commands combine to produce all the commands listed above.

Other weird shapes. Although I have described several kinds of layout cutouts, all are combinations of two kinds of basic shapes—a command to create a vertical gap in a single column, and two commands to create either a left or right indentation in a column. Careful study of the macros shows how to combine these to create a larger palette of layout commands. However, commands can be 'stacked'. That is, it is possible to create odd cutouts by combining several commands together. For example, an asymmetric straddle lasting for a single line is the result of a command like

```
\AStraddleFromTop{10pc}{1}%
 {\scratch}{\dimen0}
```

where `\dimen0` contains the amount of the left indent and `\scratch` is a scratch count register. This was the basic component that produced the layout of figure 7.

An example. Figures 8 through 10 show the first few pages of *Moby-Dick* which have been formatted with these macros. This is offered as an example of the use of these macros, *not* as an example of good typography. Even with absurd values of `\tolerance` and `\hyphenpenalty` (9600 and -100) and with nine-point type, it is difficult for TEX to generate acceptable line breaks. Furthermore, no claim is made that the placement of column cutouts is in any way pleasing.

In general, cutouts are specified starting at the top of a column and proceeding downward. Proceed to the next column on the right when this column is finished. Use `\nextcolumn` and `\nextpage` commands to get to the next column or page. After the last layout element has been specified, conclude with `\endlayout`. The following lines provide one way to generate the pages shown in figures 8 through 10.

```
\layout
\SpanColumnsFromTop{2}{4}{0}
\StraddleFromBottom{16}{4}{16}
\nextcolumn
\nextcolumn
\SkipFromBottom{\bioht}{0}
\nextpage %%
\TwoEyesFromTop{4.5pc}{7}{5}
\TwoEyesFromBottom{4.5pc}{7}{5}
\nextpage %%
\StraddleFromTop{\colwd}{4}{5}
\StraddlesFromBottom{%
  2\colwd}{1}{4}{5}
\nextcolumn
\StraddleFromBottom{%
  \colwd}{4}{13}
\endlayout
```

Limitations and bugs. No error checking to speak of has been built in to these macros. If you specify incorrect or contradictory layout parameters, you get unpredictable output rather than error messages.

There is no good way to currently place logos or other typeset snippets in the gaps created by these layout commands. A rudimentary facility does exist, but it is not robust enough to report on at present.

References

Hoenig, Alan, "TEX does windows—conclusion," *TUGboat*, **8**(2), pages 212–216, July 1987.

Hoenig, Alan, "When TEX and METAFONT work together," Proceedings of the 7th European TEX Conference, Prague (September 14–18, 1992).

Knuth, Donald E., "Saturday morning problem—conclusion," *TUGboat*, **8**(2), page 211, July 1987.

Moby-Dick **Chapter I: Loomings** *Page 2*

him. Go visit the Prairies in June, when for scores on scores of miles you wade knee-deep among Tiger-lilies—what is the one charm wanting?—Water—there is not a drop of water there! Were Niagara but a cataract of sand, would you travel your thousand miles to see it? Why did the poor poet of Tennessee, upon suddenly receiving two handfuls of silver, deliberate whether to buy him a coat, which he sadly needed, or invest his money in a pedestrian trip to Rockaway Beach? Why is almost every robust healthy boy with a robust healthy soul in him, at some time or other crazy to go to sea? Why upon your first voyage as a passenger, did you yourself feel such a mystical vibration, when first told that you and your ship were now out of sight of land? Why did the old Persians hold the sea holy? Why did the Greeks give it a separate deity, and own brother of Jove? Surely all this is not without meaning. And still deeper the meaning of that story of Narcissus, who because he could not grasp the tormenting, mild image he saw in the fountain, plunged into it and was drowned. But that same image, we ourselves see in all rivers and oceans. It is the image of the ungraspable phantom of life; and this is the key to it all.

Now, when I say that I am in the habit of going to sea whenever I begin to grow hazy about the eyes, and begin to be over conscious of my lungs, I do not mean to have it inferred that I ever go to sea as a passenger. For to go as a passenger you must needs have a purse, and a purse is but a rag unless you have something in it. Besides, passengers get sea-sick—grow quarrelsome—don't sleep of nights—do not enjoy themselves much, as a general thing;—no, I never go as a passenger; nor, though I am something of a salt, do I ever go to sea as a Commodore, or a Captain, or a Cook. I abandon the glory and distinction of such offices to those who like them. For my part, I abominate all honorable respectable toils, trials, and tribulations of every kind whatsoever. It is quite as much as I can do to take care of myself, without taking care of ships, barques, brigs, schooners, and what not. And as for going as cook,— though I confess there is considerable glory in that, a cook being a sort of officer on ship-board—yet, somehow, I never fancied broiling fowls;—though once broiled, judiciously buttered, and judgmatically salted and peppered, there is no one who will speak more respectfully, not to say reverentially, of a broiled fowl than I will. It is out of the idolatrous dotings of the old Egyptians upon broiled ibis and roasted river horse, that you see the mummies of those creatures in their huge bake-houses the pyramids.

No, when I go to sea, I go as a simple sailor, right before the mast, plumb down into the forecastle, aloft there to the royal mast-head. True, they rather order me about some, and make me jump from spar to spar, like a grasshopper in a May meadow. And at first, this sort of thing is unpleasant enough. It touches one's sense of honor, particularly if you come of an old established family in the land, the van Rensselaers, or Randolphs, or Hardicanutes. And more than all, if just previous to putting your hand into the tar-pot, you have been lording it as a country schoolmaster, making the tallest boys stand in awe of you. The transition is a keen one, I assure you, from the schoolmaster to a sailor, and requires a strong decoction of Seneca and the Stoics to enable you to grin and bear it. But even this wears off in time.

What of it, if some old hunks of a sea-captain orders me to get a broom and sweep down the decks? What does that indignity amount to, weighed, I mean, in the scales of the New Testament? Do you think the archangel Gabriel thinks anything the less of me, because I promptly and respectfully obey that old hunks in that particular instance? Who aint a slave? Tell me that. Well, then, however the old sea-captains may order me about—however they may thump and punch me about, I have the satisfaction of knowing that it is all right; that everybody else is one

Figure 7: An odd inter-column cutout (text face is Monotype Baskerville).

Chapter I
LOOMINGS

Call me Ishmael. Some years ago—never mind how long precisely—having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen, and regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is a damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and especially whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people's hats off—then, I account it high time to get to sea as soon as I can.

This is my substitute for pistol and ball. With a philosophical flourish Cato throws himself upon his sword; I quietly take to the ship. There is nothing surprising in this. If they but knew it, almost all men in their degree, some time or other, cherish very nearly the same feelings towards the ocean with me. There now is your insular city of the Manhattoes, belted round by wharves as Indian isles by coral reefs—commerce surrounds it with her surf.

Right and left, the streets take you waterward. Its extreme downtown is the battery, where that noble mole is washed by waves, and cooled by breezes, which a few hours previous were out of sight of land. Look at the crowds of water-gazers there.

Circumambulate the city of a dreamy Sabbath afternoon. Go from Corlears Hook to Coenties Slip, and from thence, by Whitehall northward. What do you see?—Posted like silent sentinels all around the town, stand thousands upon thousands of mortal men fixed in ocean reveries. Some leaning against the spiles; some seated upon the pier-heads; some looking over the bulwarks of ships from China; some high aloft in the rigging, as if striving to get a still better seaward peep. But these are all landsmen; of week days pent up in lath and plaster—tied to counters, nailed to benches, clinched to desks. How then is this? Are the green fields gone? What do they here?

HERMAN MELVILLE (1819-1891) was born in New York City, the descendant of English and Dutch families. He won fame for many of his novels, but *Moby-Dick* (1851), his greatest novel, has overshadowed almost all of them. His popularity began to decline after 1851, and he died in 1891 (in New York City) in total obscurity. This century saw a favorable revaluation of his work, and he is generally regarded now to be an outstanding writer of the sea and a master of realistic narrative and rhythmical prose.

Figure 8: An example of the use of column cutouts. The type is Monotype Columbus.

But look! here come more crowds, pacing straight for the water, and seemingly bound for a dive. Strange! Nothing will content them but the extremest limit of the land; loitering under the shady lee of yonder warehouses will not suffice. No. They must get just as nigh the water as they possibly can without falling in. And there they stand—miles of them—leagues. Inlanders all, they come from lanes and alleys, streets and avenues,—north, east, south, and west. Yet here they all unite. Tell me, does the magnetic virtue of the needles of the compasses of all those ships attract them thither?

Once more. Say, you are in the country; in some high land of lakes. Take almost any path you please, and ten to one it carries you down in a dale, and leaves you there by a pool in the stream. There is magic in it. Let the most absent-minded of men be plunged in his deepest reveries—stand that man on his legs, set his feet a-going, and he will infallibly lead you to water, if water there be in all that region. Should you ever be athirst in the great American desert, try this experiment, if your caravan happen to be supplied with a metaphysical professor. Yes, as every one knows, meditation and water are wedded for ever.

But here is an artist. He desires to paint you the dreamiest, shadiest, quietest, most enchanting bit of romantic landscape in all the valley of the Saco. What is the chief element he employs? There stand his trees, each with a hollow trunk, as if a hermit and a crucifix were within; and here sleeps his meadow, and there sleep his cattle; and up from yonder cottage goes a sleepy smoke. Deep into distant woodlands winds a mazy way, reaching to overlapping spurs of mountains bathed in their hill-side blue. But though the picture lies thus tranced, and though this pine-tree shakes down its sighs like leaves upon this shepherd's head, yet all were vain, unless the shepherd's eye were fixed upon the magic stream before him. Go visit the Prairies in June, when for scores on scores of miles you wade knee-deep among tiger lilies—what is the one charm wanting?—Water—there is not a drop of water there! Were Niagara but a cataract of sand, would you travel your thousand miles to see it? Why did the poor poet of Tennessee, upon suddenly receiving two handfuls of silver, deliberate whether to buy him a coat, which he sadly needed, or invest his money in a pedestrian trip to Rockaway Beach? Why is almost every robust healthy boy with a robust healthy soul in him, at some time

Figure 9: An example (continued) of the use of column cutouts.

or other crazy to go to sea? Why upon your first voyage as a passenger, did you yourself feel such a mystical vibration, when first told that you and your ship were now out of sight of land? Why did the old Persians hold the sea holy? Why did the Greeks give it a separate deity, and own brother of Jove? Surely all this is not without meaning. And still deeper the meaning of that story of Narcissus, who because he could not grasp the tormenting, mild image he saw in the fountain, plunged into it and was drowned. But that same image, we ourselves see in all rivers and oceans. It is the image of the ungraspable phantom of life; and this is the key to it all.

Now, when I say that I am in the habit of going to sea whenever I begin to grow hazy about the eyes, and begin to be over conscious of my lungs, I do not mean to have it inferred that I ever go to sea as a passenger. For to go as a passenger you must needs have a purse, and a purse is but a rag unless you have something in it. Besides, passengers get sea-sick—grow quarrelsome—don't sleep of nights—do not enjoy themselves much, as a general thing;—no, I never go as a passenger; nor, though I am something of a salt, do I ever go to sea as a Commodore, or a Captain, or a Cook. I abandon the glory and distinction of such offices to those who like them. For my part, I abominate all honorable respectable toils, trials, and tribulations of every kind whatsoever. It is quite as much as I can do to take care of myself, without taking care of ships, barques, brigs, schooners, and what not. And as for going as cook,—though I confess there is considerable glory in that, a cook being a sort of officer on ship-board—yet, somehow, I never fancied broiling fowls;—though once broiled, judiciously buttered, and judgmatically salted and peppered, there is no one who will speak more respectfully, not to say reverentially, of a broiled fowl than I will. It is out of the idolatrous dotings of the old Egyptians upon broiled ibis and roasted river horse, that you see the mummies of those creatures in their huge bakehouses the pyramids. No, when I go to sea, I go as a simple sailor, right before the mast, plumb down into the forecastle, aloft there to the royal mast-head. True, they rather order me about some, and make me jump from spar to spar, like a grasshopper in a May meadow. And at first, this sort of thing is unpleasant enough. It touches one's sense of honor, particularly if you come of an old established family in

Figure 10: An example (concluded) of the use of column cutouts.

Documents, Compuscripts, Programs, and Macros

Jonathan Fine
203 Coldhams Lane, Cambridge, CB1 3HY England
J.Fine@pmms.cam.ac.uk

Abstract

Certain aspects of the history and nature of the TeX typesetting program are described. This leads to a discussion of strategies for possible future developments. For clarity, the key terms document, compuscript, program and macro are defined.

The main argument is that improved macro packages and .dvi file processors will solve many problems, and that a rigorous syntax for input compuscripts should be developed and used. Such a strategy will allow a different and superior typesetting engine, should such arise, to be used in the place of TeX. It will also allow the same compuscript to be used for other, non-typesetting, purposes.

The Beginning

Much has changed since the creation of TeX by Donald Knuth in the years around 1980. Many millions now use computers for document preparation and production, and these computers are many times more powerful than those so used in 1980. Laser printers are now cheap and commonplace. PostScript has become a widely available standard for driving phototypesetters. The occupation of specialists has become a widespread daily activity. Much indeed has changed.

TeX is one typesetting system among dozens if not hundreds, counting not only DTP packages but also the various word processors available. Here are some of TeX's particular characteristics

- extremely reliable and bug-free
- available on almost all machines
- available at no or low cost
- constant unchanging behaviour
- portable ASCII input
- high quality output
- mathematical setting capabilities
- programmability via macros

which leave it without rival for use by the scientific scholarly community, and elsewhere.

TeX has limitations. If it did not, it could not be. Hegel wrote, 'that one who will do something great must learn to limit oneself'. It was wise of Knuth, not to create a text editor for use with TeX. Nor did he create general indexing or cross-referencing tools. Nor a spell-checker. All but the most basic functions are omitted, to be supplied by macros and parameter values. This gives a great flexibility, and reduces the decisions and labor involved in writing the program. Knuth supplies a basic collection of 'plain' macros. But even that most basic part of computer typesetting, persuading an output device to emit a typeset page, this vital part of the system lies outside the limited system for which Knuth himself took responsibility.

Indeed, this abdication of responsibility is a master stroke. The output devices are numerous, diverse, and more are yet to come. Therefore, typesetting is brought to a stop with the production of the .dvi file, which is a rigorously specified description of the location of every character and rule on the page. Each implementation is then responsible for transforming this .dvi file to meet the requirements of the various output devices. Because there is a rigorous standard for .dvi files, this separation of duties is a pleasant cooperation. Moreover, the same .dvi standard and processors can now be used by other typesetting systems, new and yet to be.

Knuth did not write editor, indexer, or output device driver. Nor did he write more than a few thousand lines of macros. He did write TeX the program (and METAFONT, and the Computer Modern fonts). To support this activity he also wrote the WEB system for documentation or literate programming. The skillful use of this tool has contributed greatly, I believe, to the high quality and thus durability of TeX. This lesson needs must be well learnt and comprehended by those who seek to provide an improved replacement.

I think it very important to understand just what it is we have with TeX. Richard Palais (1992)

gives another, balanced, discussion of the nature of TeX, with which I am in broad agreement. Frank Mittelbach (1990) carefully investigates and describes some of the typesetting limitations of TeX. Philip Taylor (1992) exaggerates the deficiencies of TeX. In particular, of his list (pages 438 - 440) of 10 claimed limitations, at least 5 (namely 1, 3, 4, 5 and 6) are quite possible with TeX as it is today. The same applies (see Jonathan Fine (to appear)) to his goal (page 441) of a multiwindowed interactive display. There is a difference, it is important to note, between interacting with a visual or graphic representation of a document (so far as I know Scientific Word is the only TeX-compatible system that allows this) and having immediate preview of the result of changes to the underlying ASCII representation (as provided by Textures for smaller documents). Philip Taylor (1992a) seems to have no relevance to our discussion.

Stability

It is 5 years since Knuth (1989) released version 3 of TeX, and 4 years since his announcement (Knuth 1990)

> My work on developing TeX, METAFONT, and Computer Modern has come to an end. I will make no further changes except to correct extremely serious bugs.

which triggered a continuing debate on how, or whether, a successor to TeX should be provided. But much and more can be done with TeX as it is. Knuth wrote (*loc. cit.*)

> 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. Improved macro packages can be added on the input side; improved device drivers can be added on the output side.

and it is to these possibilities that we will now turn.

The purpose of a macro package is to transform an input document, written according to some rigorous or informal syntax, into a sequence of primitive typesetting commands, and thus, via the fundamental operations of line breaking, hyphenation, ligatures, boxes and glue, table formation and so forth have TeX the program produce typeset pages in the form of a `.dvi` file, and perhaps also some auxiliary text files. However, TeX does not contain a word-processor or text editor, and so offers little or no help in the composition of the input document.

Many benefits result from having a rigorously defined syntax for input documents, and so many problems disappear. Such rigor allows the same document to be processed in different ways for different purposes, such as editing, typesetting, spell-checking, on-line documentation, hypertext, or, if a program source file, compilation. Although this is not a new idea (see Charles Goldfarb (1990), pages 7-8)

> Markup should describe a document's structure and other attributes rather than specify processing to be performed on it, as descriptive markup need be done only once and will suffice for all future processing.
>
> Markup should be rigorous so that the techniques available for processing rigorously-defined objects like programs and data bases can be used for processing documents as well.

none of the existing TeX macro packages is able to so typeset such a rigorously marked-up document. Moreover, the usual response to an error in markup is to have TeX the program generate an error message or worse, not generate an error. This behaviour is not a failing of TeX the program. Rather, it is a opportunity for improvement on the input side. The author has such work in progress.

It is worth noting that Knuth's WEB system made such a dual use (typesetting and compilation) of a single input file. This he did by writing two preprocessing programs (WEAVE and TANGLE) that convert a WEB input file into TeX and Pascal input files. For future reference note that although TeX source files are portable to any machine which has TeX installed, WEB files require the additional programs WEAVE and TANGLE to be also present.

On the output side, much can be done with `.dvi` files, provided suitable programs are available. By means of `\specials`, the device driver can be instructed to insert change bars, rotate tables, greyscale or color fonts, and so forth. All this is possible now, with TeX as it is, provided suitable programs are available.

It should be well understood that support for color, rotated tables, and other such goodies is not a matter of changing or 'improving' TeX the program. Rather, it requires matching facilities in the macro package used and in the `.dvi` file processor. TeX the program has no more involvement with the printing process that the moveable type typesetter of old, whose labor is blind to the color of the ink, or texture of the paper, used for the printing. Of course, the typographer or designer cares, or should, about these things.

There are other possibilities. Words to be indexed can be tagged using \specials (or even the whole word placed within the \special) and then extracted from the .dvi file. There are several advantages to this method. Firstly, it avoids the problems due to the asynchronous nature of the *output routine*, and also due to the expansion of macros during the \write command. Secondly, it allows the indexing software to extract additional information from the .dvi file, such as the location on the page (either by line or by physical location). Thirdly, this last data may be useful for hypertext applications. One can even cut-and-paste among .dvi files (see Asher 1992, von Bechtolsheim 1989, and Spivak et al. 1989). All this is possible so long as the TeX macros are properly set up, and so long as the .dvi file processing programs are available.

It is worth noting here that the work of the DVI driver standards committee (Reid and Hosek 1989, and Schrod 1991) seems to support my contention, that much remains to be done, to get the best out of what is already available to us. Lavagnino (1991), and Vesilo and Dunn (1993) discuss examples of how some applications require that much more than printed pages be produced. These problems can be solved by means of a suitable combination of macros and .dvi file processing programs.

Growth

This then is the background against which our use of TeX develops, and into which any successor will be introduced. TeX can still reach the highest typographical standards. But it seems that it is precisely in those areas, such as input file preparation and post-processing of the output file, which lay outside the limits that enabled Knuth's achievement, that the TeX system is deficient.

In particular, the lack of a front end for document preparation, that exploits the computing and graphical display capabilities that so many users now have available (and so few when TeX was first written) is a major obstacle to more widespread acceptance.

Elsewhere (Fine, to appear) I have indicated how TeX as it is today (and will be, major bugs aside, for the rest of time) can be used as the typesetting engine for such a visual document preparation system. However, any such will require programs that are specific to the architecture and capabilities of the host machine.

Much more can be done with TeX than is commonly realised. It is a powerful typesetting engine that can be turned to many purposes. Except for particular typographic functions (see Mittelbach 1990), such as detection and hence control of rivers of white space in paragraphs, most or all of its perceived limitations can be overcome by a judicious combination of improved macros and auxiliary programs. I have much work in progress (and less completed than I would like to admit) on improving macros.

The difficulty with auxiliary programs is that they are not automatically portable in the same manner as TeX the program is, and that they tend to become numerous and subject to change, much like macro packages.

A singular virtue of TeX, as vital to its success as the ground upon which we walk, and as commonly appreciated, is that it provides a programming environment, available and identical in operation on all machines. This is the TeX macro language. It is the basis for the portability of TeX documents. Moreover, transfer of such programs is no more than transfer of ASCII files.

Imagine now that we have a similar foundation for the writing of .dvi file processors. All manner of problems would go away, or at least be mitigated. There are about 10 standards for using \specials to access PostScript. The lack of a macro language gives an unwanted rigidity to the .dvi file processors, and so each standard is (or is not) hard-coded into each particular .dvi program.

Many indexing and hypertext problems can be resolved by post-processing the .dvi file, but not in a portable manner unless the .dvi processing program is similarly portable. Elsewhere (Fine, to appear) I have indicated how a visual front end to TeX can be assembled out of a suitable combination of a previewer (which is itself a .dvi file processor), a .dvi file editor, and TeX as it is but running a suitable and rather special macro package.

For such to be flexible, its outer form must be controlled by macros or the like. For such to be portable, the supporting programs must be both portable and ported.

Definitions

In order that my conclusions be stated as precisely as is possible, I will make some definitions.

By a *document* I will mean a physical graphical and perhaps substantial object containing text in various fonts, and perhaps other items such as symbols and photographs. Examples of a document are a book, a magazine or journal, a preprint, and a restaurant menu. These are substantial items, in the sense of their being made out of stuff. The

quality of the ink and paper, and the impression of the one on the other, are subtle aesthetic qualities of the document, in no sense determined by the typesetting process.

However, I will also regard an image on the screen of a computer to be a document, although of the insubstantial or un-stuffy kind. Such documents allow a different range of interactions with the reader, usually called the user, than the printed page. Indeed, in external form many computer programs are documents in this broad sense.

By a *compuscript*, or script for short, I mean a finite sequence of symbolic or numerically coded characters, such as ASCII, satisfying a formal or informal syntax. It may also contain references to external entities, which may be other documents, or to non-document elements such as photographs or illustrations. It is sometimes convenient to break a script down into complements, which are either mark-up or text. The syntax is then a system of rules which relate the mark-up to the text. Examples of compuscripts are TEX and LATEX document source files (these have an informal syntax), and SGML and program source files (which have a rigorous syntax).

By a *program* I mean an executable binary file. Program files cannot be read as a comprehensible sequence of characters. They contain machine instructions that are specific to the host machine on which the program is to be run. Properly written, programs will run as quickly as any software can to perform their given function, but to change a program is usually a slow and sometimes laborious process. Knuth wrote TEX the program and METAFONT the program. More exactly, he wrote documents which were then transformed via a compiler and other tools (literate programming) into versions of TEX the program, one for each machine architecture. He also wrote the 'plain' macros for TEX, and the Computer Modern source files for METAFONT.

We can now say what *macros* are. A collection of macros is a compuscript which controls or influences the operation of a program. This definition includes both the configuration or option files that many programs use to store system data and user preferences, but also the macro files used by TEX and METAFONT, or any other code written to be executed by an interpreting program. The distinction between a program and macros is not always clear-cut. For example, many microprocessors contain microcode which is called upon to perform various functions. Emulation is often achieved by expanding machine code for one processor into sequences of machine instructions for another. If not present, it is common to emulate machine instructions to a mathematics coprocessor.

The US photographer Ansell Adams compared the negative to the score for a piece of music, and the print to the performance. Adams is famed for his marvellous atmospheric photographs of Yosemite National Park. Developing his photographic analogy (is it a rule that every article should have one bad pun?), the compuscript is the negative for the production of a document, the program the fixed darkroom equipment, while the macros are the consumeable papers and chemicals and also the skill, habits, standards and creativity of the darkroom operator. Incidentally, many negatives require special human activity related to their content such as 'dodging' and 'burning' (this means giving more or less exposure to different parts of the negative) in order that they come out at their best.

Note added in Proof

There are several articles also in these proceedings that bear upon the topics discussed here. Rokicki expresses the idea of a programmable `.dvi` file processor, although as an implementor his focus is more on what is immediately possible or practical. I should have realised for myself the important 'color' motive, whose difficulties in the production setting are well expressed by Sofka. Laugier and Haralambous describe Philippe Spozio's interactive and visual `.dvi` file editor, and also Franck Spozio's TEX to SGML translation tools. These programs go some way to resolving, for documents marked up in the traditional plain TEX or LATEX manner, various real world problems, which are among the motives for the point of view I adopt in my article.

The deficiencies of TEX are once again exaggerated by Taylor. It is possible, for example, to typeset material on a grid, to flow text around insertions, to treat the two-page spread or even the chapter as the region over which page make-up and optimisation are performed, all this is possible with today's TEX, by writing admittedly tricky macros. The goal of Schrod is to provide a formal model of TEX the program (particularly its macro facilities) with which a user can interact, whereas my goal is to have formal syntax for compuscripts that can be understood by TEX (given suitable macros) and by the user alike.

Finally, the papers of Baxter, Ogawa, and Downes discuss progress and problems in the typesetting of structured documents—again, using traditional TEX macro tools. It is my contention that the macro development and performance difficulties that they face can be greatly eased by

the introduction of powerful development tools, amongst which will be sophisticated macros that will combine compuscript parsing macros with style sheet values to give rise to the document production macros.

Conclusions

It should now be clear that Knuth is responsible for only one part of the TeX typesetting system, although that part is its mighty heart or engine. It is my opinion that, good though they are, there is considerable room for improvement in those parts of the TeX system that Knuth did not provide, viz. macros and `.dvi` file processors.

Perhaps in the next 20 years, someone will write a worthy successor to TeX. This would be, like TeX itself, a great achievement. To supplant TeX, it will need to be substantially better. I would expect such a system to continue to use more-or-less if not exactly the same `.dvi` file format as TeX. It would be nice if both TeX and its successor shared at least one syntax for the compuscripts that are to be processed into documents. This will surely require that both operate to a syntax that is as rigorous as that for the `.dvi` files. Work on defining such a syntax and creating suitable TeX macros to process such documents can begin today, without knowing what the future may bring, but all the same helping to bring it about.

To hope for compatibility at the level of macros or format files is probably too much, and likely to be self-defeating. Fortunately, many though formats are, they are, or at least should be, few in relation to documents.

TeX as it is today can be used as the engine of an interactive and visual typesetting system. I encourage all those who want to write programs to join with me in turning this possibility into a reality. A valuable first step, with independent benefits and merits of its own, would be to write a 'universal' `.dvi` file processor that is controlled by macros, just as TeX is a universal typesetting engine.

If all is done properly, and to rigorous standards for both input and output, then it will be a simple matter to replace TeX the program by the new and much improved engine, when and if it arrives. Indeed, part of the whole strategy is to provide a clear rôle and interface for the typesetting engine.

Donald Knuth has not written much on successors to TeX. It is thus our responsibility to read carefully what he has written. I close by repeating his advice quoted earlier

> 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. Improved macro packages can be added on the input side; improved device drivers can be added on the output side.

Bibliography

Asher, Graham. "Inside Type & Set", *TUGboat*, **13** (1), pages 13-22, 1992.

Bechtolsheim, Stephan von. "A `.dvi` file processing program", *TUGboat*, **10** (3), pages 329-322, 1989.

Clark, Malcolm. "NEXTTeX: A personal view", *TUGboat*, **14** (4), pages 374-380, 1993.

Fine, Jonathan. "Editing `.dvi` files, or visual TeX", *TUGboat*, (to appear)

Goldfarb, Charles. *The SGML Handbook*, Oxford University Press, 1990

Knuth, Donald E. "The new versions of TeX and METAFONT", *TUGboat*, **10** (3), pages 325-328, 1989.

Knuth, Donald E. "The Errors of TeX", *TUGboat*, **10** (4), pages 529-531, 1989.

Knuth, Donald E. "The future of TeX and METAFONT", *TUGboat*, **11** (4), page 489, 1990.

Lavagnino, John. "Simultaneous electronic and paper publication", *TUGboat*, **12** (3), pages 401-405, 1991.

Mittelbach, Frank. "E-TeX: Guidelines for future TeX", *TUGboat*, **11** (3), pages 337-345, 1990.

Palais, Richard. "Moving a fixed point", *TUGboat*, **13** (4), pages 425-432, 1992.

Reid, Tom and Don Hosek. "Report from the DVI driver standards committee", *TUGboat*, **10** (2), pages 188-191, 1989.

Schrod, Joachim. "Report on the DVI Driver Standard", *TUGboat*, **12** (2), pages 232-233, 1991.

Spivak, Michael, Micheal Ballantyne, and Yoke Lee. "HI-TeX cutting & pasting", *TUGboat*, **10** (2), pages 164-165, 1989.

Taylor, Philip. "The future of TeX", *TUGboat*, **13** (4), pages 426-442, 1992.

Taylor, Philip. "NTS: the future of TeX?", *TUGboat*, **14** (3), pages 177-182, 1992.

Vesilo, R.A. and Dunn, A. "A multimedia document system based on TeX and DVI documents", *TUGboat*, **14** (1), pages 12-19, 1993.

Integrated system for encyclopaedia typesetting based on TEX

Marko Grobelnik, Dunja Mladenić, Darko Zupanič, Borut Žnidar
Artificial Intelligence Laboratory, J. Stefan Institute, Ljubljana, Slovenia.
name.secondname@ijs.si

Abstract

The paper presents a system used for several already published dictionaries, lexicons and encyclopaedias. The system is based on a TEX macro package accompanied by many special purpose utilities for editor (person dealing with contents and form) support.

Introduction

Editing and typesetting of different kinds of encyclopaedic books (encyclopaedias, dictionaries, lexicons) is a highly specialized endeavour compared with the typesetting and editing of ordinary texts. First of all, these kinds of books are produced as relatively high budget projects in commercial companies. Many people are involved in such projects, which typically have well established working procedures already. Next, these kinds of texts are never really finished. Book releases are always only better prepared stages of contents which continue to develop (e.g. language in dictionaries). From a technical point of view it is important to mention the huge amount of text in such books and simultaneous appearance of several national languages along with all their peculiarities.

To build a successful system one should consider all the aforementioned properties of such book making. In the following few sections we will briefly describe the history of our involvement in organising and executing encyclopaedic book projects, actual solutions, the pros and cons of our work and some prospects for the future.

History

Our involvement in encyclopaedic book typesetting started rather accidentally few years ago. The Slovenian publishing house *Cankarjeva založba* was in the process of editing and publishing the book titled *The Encyclopaedia of the Slovenian language* dealing with all possible aspects (grammatical, historical, linguistic, ...) of the Slovenian language. The attempt with the classical way to publish the book in a printing house appeared to be very inappropriate and expensive, because of the large quantity of very technical text, with many major revision changes. The publishing house tried to use a standard interactive desktop publishing package to accomplish the job. The attempt failed again. After that, we decided to do it with TEX, which led to successful completition of the project. We made, of course, some mistakes, but this proved a useful experience for further work.

Because of the successful start, we were asked to build a general system for editing and typesetting the encyclopaedic type of books. Firstly, we adapted ourselves to the already established organisation of work in the publishing house, changing it slightly in the direction of the automatisation of all possible phases of work. The first project on which we developed our system, was the composition of several smaller dictionaries. After that we got the job to technically organise the work for the biggest Slovenian general lexicon *Sova* (in English *Owl*), where we finally developed the technology and the system. Currently, we are involved in several minor and major projects, the biggest being *The Encyclopaedia of Slovenia* (12 books + index).

Solutions

Because of the existing practice in the Slovenian publishing houses, the system was prepared for IBM-PC, although all components are portable. The use of the system is text-editor independent, however, we suggest the use of open and flexible integrated environments (e.g. TEX-Shell, Emacs, Borland-IDE, ...).

For the purpose of the common text input, a language called LEX was defined. The language is primarily entry-oriented with special elements like pictures, capitals, phonetic support, entry qualifying and many commands for semantic structuring of text. Characters used for LEX constructs are independent of the natural character set. The whole text corpus of a book is written in LEX format.

The following is an entry written in LEX format from *The English-Slovene Modern Dictionary*:

```
<entry:lx>
<head:action> <ipa:"aeKŠN>
  dejanje; delovanje, proces; tožba;
<p:to be killed in ~> pasti v boju;
<p:to put into ~> sprožiti, pognati;
<p:to take ~> ukrepati;
<p:out of ~> pokvarjen, izločen;
<p:social ~> družbena akcija
<end>
```

For the printout, the text in LEX is processed in three passes. First, the text in LEX is converted to the TEX format with a separate utility program which also performs several consistency controls. Next, TEX needs two passes. In the first pass correct picture positions and column lengths are determined. The second TEX pass builds final layout. The whole 3-pass process for 400 kbytes of text takes approximately 2 minutes on an IBM-PC i486/66 computer.

Besides typesetting, additional software was developed for editorial support which works on text in LEX format. This includes dictionary inversion, multi-author support (e.g. one published book had 40 authors), sorting support, etc.

Pros and Cons

Advantages of our system are:

- Working with the system and LEX format is extremely simple. For most of the books prepared with our system, only three people were involved: author(s), editor and typist.
- There is no need to change text editor habits. The only demand for the text editor used, is ability to export ASCII files.
- The cycle time between the corrections in the text and the printout of the finallayout is in the range of minutes.
- Additional editorial support is provided with text-manipulation utilities.
- The system is designed to support multilingual texts (dictionaries).
- The system is easily extended.
- The system runs on any platform with TEX. Minimal platform is IBM-PC i386 with DOS.
- The system was tested on several real encyclopaedic books, some of them very extensive and complex.

Disadvantages are the following:

- The system is not WYSIWYG (is this really a disadvantage?).
- When preparing the text for the final printout two things must be done manually: unresolved hyphens (narrow columns) and picture repositioning (to achieve an artistic look).

Conclusions and Future prospects

We have presented a system for editing and typesetting of encyclopaedic type of books. The system is based on TEX with additional utilities for editorial support. All components are hidden within an integrated environment. For the purpose of text input, we have defined a language called LEX, which allows us full control across the text corpus for checking and other text manipulation operations.

Our plans for the near future are to make a complete commercial product for dealing with encyclopaedic type of books along with all necessary interactive typesetting features.

An Example of a Special Purpose Input Language to LaTeX

Henry Baragar
Instantiated Software Inc., 20 Woodmount Crescent, Nepean, Ontario, K2E 5R1 Canada.
henry@instantiated.on.ca

Gail E. Harris
RES Policy Research Inc., 6th Floor, 100 Sparks Street, Ottawa, Ontario, K1P 5B7 Canada.
ak753@freenet.carleton.ca

Abstract

A special purpose language for documenting knowledge bases demonstrates how LaTeX can be augmented to add expressiveness for specific situations. The language, called TESIA, enables expert system analysts to mark up groups of rules into tables in a way which reflect the logical structure of the knowledge base. The TESIA style options generate LaTeX tables for use by expert system programmers and the equivalent English text typeset in a subsection for use by domain experts. This paper presents the syntax and implementation of this special purpose language. Despite the complex output requirements, the TeX implementation has proven to be very flexible and remarkably short.

Introduction

If I have seen further than other men,
it is because I have stood on the shoulders of the giants.

—Isaac Newton

The logical treatment of documents is one of LaTeX's most important features. A benefit of this approach is that the source files for most LaTeX documents are usually almost as readable as the final output. As is true with any general purpose tool, there are cases that are not easily expressed in the input language of the tool. In this case, a special purpose language (or "little language"), as advocated by Jon Bentley (1990, page 83), can be of great benefit. A well-designed "little language" — in which the special case can be easily expressed — follows more closely the philosophy of LaTeX than does the contortion of LaTeX commands to achieve a desired result.

This paper presents a special purpose language for documenting knowledge bases which has a much more natural syntax than pure LaTeX for marking up the rules of a knowledge base. It has been used successfully to typeset the system documentation for the knowledge base portion of an application on a project where the documentation tool of choice for the rest of the system was Microsoft Word. We begin by describing problems with documenting knowledge bases. Then we present the "little language" that was designed specially for documenting knowledge bases, and show how it was implemented in TeX, yielding a special purpose input language to LaTeX. This is followed by some observations on the suitability and success of the solution. We conclude with a discussion of future directions and some recommendations for others wishing to implement special purpose languages, and in particular special purpose input languages to LaTeX.

The Challenge

The problem of documenting knowledge bases was encountered on a project where an existing knowledge base with no external documentation had to be maintained and expanded. The first step in the project was to document, or reverse engineer, the knowledge base. This in itself is a challenge because expert system analysts are still struggling to find effective methods to document knowledge bases. Some methods, such as KADS[1], are too high level and do not document individual rules. Other lower level methods are usually tools tied to specific products — products not being used on this project. This project required a tool for documenting knowledge bases at the rule level, but not tied to a specific product.

The challenge, to the expert system analyst, in documenting the rules of a knowledge base is in the need to present the documentation to two audiences. The first audience, the expert system programmer, uses the documentation to program the rules in the knowledge base. The second audience, the system owner or domain expert, uses the documentation to verify the correctness of the rules in the knowledge

[1] Although "KADS" was an acronym at one time (Knowledge Acquisition and Documentation System), it has changed and it is now considered a proper name in itself.

Goldilocks' Rules		
Conditions		Conclusion
These are the rules that model Goldilocks' decision process.		
T < min		too-cold
	T > max	too-hot
min ≤ T	T ≤ max	just-right

Figure 1: The tabular form for the expert system programmer.

Goldilocks' Rules

These are the rules that model Goldilocks' decision process.

If the temperature is less than the minimum acceptable temperature **then** the porridge is too cold.

If the temperature is greater than the maximum acceptable temperature **then** the porridge is too hot.

If the minimum acceptable temperature is less than or equal to the temperature **and** the temperature is less than or equal to the maximum acceptable temperature **then** the porridge is just right.

Figure 2: The English form for the domain expert.

base. The tabular presentation of Figure 1, preferred by the expert system programmer, is usually incomprehensible to the domain expert, who prefers English sentences and paragraphs, as in Figure 2. The challenge of accurately presenting both sets of documentation is often so great that the domain expert is often given inadequate summaries of the rules or is left to struggle with just the tabular representation of the rules. This often leads to a loss of confidence in the Expert System, as had happened on the project in question.

The challenge of presenting two sets of documentation would be considerably simplified if they could both be generated from the same source. This is not possible in Microsoft Word, the tool specified for documentation in this particular project. Considering the differences between the tabular form and the English language form illustrated in Figures 1 and 2, it was not even clear this would be possible in LaTeX. Nor was it clear that a LaTeX source file would be easily readable and maintainable. Thus, the challenge was to find a mechanism to document the rules of the knowledge base in a single source file, where the structure of the rules is visually apparent to the expert system analyst and where the documentation sets are appropriate to their intended audience.

The New Input Language

The best way to ensure that the structure of the rules is visually apparent in a source file documenting a knowledge base is to develop a new syntax for marking up rules that has a clean visual presentation. In this section, we present a special purpose language, or "little language" à la Jon Bentley (page 83), that has a syntax with the desired properties. We leave the details of implementing the language until the next section.

The syntax of the new input language — called TeX Expert System Language (TeSLA)[2] — is very simple and has only five commands. These commands can be divided into three groups: definitions, groups of rules, and other commands. A clean visual presentation of the source file has been achieved by defining a syntactic structure for these commands which allows an ASCII text source file to be modeled after the layout of the tabular representation to be presented to the expert system programmer; this reflects the common backgrounds of the expert system analyst and the expert system programmer.

Definitions. A variable in a knowledge base is documented by giving an English language phrase that defines the variable. TeSLA allows variable names from the knowledge base to be used directly in the TeSLA source file. The knowledge base variable is left unchanged when it is presented to the expert system programmer, whereas it is mapped to the English Language phrase when it is presented to the domain expert. A TeSLA definition, which has the following syntax:

`[tvar␣|␣`*KB-var*`␣|␣`*English description*`␣]`

is used to specify the mapping of the knowledge base variable to its English Language description. An example of a definition is:

`[tvar␣|␣T␣|␣the␣temperature␣]`

which defines the knowledge base variable `T` as the phrase "the temperature". That is, a reference to the knowledge base variable `T` in a TeSLA rule is represented by the string "T" in the tabular form presented to the expert system programmer, whereas it is represented by the string "the temperature" in the English form presented to the domain expert.

[2] The language was developed for the Travel Expert System (TES) project, which also explains why all the commands begin with a "t".

Groups of rules. The rules of a knowledge base are usually documented as groups of related rules. For the expert system programmer this means that a group of related rules is presented as a table, whereas the group is presented as a subsection to the domain expert. A group of rules begins with:

`[tgroup␣|␣`*Group Name*`␣|␣`*n*`␣]`

where *Group Name* is a label for the group and *n* is the maximum number of *conditions*, excluding the *conclusion*, in the rules in this group. In the tabular form presented to the expert system programmer, *n* is one less than the number of columns in the table.

A group of rules ends with:

`[tgroup␣|␣`*Group Name*`␣|␣e␣]`

where *Group Name* should be the same as at the beginning of the group.

Occasionally, it may be desirable to visually separate subgroups of rules within a large group of rules. This is accomplished with the:

`[tgroup␣|␣`*Group Name*`␣|␣-␣]`

command, which inserts a horizontal line (`\hline`) into the table. Currently, it does nothing in the English form presented to the domain expert.

A rule in TESLA has the following syntax:

`[trule␣|␣`*cond1*`␣|␣`*cond2*`␣|␣...␣|␣`*concl*`␣]`

where *cond1*, *cond2*, ..., are the *n* conditions of the rule and *concl* is the conclusion of the rule.

Each condition — as well as the conclusion — is a *relation* that has one of the following forms:

lhs␣*rel*␣*rhs*
rel␣*rhs*␣_
rhs␣_␣_

where *lhs* and *rhs* are TESLA variables, and *rel* is a relation operator. In the tabular form, each condition and the conclusion is put in its own column. With suitable groupings of rules and arrangements of relations within columns, an expert system programmer can easily check that all possible combinations of relations have a known conclusion and that no two rules conflict with one another. The English form given to the domain expert, on the other hand, has every variable, relation, and implicit conjunction spelled out in full.

As an example, consider the rules already presented in Figure 1 and Figure 2, which would appear in the TESLA source file as:

```
[tgroup | Goldilocks' | 3 ]
[trule | T < min  |            | too-cold _ _   ]
[trule | _        | T > max    | too-hot _ _    ]
[trule | min <= T | T <= max   | just-right _ _]
[tgroup | Goldilocks' | e ]
```

Note that this code fragment lacks the variable definitions and the command to add the extra descriptive text found in the figures. Also note that a quirk in the implementation requires that leading empty conditions must have a single "_" character, as in the `too-hot` rule above.

Other commands. There are two commands in TESLA for adding annotations to the rules. The first, which has the following syntax:

`[ttext␣|␣`*text*`␣]`

provides a mechanism for adding arbitrary explanatory text into both the tabular and the English forms. The second, which has the following syntax:

`[trem␣|␣`*text*`␣]`

provides a mechanism for adding extra text, for remarks, only to the tabular form used by the expert system programmer. There has yet to be a requirement to add text to the English form used by the domain expert which is not also required by the expert system programmer.

Other syntax. There is little requirement for additional syntax in TESLA. Syntax was added to TESLA to treat all text between the "`;`" character and the end of a line as source file comments. The "%" character was rejected for introducing comments because percentages are used frequently in the knowledge base on this project. All other considerations for adding syntax have been rejected because of the extra effort that would be required to explain them.

The Implementation

Now that the syntax of TESLA has been defined, the implementation details can be discussed. Two approaches to implementation were considered: either, build a preprocessor, or implement TESLA directly in (LA)TEX. At first, it seemed that the preprocessor approach would be easier to implement. This had the advantage that the output could be switched to Microsoft Word code if and when a definition of the file format for Microsoft Word could be found. However, good string manipulation tools, such as `perl` and awk, needed to implement the preprocessor were not readily available for the target environment (Microsoft DOS). Thus, the approach to implement TESLA directly in (LA)TEX was selected.

TESLA is implemented in TEX as three style options. The first, `tesla.sty`, has the definitions of the TESLA commands described above as well as all the other definitions common to both output forms. The other two files are `eng-form.sty`, which contains code specific to the English form, and `tab-form.sty` which contains code specific to the tabular form. This section begins by describing how to use these style files. Then it defines the implementation of TESLA in detail as coded in the style files.

The structure of TESLA documents. TESLA documents are composed of three main files and one or more rule files. The main files are usually named `main-eng.tex`, `main-tab.tex` and `main.tex`. The

first two of these files, which are used to select the form, simply contain:

```
\documentstyle[tesla,form]{article}
\input{main}
```

where the *form* is either `eng-form` or `tab-form`, depending on whether it is in the file `main-eng.tex` or `main-tab.tex`, respectively. As can be seen from this code, these files input the file `main.tex`, which is the real TESLA document.

To input a rule file into the main TESLA file, the following command is used:

```
\inputrulefile{rule-file}
```

where *rule-file* is the name of a file containing TESLA code. This command is defined as:

```
\newcommand{\inputrulefile}[1]{%
  \changecatcodes\input{#1}}
```

in `tesla.sty`.

The `\changecatcodes` command changes the categories of the digits, arithmetic and relation operators, and a few other characters to category 11, the same category as the alphabetic characters. This allows these characters to be used in TESLA variable names and enables a broad range of variable names, including operators and numbers! The `\changecatcodes` command also changes the category of the "`;`" character to category 14 to make it the comment character, which is the "%" character in (LA)TEX. Finally, `\changecatcodes` changes the category of the "`[`" character to category 0 to make it an escape character, the same category as the "`\`" character in (LA)TEX. This allows the commands of TESLA, such as `[tvar`, to be implemented directly as TEX commands.

The `[tvar` command. The `[tvar` command, like all the TESLA commands, makes use of TEX's pattern-matching capability to implement TESLA syntax. It is defined in `tesla.sty` as:

```
\gdef\tvar␣|␣#1␣|␣#2␣]{%
  \expandafter\gdef\csname #1\endcsname
    {\xform{#1}{#2}}}
```

There are two things to note in this definition:

1. The command has been *defined* as `\tvar` even though it is *used* as `[tvar`. The reason is that the category of the "`[`" character has not yet been set to category 0 (the same category as "`\`") as it will be when a rule file is read.
2. The spaces in the `\gdef\tvar␣|␣#1␣|␣#2␣]` are important. The implementation of TESLA relies on the fact that TEX compresses all strings of white space to a single "␣", which means that the arguments to `[tvar` can be spaced out to improve the readability of the rule files. Although this may not be important for `[tvar` commands, it is important for `[trule`.

The `[tvar` command relies on the `\xform` command, which is defined differently in `eng-form.sty` than it is in `tab-form.sty`, to do its work. All of the TESLA commands use the same mechanism, where a command beginning with "`\x`" is called to implement the real behavior, to achieve different behaviors depending on whether the English form or the tabular form is to be presented. The desired behavior for `[tvar` is that a TESLA variable is defined as the definition supplied by the user in the English form and as itself in the tabular form. This is accomplished by defining `\xform` as:

```
\gdef\xform#1#2{#2}
```

in `eng-form.sty`, and as:

```
\gdef\xform#1#2{#1}
```

in `tab-form.sty`. Thus, for the TESLA variable `T` from our Goldilocks example, we now have `\T` which expands to "the temperature" in the English form and to "T" in the tabular form.

The `[tgroup` command. The `\tgroup` command is really several commands and it looks at its second argument to determine which command to run. It is defined in `tesla.sty` as:

```
\gdef\tgroup␣|␣#1␣|␣#2␣]{%
\if 1#2 \xbegin{xone}{1}{2}{.466}{#1}\fi
\if 2#2 \xbegin{xtwo}{2}{3}{.300}{#1}\fi
\if 3#2 \xbegin{xthree}{3}{4}{.216}{#1}\fi
\if 4#2 \xbegin{xfour}{4}{5}{.166}{#1}\fi
\if 5#2 \xbegin{xfive}{5}{6}{.133}{#1}\fi
\if 6#2 \xbegin{xsix}{6}{7}{.1095}{#1}\fi
\if -#2 \xsep\fi
\if e#2 \xend\fi
}
```

where `\xbegin`, `\xsep` and `\xend` are defined differently, depending on the form.

The definition for `\xbegin` in `eng-form.sty` is to define `[trule` to be the command specified by the first argument and to introduce a new subsection using the fifth argument. It is implemented as:

```
\gdef\xbegin#1#2#3#4#5{%
  \gdef\trule{\csname #1\endcsname}
  \subsection*{#5 Rules}}
```

The second, third and fourth arguments to `\xbegin` are ignored in `eng-form.sty`.

The definition for `\xbegin` in `tab-form.sty` is much longer (10 lines) and is not presented here. Like the definition in `eng-form.sty`, the definition of `\xbegin` in `tab-form.sty` defines `[trule` to be the command specified by the first argument. Then it introduces the code necessary to set up a `tabular` environment where the number of columns is specified by the third argument and where each column is typeset as a `parbox` whose width is specified by the fourth argument[3] multiplied by the text width

[3] The values of the fourth argument were determined by experimentation.

(\textwidth). The second argument always has a value of one less than the third argument and is required in order to avoid having to do arithmetic in TeX. It is used to specify the number of columns to be spanned by the "Conditions" heading. The fifth argument is used in the title of the table. Finally, the \xbegin makes some minor adjustments to the tabular environment to improve the visual presentation of the tables.

The \xsep command is defined to do nothing in eng-form.sty and to insert a horizontal line — using \hline — in tab-form.sty.

Similarly, the \xend command is defined as nothing in eng-form.sty, and as

```
\gdef\xend{\hline\end{tabular}}
```

in tab-form.sty.

The [trule command. As was seen in the discussion on [tgroup, the [trule command is defined in \xbegin to be one of \xone through \xsix. These commands are all very similar with \xtwo, for example, being defined in tesla.sty as:

```
\gdef\xtwo␣|␣#1|␣#2|␣#3]{%
  \if_#1␣\xpre␣\xone␣|␣#2|␣#3]%
  \else\xif{#1}␣\xand{#2}\xthen{#3}
  \fi
  }
```

Note the lack of spaces between each argument and the "|"or "]" character that follows the argument ("...␣#1|␣#2|␣#3]...") which preserves a space at the end of the argument and which will be used as a delimiter when the argument itself is parsed.

The \xtwo code says that if the first condition has been set to the character "_", then this is like a one-rule condition: do something specific to the form (\xpre) and call \xone. Note that \xthree will call \xtwo and \xfour will call \xthree, etc. The \xpre does nothing in the English form but is required in the tabular form to insert a "&" character to skip the first column. Otherwise, if the first argument is not the "_" character, then build up the rule using \xif, \xand and \xthen.

In the English form:

```
\xif{#1} \xand{#2}\xthen{#3}
```

produces the expected result of:

If *#1* **and** *#2* **then** *#3*.

except when *#2* is empty, in which case the *and-clause* is elided. This is implemented as:

```
\gdef\xif#1{\par {\bf If} \xrel #1}
\gdef\xand#1{\ifx#1\empty
             \else{\bf and} \xrel #1
             \fi
  }
\gdef\xthen#1{{\bf then} \xrel #1.}
```

in eng-form.sty.

In the tabular form:

```
\xif{#1} \xand{#2}\xthen{#3}
```

produces the expected result of putting the *if-clause* into the first column, the *and-clause* (if there is one) into the second column, and the *then-clause* into the third column. This is implemented in tab-form.sty as:

```
\gdef\xif#1{\RS\xrel #1}
\gdef\xand#1{& \RS\ifx#1\empty
                  \else\xrel #1
                  \fi
  }
\gdef\xthen#1{& \PBS\RS\xrel #1 \\}
\gdef\RS{\raggedright\sloppy\hspace{0pt}}
```

where \PBS is the \PreserveBackslash command as described in Goossens et al. (1994, page 108). The command \xrel is discussed below.

The \xrel command forms the heart of the TeSLA style. It takes three arguments: *lhs*, *rel* and *rhs*, as described above in the description of the TeSLA [trule command. Each argument must end in a space; this is why the spaces were left in by the \xone through \xsix commands. If the third argument, or the second and third arguments, is the character "_", then these arguments are elided. This is necessary to prevent extra space being inserted, particularly in the English form and especially just before a period. The \xrel command is implemented as:

```
\gdef\xrel␣#1␣#2␣#3␣{%
  \if_#3\if_#2\xvar{#1}%
        \else\xvar{#1}␣\xvar{#2}%
        \fi
  \else\xvar{#1}␣\xvar{#2}␣\xvar{#3}%
  \fi
  }
```

in tesla.sty. Once again, the spaces between \xvar commands are important, this time to put spaces between the text expansions of *lhs*, *rel* and *rhs* in the English form.

Finally, the \xvar command simply expands the TeSLA variable passed to it from \xrel. If the variable is undefined, then the variable name is used, typeset in italics. It is implemented as:

```
\gdef\xvar#1{\expandafter
   \ifx\csname #1\endcsname\relax{\it #1}%
   \else \csname #1\endcsname
   \fi
  }
```

in tesla.sty. Note that the different representations for the different forms have already been encoded in the variable by the [tvar command.

The [ttext command. The [ttext command is defined in tesla.sty as:

```
\gdef\ttext␣|␣#1␣]{\xtext{#1}}
```

where \xtext is defined as \par{\em #1} in the English form, and as

```
\gdef\xtext#1{%
 \multicolumn{\numcols}{|p{.9\textwidth}|}
   {\em #1}\\\hline
 }
```

in tabular form. Note that `\numcols` was defined by the `[tgroup` command.

The `[trem` command. The definition of the `[trem` command is analogous to the definition of `[ttext`. However, `\xrem` is defined as nothing in the English form and as `\xtext{\sc #1}` in the tabular form.

Pre-defined variables. Since the relationship and arithmetic operators are treated like normal TESLA variables, it is trivial to predefine many of the these operators in `tesla.sty`. For example,

```
\gdef\<{\xform{$<$}{is less than}}
```

predefines the < relation.

Observations

We hope that our paper has shown the TEX implementation of the TESLA language is elegant and remarkably compact. The `tesla.sty` file is only 102 lines (of undocumented TEX code), the `tab-form.sty` is 32 lines and the `eng-form.sty` is 18 lines. The implementation in TEX was less difficult than anticipated. It is also shorter than the anticipated preprocessor solution, yet is at least as robust and flexible. It also has the benefit of handling the inclusion, with some restrictions, of LATEX code into the rules.

Comparison with "pure" LATEX. The improvement in the visual presentation of the source code of TESLA compared with "pure" LATEX is striking. Consider our Goldilocks example as it might be written in LATEX:

```
\begin{tgroup}{Goldilocks'}
\trule{\T\LT\min}{          }{\tooCold}
\trule{          }{\T\GT\max}{\tooHot}
\trule{\min\LE\T}{\T\LE\max}{\justRight}
\end{tgroup}
```

where the content of the document is obscured by too many "\", "{" and "}" characters.

The dictionary. At one point in the project, there was a great rush to produce a dictionary of the knowledge base variables. It was a simple matter to search the source files for all lines with `[tvar`, sort this list, and process it with a simple style that implements the `[tvar` command as an item in a description list — all in less than half an hour. This activity revealed several duplicate variable definitions that might not otherwise have been caught, and forms a counterpart to the implementation of `[tvar` which typesets, in italics, undefined variables as their variable name. This reinforces the advantages of separating the logical structure of a document from the details of typesetting.

Conclusions

I have stood on the shoulders of
Jon Bentley and Donald Knuth.
— Henry Baragar

The TESLA language has met its original goals. The structure of the rules is visually apparent in a TESLA source file and it has been used successfully for a knowledge base with more than 270 rules using over 250 variables. The two forms of output have been well received by their intended audiences. Surprisingly, some of the expert system programmers have found the English form has helped them to understand the context of the rules that they were reading in the tabular form, a context that is sometimes lost in the brevity of using only variable names.

Spurred by the success of the implementation of TESLA, we would like to enhance the functionality of the language. First, we would like to expand the `[tgroup` command to express relationships between the tables, which then could be graphed and included in the documentation. Second, we would like to enhance the `[trule` command to generate code for a particular Expert System shell, which would significantly reduce the consistency problems between the documentation and the code. This capability could be extended to multiple Expert System shells.

This example of a special purpose input language to LATEX illustrates the utility of application-specific mark-up languages and the suitability of using TEX for the implementation. We hope this example will encourage others to consider creating "little languages" in TEX in those cases where the logical structure of their documents is lost in the typesetting commands in their source files. We certainly have found the benefits have been extraordinary and the difficulties surprisingly minor.

Acknowledgements

We would like to thank Christina Thiele who was the one to finally convince us to write this paper and was kind enough to preview it for us. Also, we would like to thank Christine Detig who reviewed the paper and provided helpful comments.

Bibliography

Bentley, Jon, *More Programming Pearls: Confessions of a Coder*, Reading Mass.: Addison-Wesley, 1990.

Goossens, Michel, Frank Mittelbach, and Alexander Samarin, *The LATEX Companion*, Reading Mass.: Addison-Wesley, 1994.

Knuth, Donald, *The TEXbook*, Reading Mass.: Addison-Wesley, 1989.

Lamport, Leslie, *LATEX: A Document Preparation System*, Reading Mass.: Addison-Wesley, 1986.

Appendix

The `tesla.sty` file.

```
\setlength{\parskip}{\baselineskip}
\setlength{\parindent}{0pt}

\gdef\empty{}

\gdef\tvar | #1 | #2 ]{%
    \expandafter\gdef\csname #1\endcsname{\xform{#1}{#2}}%
    }
\gdef\tgroup | #1 | #2 ]{%
    \if 1#2 \xbegin{xone}{1}{2}{.466}{#1}\fi
    \if 2#2 \xbegin{xtwo}{2}{3}{.300}{#1}\fi
    \if 3#2 \xbegin{xthree}{3}{4}{.216}{#1}\fi
    \if 4#2 \xbegin{xfour}{4}{5}{.166}{#1}\fi
    \if 5#2 \xbegin{xfive}{5}{6}{.133}{#1}\fi
    \if 6#2 \xbegin{xsix}{6}{7}{.1095}{#1}\fi
    \if -#2 \xsep\fi
    \if e#2 \xend\fi
    }
\gdef\ttext | #1 ]{\xtext{#1}}
\gdef\trem | #1 ]{\xrem{#1}}

\gdef\xone | #1| #2]{%
    \xif{#1} \xthen{#2}
    }
\gdef\xtwo | #1| #2| #3]{%
    \if_#1 \xpre \xone | #2| #3]%
    \else\xif{#1} \xand{#2}\xthen{#3}
    \fi
    }
\gdef\xthree | #1| #2| #3| #4]{%
    \if_#1 \xpre \xtwo | #2| #3| #4]%
    \else\xif{#1} \xand{#2}\xand{#3}\xthen{#4}
    \fi
    }
\gdef\xfour | #1| #2| #3| #4| #5]{%
    \if_#1 \xpre \xthree | #2| #3| #4| #5]%
    \else\xif{#1} \xand{#2}\xand{#3}\xand{#4}\xthen{#5}
    \fi
    }
\gdef\xfive | #1| #2| #3| #4| #5| #6]{%
    \if_#1 \xpre \xfour | #2| #3| #4| #5| #6]%
    \else\xif{#1} \xand{#2}\xand{#3}\xand{#4}\xand{#5}\xthen{#6}
    \fi
    }
\gdef\xsix | #1| #2| #3| #4| #5| #6| #7]{%
    \if_#1 \xpre \xfive | #2| #3| #4| #5| #6| #7]%
    \else\xif{#1} \xand{#2}\xand{#3}\xand{#4}\xand{#5}%
        \xand{#6}\xthen{#7}
    \fi
    }

\gdef\xrel #1 #2 #3 {%
   \if_#3\if_#2\xvar{#1}\else\xvar{#1} \xvar{#2}\fi
   \else\xvar{#1} \xvar{#2} \xvar{#3}%
   \fi
```

```
    }
\gdef\xvar#1{%
    \expandafter
        \ifx\csname #1\endcsname\relax{\it #1}%
        \else \csname #1\endcsname%
        \fi
    }

\newcommand{\inputrulefile}[1]{%
    \changecatcodes
    \input{#1}
    }

\newcommand{\changecatcodes}{
    \catcode`\+=11          \catcode`\0=11
    \catcode`\-=11          \catcode`\1=11
    \catcode`\*=11          \catcode`\2=11
    \catcode`\(=11          \catcode`\3=11
    \catcode`\)=11          \catcode`\4=11
    \catcode`\<=11          \catcode`\5=11
    \catcode`\!=11          \catcode`\6=11
    \catcode`\>=11          \catcode`\7=11
    \catcode`\:=11          \catcode`\8=11
    \catcode`\==11          \catcode`\9=11
    \catcode`\_=11
    \catcode`\[=0
    \catcode`\;=14
    }

{
\changecatcodes
\gdef\<{\xform{$<$}{is less than}}
\gdef\>{\xform{$>$}{is greater than}}
\gdef\!={\xform{$\neq$}{is not equal to}}
\gdef\=={\xform{$=$}{is equal to}}
\gdef\<={\xform{$\leq$}{is less than or equal to}}
\gdef\>={\xform{$\geq$}{is greater than or equal to}}
\gdef\:={\xform{$\leftarrow$}{is assigned}}
\gdef\+{\xform{$+$}{added to}}
\gdef\+={\xform{$+$$=$}{is incremented by}}
\gdef\decrement{\xform{$-$$=$}{is decremented by}}
\gdef\minus{\xform{$-$}{less}}
\gdef\*{\xform{$*$}{multiplied by}}
\gdef\memberOf{\xform{$\in$}{is one of}}
\gdef\notMemberOf{\xform{$\not\in$}{is not one of}}
\gdef\isnot{\xform{$\neg$}{not}}
}
```

The `eng-form.sty` file.

```
\gdef\xform #1#2{#2}

\gdef\xbegin#1#2#3#4#5{
    \gdef\trule{\csname #1\endcsname}
    \subsection*{#5 Rules}
    }
\gdef\xsep{}
\gdef\xend{}
```

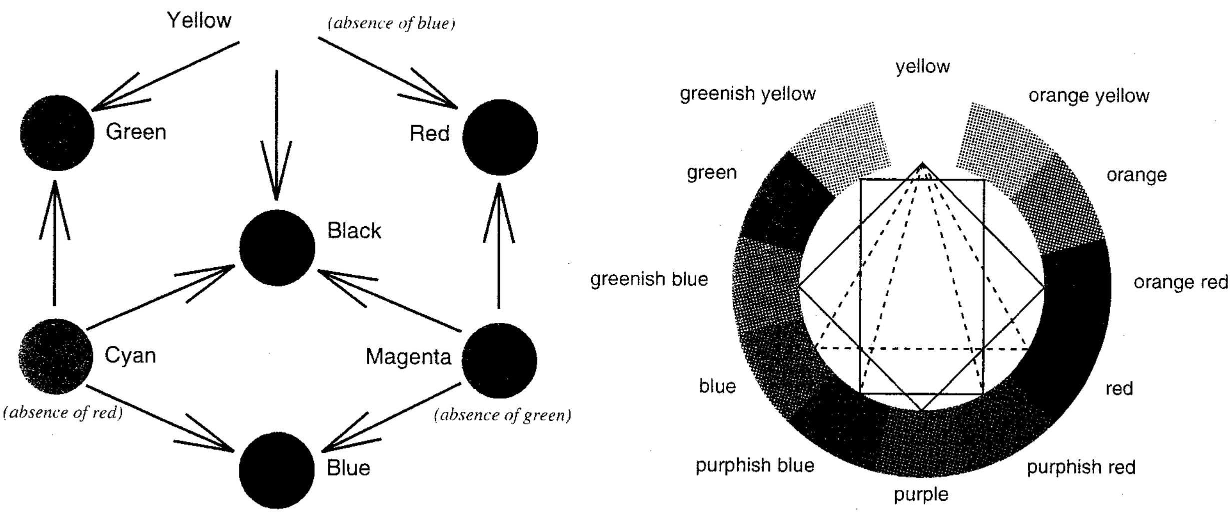
```
\gdef\xpre{}

\gdef\xif#1{\par {\bf If} \xrel #1}
\gdef\xand#1{\ifx#1\empty\else{\bf and} \xrel #1 \fi }
\gdef\xthen#1{{\bf then} \xrel #1.}

\gdef\xtext#1{\par{\em #1}}
\gdef\xrem#1{}
```

The `tab-form.sty` **file.**

```
\gdef\xform#1#2{#1}

\gdef\xbegin#1#2#3#4#5{
    \gdef\trule{\csname #1\endcsname}
    \gdef\numcols{#3}
    \par\begin{tabular}{*{#3}{|p{#4\textwidth}}|}
    \hline
    \multicolumn{#3}{|c|}{\rule{0pt}{2.8ex}\large\bf #5 Rules}\\
    \hline
    \multicolumn{#2}{|c|}{\rule{0pt}{2.8ex}\large Conditions}
        & \large Conclu\-sion \\
    \hline\hline
    }
\gdef\xsep{\hline}
\gdef\xend{\hline\end{tabular}}

\gdef\xpre{&}

\gdef\xif#1{\RS\xrel #1}
\gdef\xand#1{& \RS\ifx#1\empty\else\xrel #1\fi}
\gdef\xthen#1{& \PBS\RS\xrel #1 \\}

\gdef\xtext#1{\multicolumn{\numcols}{|p{.9\textwidth}|}{\em #1}\\
              \hline}
\gdef\xrem#1{\xtext{\sc #1}}

%see LaTeX Companion, page 132 (for \hspace{0pt})
\gdef\RS{\raggedright\sloppy\hspace{0pt}}

%see LaTeX Companion, page 108
\gdef\PreserveBackslash#1{\let\temp=\\#1\let\\=\temp}
\let\PBS=\PreserveBackslash
```

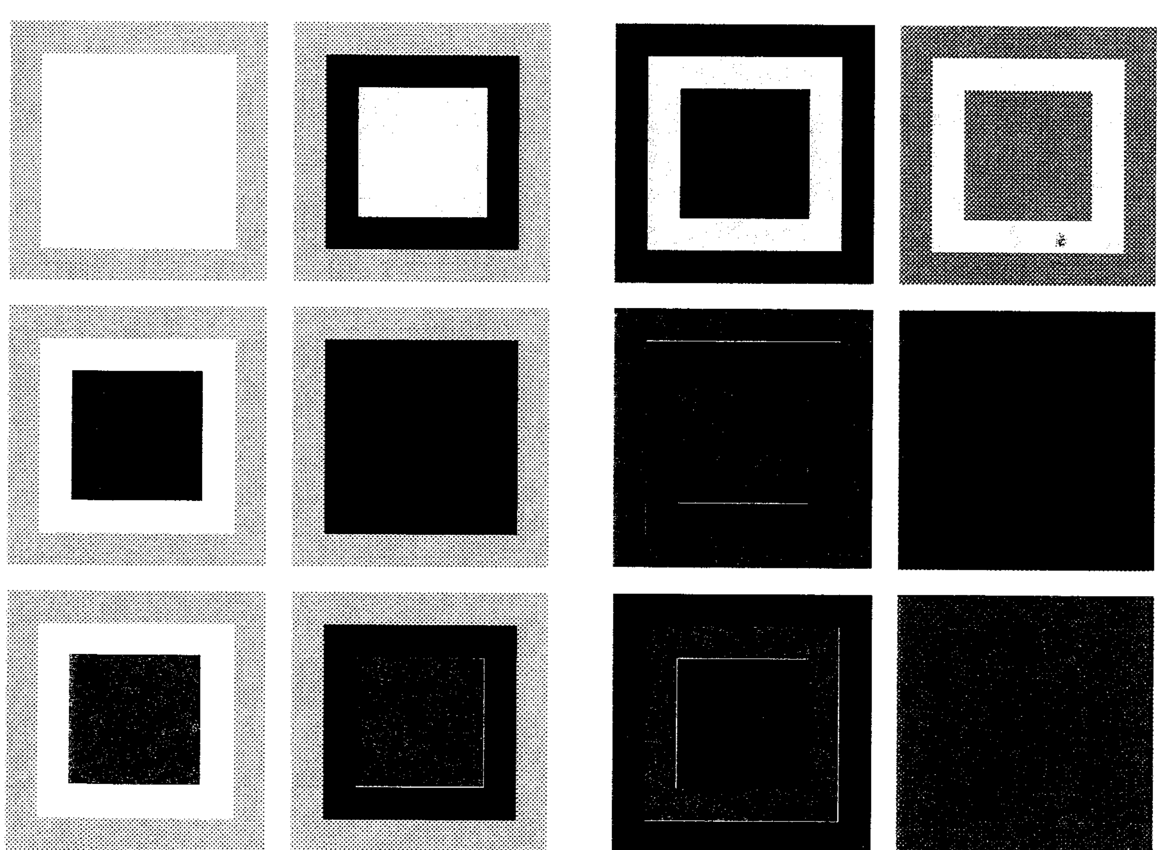
Appendix

Color figures

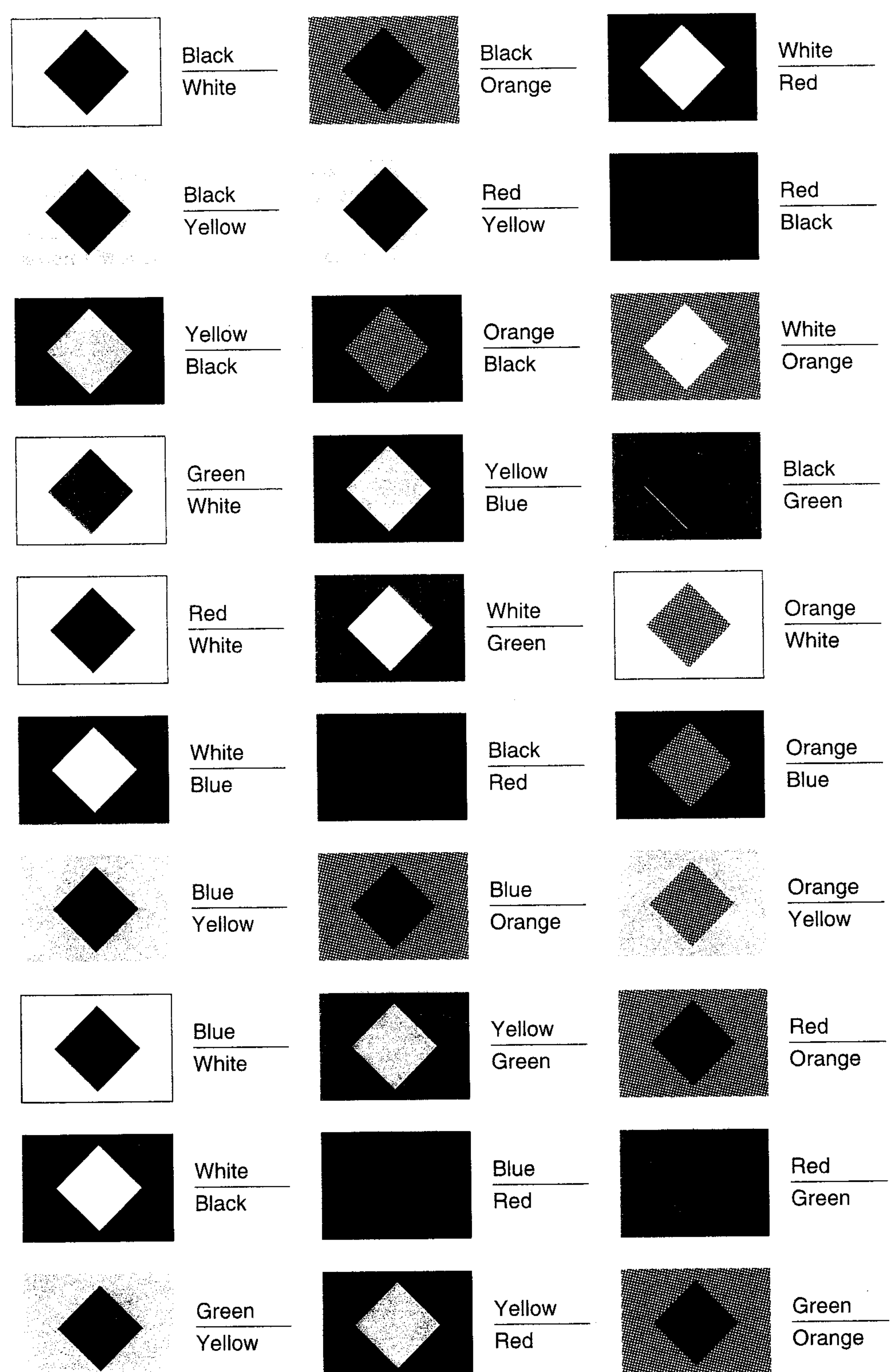


Color Example 1: The RGB and CMYK colour models

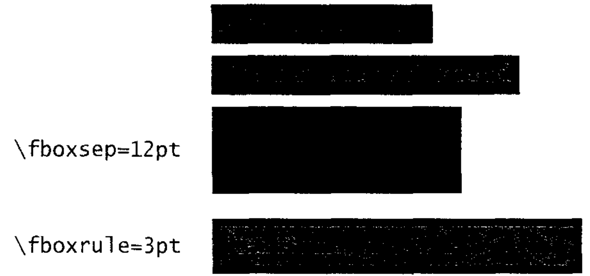
Color Example 2: Colour harmonies and the chromatic circle



Color Example 3: Facets of colour harmony for the primary process colours



Color Example 4: Effective colour contrasts for maximum visibility and redability



Color Example 5: Examples of colored and framed boxes in $\LaTeX2_\varepsilon$

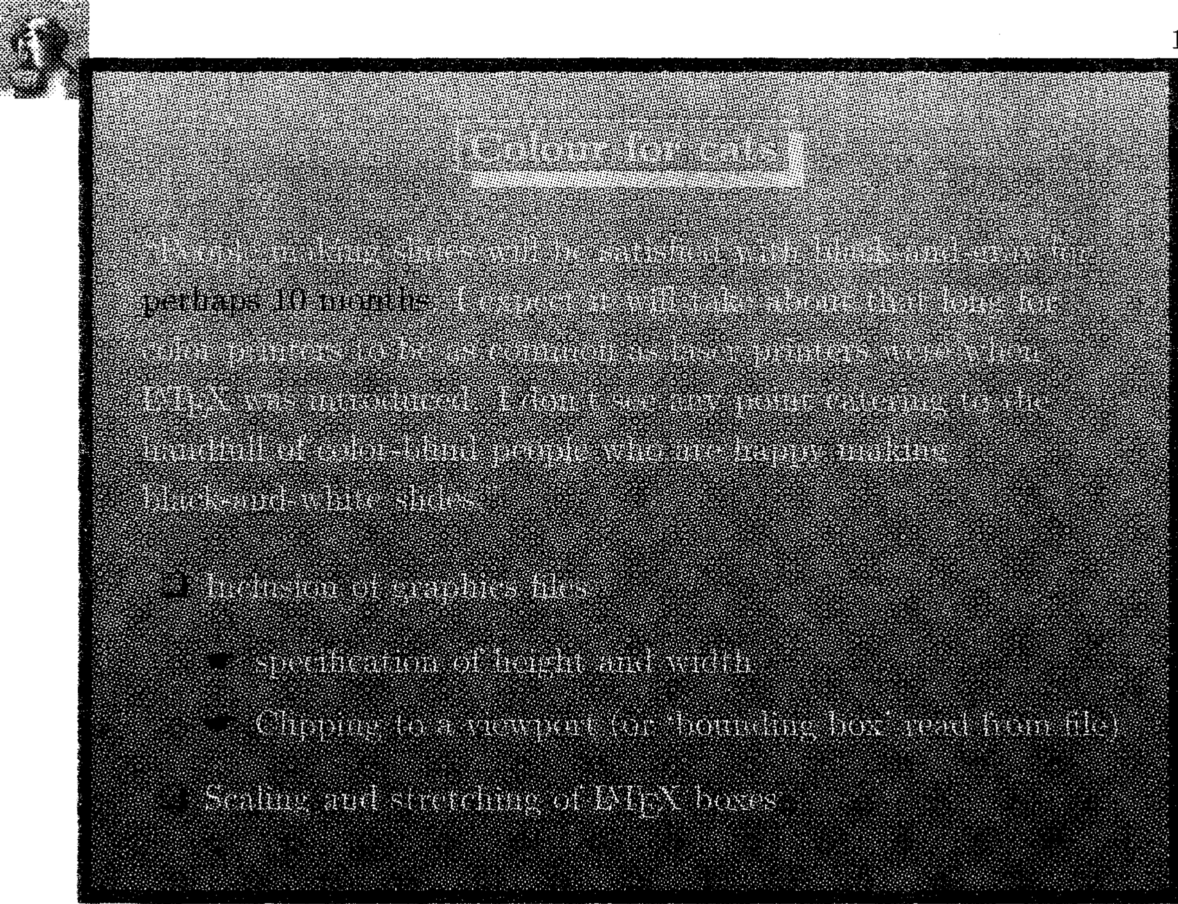
known : ℙ *NAME*
birthday : *NAME* ⇸ *DATE*

known = dom*birthday*
where*birthday* = 13.2.55

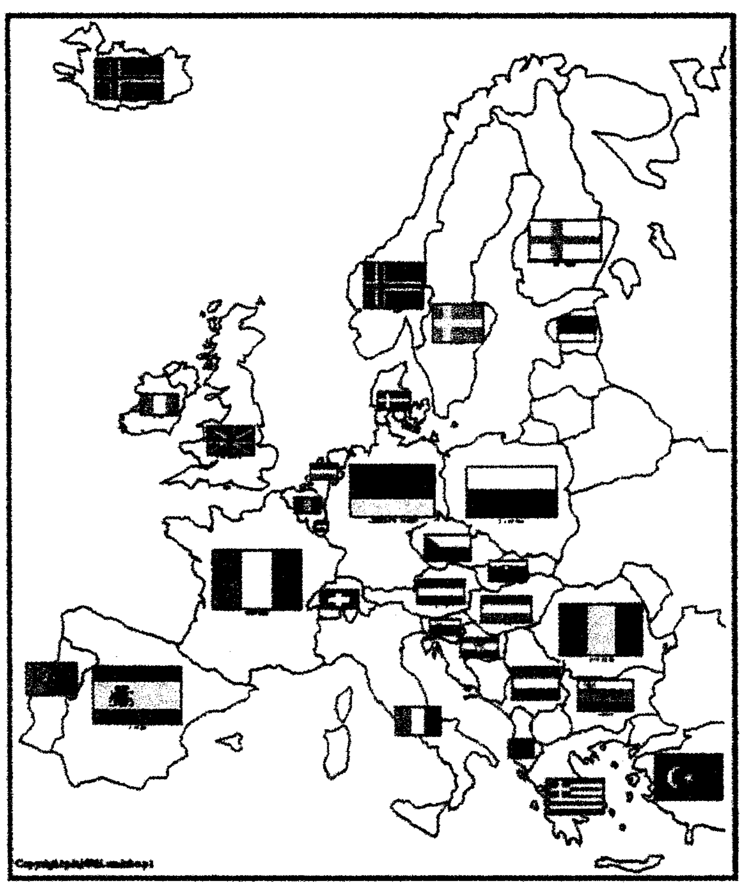
Color Example 6: Z schema using colour to mark keywords

ORLANDO					MCO
Glasgow	OG4G	Thu 20/10	Mon 31/10 or 07/11	11 or 18 days	
		Thu 27/10	Mon 31/10 or 07/11	4 or 11 days	
	OG7A	Sun 23/10	Mon 31/10 or 07/11	8 or 15 days	
		Sun 30/10	Mon 07/11	8 days	

Color Example 7: A coloured table

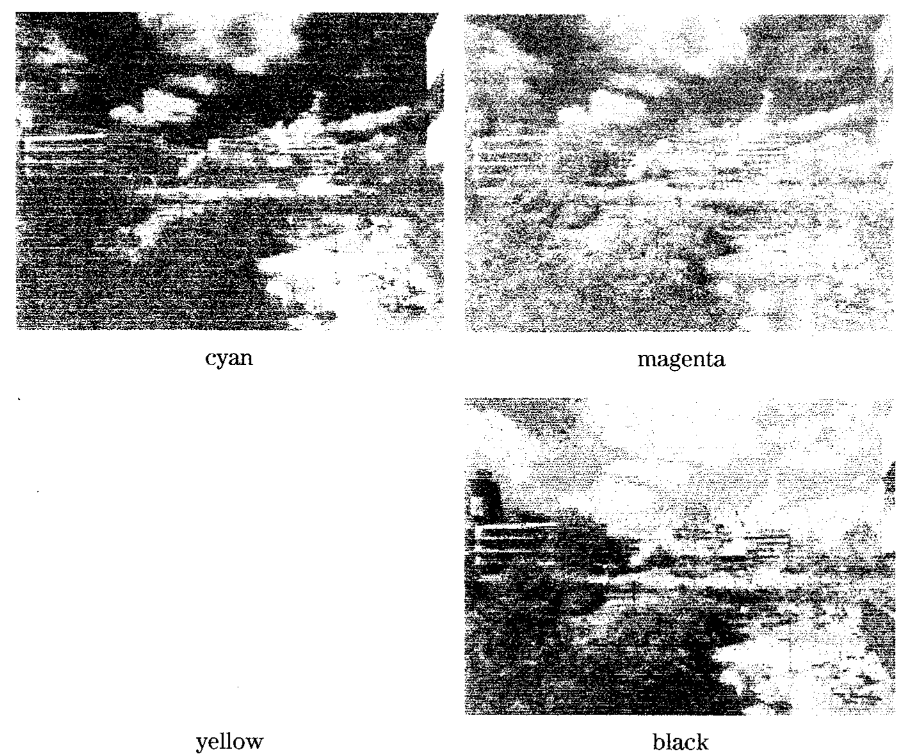


Color Example 8: Colour slide with colour list

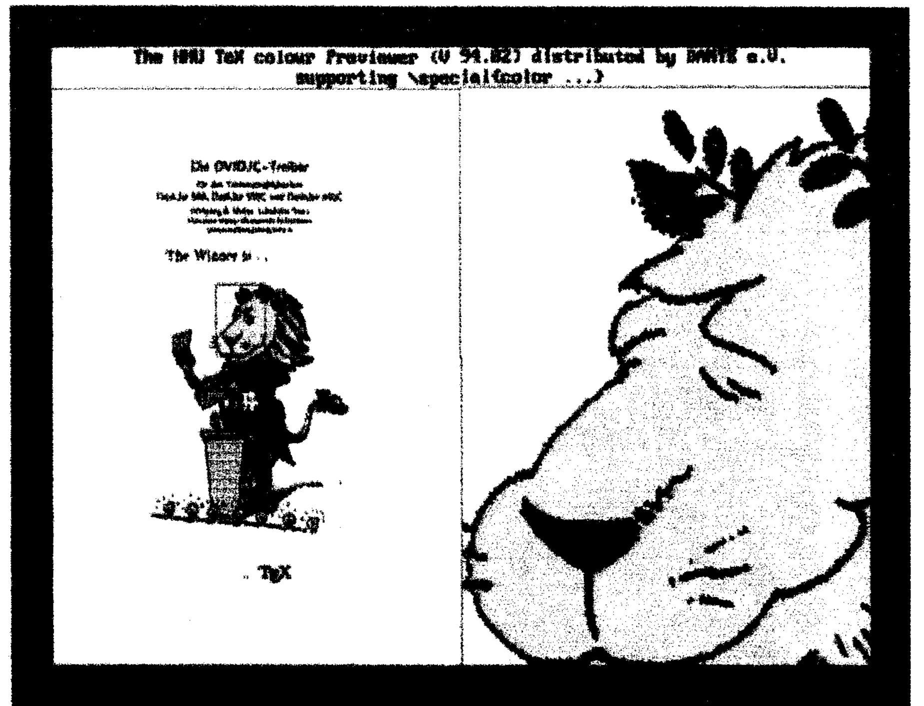


Color Example 9: WWW Map of European Home Pages

Color Example 10: Campus at the University of Dortmund



Color Example 11: Colour separations of previous colour figure



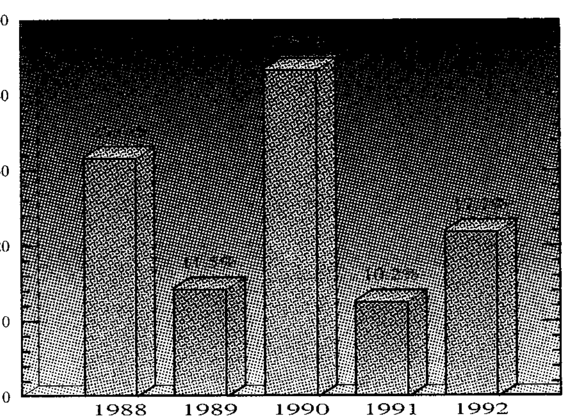
Color Example 12: Colour preview with `dvivgac`

Happy Birthday!!

ann

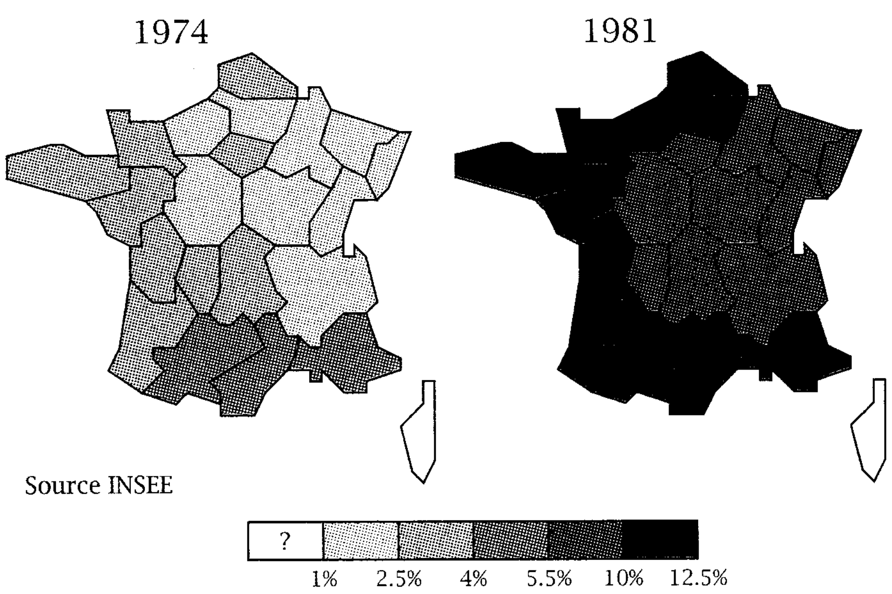
Color Example 13: **Typesetting text on a path, filling character outlines, and using a TEX box as a fill pattern.**

Evolution of revenues

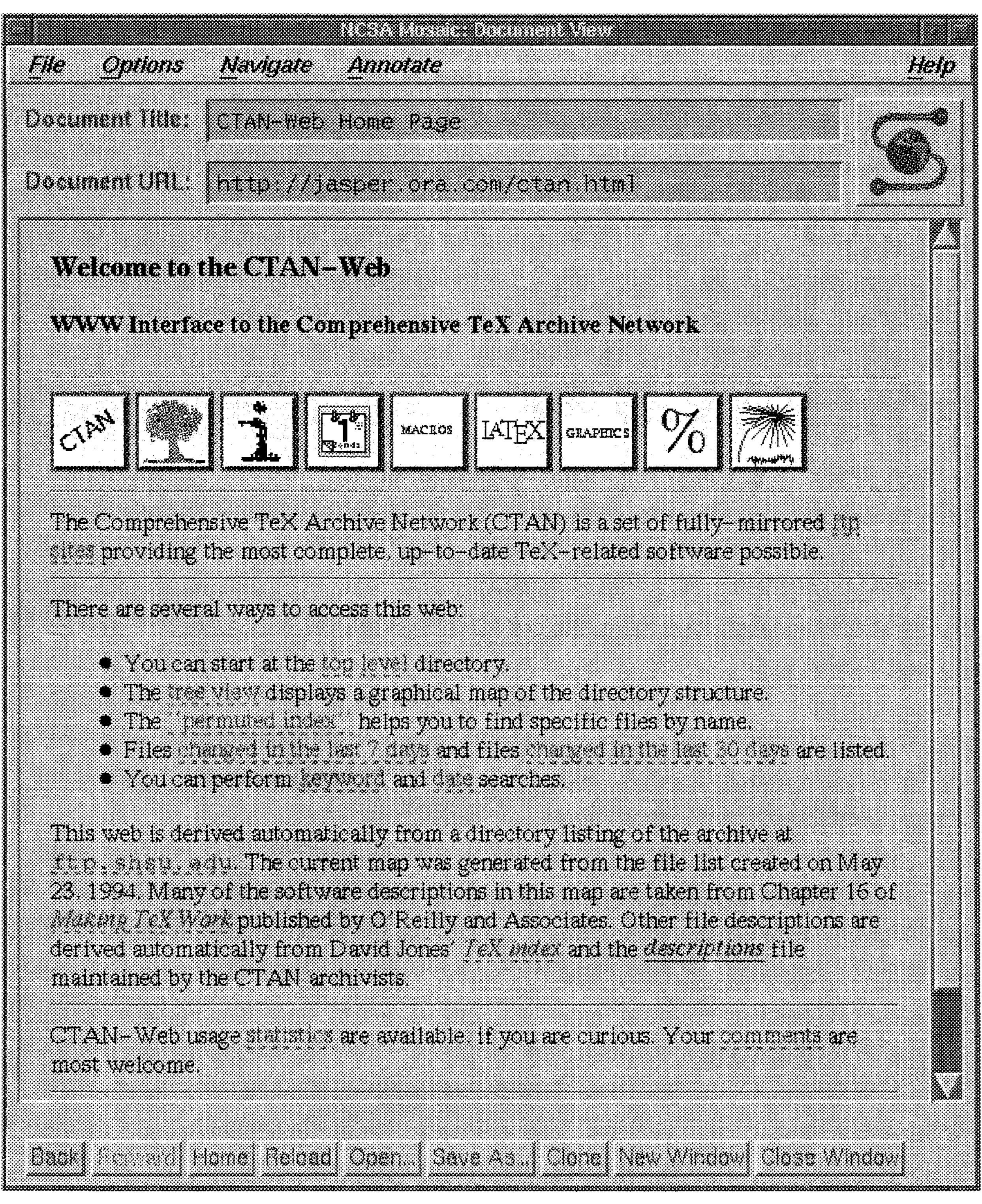


Color Example 14: A bar graph made with `pstchart`**.**

Unemployment rate in 1974 and 1981 in France



Color Example 15: Coloration of maps.



Color Example 16: The CTAN-Web home page.

Participants at the 15th Annual TUG Meeting July 31–August 4, 1994 Santa Barbara, California

Shelley-Lee Ames
University of Manitoba
Winnipeg, Manitoba, Canada
shelley@cmstex.maths.umanitoba.ca

Harumi Ase
Tokyo, Japan
ase@cicso.co.jp

Henry Baragar
Instantiated Software Inc
Nepean, Ontario, Canada
henry@instantiated.on.ca

Michael Barnett
Brooklyn College of the City of New York
Princeton, New Jersey, USA

William Baxter
SuperScript
Oakland, California, USA
web@superscript.com

Nelson H F Beebe
University of Utah
Salt Lake City, Utah, USA
beebe@math.utah.edu

Barbara Beeton
American Mathematical Society
Providence, Rhode Island, USA
bnb@math.ams.org

Arrigo Benedetti
University of Bologna
Modena, Italy
arrigo@cube.systemy.org

John Berlin
TEX Users Group
Santa Barbara, California, USA
john@tug.org

Arvind Borde
Long Island University
Cambridge, Massachusetts, USA
borde@bnlcl6.bnl.gov

Harriet Borton
Rensselaer Polytechnic Institution
Troy, New York, USA
bortonh@rpi.edu

Johannes Braams
Zoetermeer, The Netherlands
j.l.braams@research.ptt.nl

Peter Breitenlohner
Max-Planck-Institut für Physik
München, Germany
peb@dmumpiwh.bitnet

Tan Bui
Concordia University Computing
Montreal, Quebec, Canada
ilfc326@vax2.concordia.ca

Mary R Burbank
SCRI / Florida State University
Tallahassee, Florida, USA
mimi@scri.fsu.edu

Katherine Butterfield
Center for EUV Astrophysics
University of California at Berkeley
Berkeley, California, USA
kathb@cea.berkeley.edu

David Carlisle
Manchester University
Manchester, England
carlisle@cs.man.ac.uk

Lance Carnes
Personal TEX Inc
Mill Valley, California, USA
pti@crl.com

Leslie Chaplin
SFA Inc
Waldorf, Maryland, USA
chaplin@orion.nrl.navy.mil

Ampon Chumpia
Raj Burana
Bangkok, Thailand

Malcolm Clark
University of Warwick
Coventry, England
m.clark@warwick.ac.uk

David Cobb
SAIC/OSTI
Oak Ridge, Tennessee, US

Michael Cohen
University of Aizu
Aizu, Japan
mcohen@u-aizu.ac.jp

Arvin C Conrad
Menil Foundation
Houston, Texas, USA
acc@rice.edu

Jackie Damrau
Fluor Daniel
Red Oak, Texas, USA
damrau@amber.unm.edu

Donald DeLand
Integre Technical Publishing
Albuquerque, New Mexico, USA
deland@sc.unm.edu

Susan DeMeritt
IDA/CCR La Jolla
San Diego, California, USA
sue@ccrwest.org

Christine Detig
Technical University Darmstadt
Darmstadt, Germany
detig@iti.informatik.th-darmstadt.de

Luzia Dietsche
DANTE e.V
Heidelberg, Germany
secretary@dante.de

Michael Doob
University of Manitoba
Winnipeg, Manitoba, Canada
mdoob@cc.umanitoba.ca

Jean-Luc Doumont
JL Consulting
Zaventem, Belgium
j.doumont@ieee.org

Michael Downes
American Mathematical Society
Providence, Rhode Island, USA
mjd@math.ams.org

Ken Dreyhaupt
Springer-Verlag
New York, New York, USA
kend@springer-ny.com

Angus Duggan
Harlequin Ltd
Barrington, Cambridge, England
angus@harlequin.co.uk

Laura Falk
University of Michigan
Ann Arbor, Michigan, USA
lauraf@eecs.umich.edu

Michael Ferguson
INRS Telecommunications
University of Quebec
Verdun, Quebec, Canada
mike@inrs-telecom.uquebec.ca

Barbara Ficker
Northern Arizona University
Flagstaff, Arizona, USA

Peter Flynn
University College
Cork, Ireland
pflynn@curia.ucc.ie

Jim Fox
University of Washington
Seattle, Washington, USA
jmbrown@uci.edu

Yukitoshi Fujimura
Addison-Wesley Publishers
Tokyo, Japan
gbfoo174@niftyserve.or.jp

Bernard Gaulle
CNRS-IDRIS
Orsay Cedex, France
gaulle@idris.fr

Laurie Gennari
Stanford Linear Accelerator Center
Stanford, California, USA
gennari@slac.stanford.edu

Michel Goossens
CERN
Genève, Switzerland
goossens@cern.ch

Peter Gordon
Addison-Wesley Publishing Co
One Jacob Way
Reading, Massachusetts, USA

George D Greenwade
Sam Houston State University
Huntsville, Texas, USA
bed_gdg@shsu.edu

James Hafner
IBM Research
Almaden Research Center
San Jose, California, USA
hafner@almaden.ibm.com

Hisato Hamano
Impress Corporation
Tokyo, Japan
hisatoh@impress.co.jp

Genevra Hanke
Addison-Wesley Publishing Co.
Reading, Massachusetts, USA
nevh@aw.com

Yannis Haralambous
Lille, France
yannis@univ-lille1.fr

Robert L Harris
Micro Programs, Inc.
Syosset, New York, USA

Michael Hitchcock
ETP Service
Portland, Oregon, USA
mike@etp.com

Michael Hockaday
TSI Graphics
Effingham, IL, USA

Alan Hoenig
John Jay College/CUNY
Huntington, New York, USA
ajhjj@cunyum.cuny.edu

Anita Z Hoover
University of Delaware
Newark, Delaware, USA
anita@zebra.cns.udel.edu

Berthold KP Horn
Y&Y Inc
Concord, Massachusetts, USA

Blenda Horn
Y&Y Inc
Concord, Massachusetts, USA

Don Hosek
Quixote Digital Typography
Claremont, California, USA
dhosek@quixote.com

Charles Hurley
Elsevier Science Inc
New York, New York, USA
c.hurley@panix.com

Calvin Jackson
California Institute of Technology
Sherman Oaks, California, USA
calvin@csvax.caltech.edu

Alan Jeffrey
COGS
University of Sussex
Brighton, England
alanje@cogs.susx.ac.uk

Jennifer E Jeffries
US Naval Observatory
Washington, DC, USA
jeffries@spica.usno.navy.mil

Judy Johnson
ETP Services
2906 NE Glisan Street
Portland, Oregon, USA

David M Jones
MIT Laboratory for CS
Cambridge, Massachusetts, USA
dmjones@theory.lcs.mit.edu

Michele Jouhet
CERN
Genève, Switzerland
jouhet@cernum.cern.ch

Joe Kaiping
Wolfram Research Inc
Champaign, Illinois, USA
kaiping@wri.com

Minato Kawaguti
Fukui University
Fukui, Japan
kawaguti@ilmps.fuis.fukui-u.ac.jp

Joachim Lammarsch
DANTE e.V.
Heidelberg, Germany
president@dante.de

Leslie Lamport
Digital Equipment Corp
Palo Alto, California, USA
lamport@src.aec.com

Dan Latterner
American Mathematical Society
Mathematical Reviews
Ann Arbor, Michigan, USA
dcl@math.ams.org

Maurice Laugier
Imprimerie Louis-Jean
GAP Cedex, France
louisean@cicg.grenet.fr

Jacques Legrand
JL International Publishing
Bassillac, France

Pierre A MacKay
University of Washington
Seattle, WA, USA
mackay@cs.washington.edu

Basil K Malyshev
malyshev@desert.ihep.su

Robert McGaffey
Martin Marietta Energy Systems
Oak Ridge, Tennessee, USA
rwm@ornl.gov

Wendy McKay
Mathematical Sciences Research Institute
Berkeley, California, USA
mckwendy@msri.org

Lothar Meyer-Lerbs
Bremen, Germany
TeXSatz@zfn.uni-breman.de

Frank Mittelbach
Mainz Bretzenheim, Germany
mittelbach@mzdmza.zdv.uni-mainz.de

Patricia A Monohon
TeX Users Group
Santa Barbara, California, USA
monohon@tug.org

André Montpetit
Université de Montreal
Montréal, Québec, Canada
montpeti@crm.umontreal.ca

Norman Naugle
Texas A&M University
College Station, Texas, USA
norman@math.tamu.edu

Darlene O'Rourke
San Francisco, California, USA
loyola@crl.com

John O'Rourke
San Francisco, California, USA
loyola@crl.com

Arthur Ogawa
TeX Consultants
Kaweah, California, USA
ogawa@teleport.com

Oren Patashnik
Institute for Defense Analyses
Center for Communications Research
San Diego, California, USA
opbibtex@cs.stanford.edu

Geraldine Pecht
Princeton University
Princeton, New Jersey, USA
gerree@math.princeton.edu

John Plaice
Université Laval
Ste-Foy, Québec, Canada
plaice@ift.ulaval.ca

Cheryl Ponchin
Institute for Defense Analyses
Princeton, New Jersey, USA
cheryl@ccr-p.ida.org

Richard Quint
Ventura College
Ventura, California, USA

Sebastian Rahtz
ArchaeoInformatica
York, England
spqr@ftp.tex.ac.uk

David F Richards
University of Illinois
Urbana, Illinois, USA
dfr@uiui.edu

Tom Rokicki
Stanford, California, USA
rokicki@cs.stanford.edu

Chris Rowley
Open University
London, England
carowley@open.ae.uk

Andrea Salati
Pubblicita Italia SAS
Modena, Italy
andrea.salati@galactica.it

David Salomon
California State University
Northridge, California, USA

Volker RW Schaa
Gesellschaft für Schwerionfor.
Nat. Lab
Muhltal, Germany
v.r.w.schaa@gsi.de

Joachim Schrod
Technische Universität Darmstadt
Darmstadt, Germany
schrod@iti.informatik.th-darmstadt.de

Maureen Schupsky
Annals of Mathematics
Princeton University
Princeton, New Jersey, USA
annals@math.princeton.edu

Caroline Skurat
Addison-Wesley Publishing Co
Reading, Massachusetts, USA
carolins@aw.com

Barry Smith
Blue Sky Research
Portland, Oregon, USA

Lowell Smith
Salt Lake City, Utah, USA

Michael Sofka
Publication Services Inc
Albert, New York, USA
mike@psarc.com

Friedhelm Sowa
Heinrich-Heine-University
Düsseldorf, Germany
tex@mail.rz.uni-duesseldorf.de

Wiesław Stajszczyk
ECRC GmbH
München, Germany
smitey@ecrc.de

Dave Steiner
Rutgers University
Piscataway, New Jersey, USA
steiner@cs.rutgers.edu

Jon Stenerson
TCI Software Research
Las Cruces, New Mexico, USA
jon_stenerson@tcisoft.com

Janet Sullivan
TEX Users Group
Santa Barbara, California, USA
janet@tug.org

Philip Taylor
RHBNC / University of London
Egham, Surrey, England
p.taylor@vax.rhbnc.ac.uk

Christina Thiele
Carleton Production Centre
Nepean, Ontario, Canada
cthiele@ccs.carleton.ca

Lee F Thompson
University of Wisconsin
Madison, Wisconsin, USA

Andy Trinh
Beckman Instruments
Brea, California, USA
axtrinh@biivax.dp.backman.com

Norman Walsh
O'Reilly and Asociates, Inc
Cambridge, Massachusetts, USA
norm@ora.com

Tao Wang
Personal TEX Inc
Mill Valley, California, USA
pti@crl.com

Alan Wetmore
US Army Research Lab
White Sands Missile Range,
New Mexico, USA
awetmore@arl.mil

Bill White
TSI Graphics
Effingham, Illinois, USA

Yusof Yaacob
Universiti Teknologi Malaysia
Johor Darul Ta'zim, Malaysia

Ralph Youngen
American Mathematical Society
Providence, Rhode Island, USA
rey@math.ams.org

Jíři Zlatuška
Masaryk University
Brno, Czech Republic
zlatuska@muni.cz

Borut Žnidar
Josef Stefan Institute
Kranj, Slovenia
borut.znidar@ijs.si

Darko Zupanić
Josef Stefan Institute
Kranj, Slovenia
darko.zupanic@ijs.si

Calendar

1994

Sep 26 – 30 EuroTEX '94, Sobieszewo, Poland. For information, contact Włodek Bzyl (`EuroTeX@Halina.Univ.Gda.Pl`). (See announcement, *TUGboat* 15, no. 1, p. 69.)

Oct 6 DANTE TEX–Stammtisch at the Universität Bremen, Germany. For information, contact Martin Schröder (`MS@Dream.HB.North.de`; telephone 0421/628813). First Thursday, 18:30, Universität Bremen MZH, 4th floor, across from the elevator.

Oct 13 DANTE TEX–Stammtisch, Wuppertal, Germany. For information, contact Andreas Schrell (`Andreas.Schrell@FernUni-Hagen.de`, telephone (0202) 666889). Second Thursday, 19:30, Gaststätte Yol, Ernststraße 43.

Oct 17 DANTE TEX–Stammtisch in Bonn, Germany. For information, contact Herbert Framke (`Herbert_Framke@SU2.MAUS.DE`; telephone 02241 400018, Mailbox 02241 390082). Third Monday, Anno, Kölnstraße 47.

Oct 18 DANTE TEX–Stammtisch in Duisburg, Germany. For information, contact Friedhelm Sowa (`tex@ze8.rz.uni-duesseldorf.de`; telephone 0211/311 3913). Third Tuesday, 19:30, at Gatz an der Kö, Königstraße 67.

Oct 19 UK TEX Users' Group, University of Warwick. Annual General Meeting. For information, e-mail `uktug-enquiries@ftp.tex.ac.uk`

Oct 26 DANTE TEX–Stammtisch, Hamburg, Germany. For information, contact Reinhard Zierke (`zierke@informatik.uni-hamburg.de`; telephone (040) 54715-295). Last Wednesday, 18:00, at TEX's Bar-B-Q, Grindelallee 31.

Nov 3 DANTE TEX–Stammtisch at the Universität Bremen, Germany. (For contact information, see Oct 6.)

Nov 10 DANTE TEX–Stammtisch, Wuppertal, Germany. (For contact information, see Oct 13.)

Nov 15 DANTE TEX–Stammtisch in Duisburg, Germany. (For contact information, see Oct 18.)

Nov 17 – 18 NTG 14th Meeting, and short course on LATEX 2.09 → LATEX2ε, University of Antwerpen, Belgium. For information, contact Gerard van Nes (`vannes@ecn.nl`).

Nov 21 DANTE TEX–Stammtisch in Bonn, Germany. (For contact information, see Oct 17.)

Nov 30 DANTE TEX–Stammtisch, Hamburg, Germany. (For contact information, see Oct 26.)

Dec 1 DANTE TEX–Stammtisch at the Universität Bremen, Germany. (For contact information, see Oct 6.)

Dec 8 DANTE TEX–Stammtisch, Wuppertal, Germany. (For contact information, see Oct 13.)

Dec 19 DANTE TEX–Stammtisch in Bonn, Germany. (For contact information, see Oct 17.)

Dec 20 DANTE TEX–Stammtisch in Duisburg, Germany. (For contact information, see Oct 18.)

Dec 28 DANTE TEX–Stammtisch, Hamburg, Germany. (For contact information, see Oct 26.)

1995

Jan 5 – 8 Linguistic Society of America, 69th Annual Meeting, Fairmont Hotel, New Orleans. For information, contact the LSA office, Washington, DC (202-834-1714, `zzlsa@gallua.gallaudet.edu`).

Jan 12 DANTE TEX–Stammtisch at the Universität Bremen, Germany. (For contact information, see Oct 6.)

Status as of 15 October 1994

Jan 19 Journée d'information sur la Diffusion des Documents Electroniques de LaTeX à HTML, WWW, et Acrobat, Nanterre, France. For information, e-mail tresorerie.gutenberg@ens.fr.

Jan 19 Portable Documents: Acrobat, SGML & TeX, Joint meeting of the UK TeX Users' Group and BCS Electronic Publishing Specialist Group, The Bridewell Theatre, London, UK. For information, contact Malcolm Clark (m.clark@warwick.ac.uk).

TUG Courses, Santa Barbara, California

Jan 30 – Feb 3 Intensive LaTeX2$_\varepsilon$

Feb 6–10 Beginning/Intermediate TeX

Feb 13–17 Advanced TeX and Macro Writing

Feb 28 – Mar 3 TeX-Tagung DANTE '95, University of Gießen, Germany. For information, contact Günter Partosch ((0641) 702 2170, dante95@hrz.uni-giessen.de).

Apr UK TeX Users' Group, location to be announced. Topic: Maths is what TeX does best of all. For information, e-mail uktug-enquiries@ftp.tex.ac.uk

Jun 1–2 GUTenberg '95, "Graphique, TeX et PostScript", La Grande Motte, France. For information, call (33-1) 30-87-06-25, or e-mail treasorerie.gutenberg@ens.fr or aro@lirmm.fr.

Jun 1–2 IWHD '95: International Workshop on Hypermedia Design, Montpellier, France. For information, contact the conference secretariat, Corine Zicler, LIRMM, Montpellier ((33) 6741 8503, zicler@lirmm.fr).

Jul 24–28 **TUG 16th Annual Meeting**: Real World TeX, St. Petersburg Beach, Florida. For information, send e-mail to tug95@scri.fsu.edu. (For a preliminary announcement, see *TUGboat* 15, no. 2, p. 160.)

For additional information on the events listed above, contact the TUG office (805-963-1338, fax: 805-963-8358, e-mail: tug@tug.org) unless otherwise noted.

Announcements

The Donald E. Knuth Scholarship: 1994 Scholar and 1995 Announcement

At the 15th Annual Meeting of TUG, Shelley-Lee Ames was honored as the 1994 Donald E. Knuth Scholarship winner. Shelley works at the University of Manitoba for the Canadian Mathematical Society (Société mathématique du Canada) where she prepares, formats and proofs all papers published by the society in their Journal and Bulletin.

Announcement of the 1995 competition

One Knuth Scholarship will be available for award in 1995. The competition will be open to all TeX users holding support positions that are secretarial, clerical or editorial in nature. It is therefore not intended for those with a substantial training in technical, scientific or mathematical subjects and, in particular, it is not open to anyone holding, or studying for, a degree with a major or concentration in these areas.

The award will consist of an expense-paid trip to the 1995 TUG Annual Meeting at St. Petersburg, Florida, and to the Scholar's choice from the short courses offered in conjunction with that meeting; and TUG membership for 1995, if the Scholar is not a TUG member, or for 1996, if the Scholar is already a TUG member. A cap of $2000 has been set for the award; however, this does not include the meeting or course registration fee, which will be waived.

To enter the competition, applicants should submit to the Scholarship Committee, by the deadline specified below, the input file and final TeX output of a project that displays originality, knowledge of TeX, and good TeXnique.

The project as submitted should be compact in size. If it involves a large document or a large number of documents then only a representative part should be submitted, together with a description of

the whole project. For example, from a book just one or two chapters would be appropriate.

The project may make use of a macro package, either a public one such as LaTeX or one that has been developed locally; such a macro package should be identified clearly. Such features as sophisticated use of math mode, of macros that require more than "filling in the blanks", or creation and use of new macros will be taken as illustrations of the applicant's knowledge.

All macros created by the candidate should be well documented with clear descriptions of how they should be used and an indication of how they work internally.

All associated style files, macro-package files, etc., should be supplied, or a clear indication given of any widely available ones used (including version numbers, dates, etc.); clear information should be provided concerning the version of TeX used and about any other software (e.g. particular printer drivers) required. Any nonstandard fonts should be identified and provided in the form of `.tfm` and `.pk` files suitable for use on a 300dpi laser printer.

While the quality of the typographic design will not be an important criterion of the judges, candidates are advised to ensure that their printed output adheres to sound typographic standards; the reasons for any unusual typographic features should be clearly explained.

All files and documents comprising the project must be submitted on paper; the input files should be provided in electronic form as well. Suitable electronic media are IBM PC-compatible or Macintosh diskettes, or a file sent by electronic mail.

A brochure with additional information is available from the TUG office. To obtain a copy, or to request instructions on e-mail submission, write to the address at the end of this announcement, or send a message by e-mail to `tug@tug.org` with the subject "Knuth Scholarship request".

Along with the project, each applicant should submit a letter stating the following:

1. affirmation that he/she will be available to attend the 1995 TUG Annual Meeting;
2. affirmation of willingness to participate on the committee to select the next Scholar.

Each applicant should also submit a *curriculum vitae* summarizing relevant personal information, including:

1. statement of job title, with a brief description of duties and responsibilities;
2. description of general post-secondary school education, TeX education, identifying courses attended, manuals studied, personal instruction from experienced TeX users, etc.;
3. description of TeX resources and support used by the candidate in the preparation of the project.

Neither the project nor the *curriculum vitae* should contain the applicant's name or identify the applicant. These materials will be reviewed by the committee without knowledge of applicants' identities. If, despite these precautions, a candidate is identifiable to any judge, then that judge will be required to make this fact known to the others and to the TUG board members responsible for the conduct of the judging.

The covering letter, *curriculum vitae*, and all macro documentation that is part of the project input should be in English. (English is not required for the output of the project.) However, if English is not the applicant's native language, that will not influence the decision of the committee.

Selection of the Scholarship recipient will be based on the project submitted.

Schedule

The following schedule will apply (all dates are in 1995):

7 April	Deadline for receipt of submissions
21 April–2 June	Judging period
9 June	Notification of winner
24–28 July	1995 Annual Meeting, St. Petersburg, Florida

The 1995 Scholarship Committee consists of

- Chris Rowley, Open University, UK (Chair);
- David Salomon, California State University, Northridge, USA;
- Shelly-Lee Ames, University of Manitoba, Canada.

Where to write

All applications should be submitted to the Committee in care of the TUG office:

TeX Users Group
Attn: Knuth Scholarship Competition
P. O. Box 869
Santa Barbara, CA 93102 USA
e-mail: `tug@tug.org`

Nico Poppelier
Liaison to the Donald E. Knuth Scholarship Committee

Institutional Members

The Aerospace Corporation,
El Segundo, California

Air Force Institute of Technology,
Wright-Patterson AFB, Ohio

American Mathematical Society,
Providence, Rhode Island

ArborText, Inc.,
Ann Arbor, Michigan

Brookhaven National Laboratory,
Upton, New York

Brown University,
Providence, Rhode Island

California Institute of Technology,
Pasadena, California

Carleton University,
Ottawa, Ontario, Canada

Centre Inter-Régional de
Calcul Électronique, CNRS,
Orsay, France

CERN, *Geneva, Switzerland*

College Militaire Royal de Saint
Jean, *St. Jean, Quebec, Canada*

College of William & Mary,
Department of Computer Science,
Williamsburg, Virginia

Communications
Security Establishment,
Department of National Defence,
Ottawa, Ontario, Canada

Cornell University,
Mathematics Department,
Ithaca, New York

CSTUG, *Praha, Czech Republic*

Elsevier Science Publishers B.V.,
Amsterdam, The Netherlands

Escuela Superior de
Ingenieres Industriales,
Sevilla, Spain

European Southern Observatory,
Garching bei München, Germany

Fermi National Accelerator
Laboratory, *Batavia, Illinois*

Florida State University,
Supercomputer Computations
Research, *Tallahassee, Florida*

GKSS, Forschungszentrum
Geesthacht GmbH,
Geesthacht, Germany

Grinnell College,
Computer Services,
Grinnell, Iowa

Hong Kong University of
Science and Technology,
Department of Computer Science,
Hong Kong

Institute for Advanced Study,
Princeton, New Jersey

Institute for Defense Analyses,
Communications Research
Division, *Princeton, New Jersey*

Iowa State University,
Ames, Iowa

Los Alamos National Laboratory,
University of California,
Los Alamos, New Mexico

MacroSoft, *Warsaw, Poland*

Marquette University,
Department of Mathematics,
Statistics and Computer Science,
Milwaukee, Wisconsin

Mathematical Reviews,
American Mathematical Society,
Ann Arbor, Michigan

Max Planck Institut
für Mathematik,
Bonn, Germany

New York University,
Academic Computing Facility,
New York, New York

Nippon Telegraph &
Telephone Corporation,
Basic Research Laboratories,
Tokyo, Japan

Personal TEX, Incorporated,
Mill Valley, California

Princeton University,
Princeton, New Jersey

Rogaland University,
Stavanger, Norway

Rutgers University,
Computing Services,
Piscataway, New Jersey

Space Telescope Science Institute,
Baltimore, Maryland

Springer-Verlag,
Heidelberg, Germany

Springer-Verlag New York, Inc.,
New York, New York

Stanford Linear Accelerator
Center (SLAC),
Stanford, California

Stanford University,
Computer Science Department,
Stanford, California

Texas A & M University,
Department of Computer Science,
College Station, Texas

United States Naval
Postgraduate School,
Monterey, California

Universität Augsburg,
Augsburg, Germany

University of California, Berkeley,
Space Astrophysics Group,
Berkeley, California

University of California, Irvine,
Information & Computer Science,
Irvine, California

University of Canterbury,
Christchurch, New Zealand

University College,
Cork, Ireland

University of Delaware,
Newark, Delaware

University of Groningen,
Groningen, The Netherlands

University of Heidelberg,
Computing Center,
Heidelberg, Germany

University of Illinois at Chicago,
Computer Center,
Chicago, Illinois

Universität Koblenz–Landau,
Koblenz, Germany

University of Manitoba,
Winnipeg, Manitoba

University of Oslo,
Institute of Informatics,
Blindern, Oslo, Norway

University of Salford,
Salford, England

University of South Carolina,
Department of Mathematics,
Columbia, South Carolina

University of Southern California,
Information Sciences Institute,
Marina del Rey, California

University of Stockholm,
Department of Mathematics,
Stockholm, Sweden

University of Texas at Austin,
Austin, Texas

Università degli Studi di Trento,
Trento, Italy

Uppsala University,
Uppsala, Sweden

Villanova University,
Villanova, Pennsylvania

Vrije Universiteit,
Amsterdam, The Netherlands

Wolters Kluwer,
Dordrecht, The Netherlands

Yale University,
Department of Computer Science,
New Haven, Connecticut

TeX Consulting & Production Services

North America

Abrahams, Paul
214 River Road, Deerfield, MA 01342; (413) 774-5500
Composition and typesetting of high-quality books and technical documents. Complete production services using any PostScript fonts. Assistance with book design and copy. I am a computer consultant with a computer science education.

American Mathematical Society
P. O. Box 6248, Providence, RI 02940; (401) 455-4060
Typesetting from DVI files on an Autologic APS Micro-5 or an Agfa Compugraphic 9600 (PostScript). Times Roman and Computer Modern fonts. Composition services for mathematical and technical books and journal production.

Anagnostopoulos, Paul C.
433 Rutland Street, Carlisle, MA 01741; (508) 371-2316
Composition and typesetting of high-quality books and technical documents. Production using Computer Modern or any available PostScript fonts. Assistance with book design. I am a computer consultant with a Computer Science education.

ArborText, Inc.
1000 Victors Way, Suite 400, Ann Arbor, MI 48108; (313) 996-3566
TeX installation and applications support. TeX-related software products.

Archetype Publishing, Inc., Lori McWilliam Pickert
P. O. Box 6567, Champaign, IL 61821; (217) 359-8178
Experienced in producing and editing technical journals with TeX; complete book production from manuscript to camera-ready copy; TeX macro writing including complete macro packages; consulting.

The Bartlett Press, Inc., Frederick H. Bartlett
Harrison Towers, 6F, 575 Easton Avenue, Somerset, NJ 08873; (201) 745-9412
Vast experience: 100+ macro packages, over 30,000 pages published with our macros; over a decade's experience in all facets of publishing, both TeX and non-TeX; all services from copyediting and design to final mechanicals.

Cowan, Dr. Ray F.
141 Del Medio Ave. #134, Mountain View, CA 94040; (415) 949-4911
Ten Years of TeX and Related Software Consulting: Books, Documentation, Journals, and Newsletters
TeX & LaTeX macropackages, graphics; PostScript language applications; device drivers; fonts; systems.

Hoenig, Alan
17 Bay Avenue, Huntington, NY 11743; (516) 385-0736
TeX typesetting services including complete book production; macro writing; individual and group TeX instruction.

Magus, Kevin W. Thompson
P. O. Box 390965, Mountain View CA 94039-0965; (800) 848-8037; (415) 940-1109; `magus@cup.portal.com`
LaTeX consulting from start to finish. Layout design and implementation, macro writing, training, phone support, and publishing. Can take LaTeX files and return camera ready copy. Knowledgeable about long document preparation and mathematical formatting.

NAR Associates
817 Holly Drive E. Rt. 10, Annapolis, MD 21401; (410) 757-5724
Extensive long term experience in TeX book publishing with major publishers, working with authors or publishers to turn electronic copy into attractive books. We offer complete free lance production services, including design, copy editing, art sizing and layout, typesetting and repro production. We specialize in engineering, science, computers, computer graphics, aviation and medicine.

Ogawa, Arthur
920 Addison, Palo Alto, CA 94301; (415) 323-9624
Experienced in book production, macro packages, programming, and consultation. Complete book production from computer-readable copy to camera-ready copy.

Pronk&Associates Inc.
1129 Leslie Street, Don Mills, Ontario, Canada M3C 2K5; (416) 441-3760; Fax: (416) 441-9991
Complete design and production service. One, two and four-color

books. Combine text, art and photography, then output directly to imposed film. Servicing the publishing community for ten years.

Quixote Digital Typography, Don Hosek
555 Guilford, Claremont, CA 91711; (909) 621-1291; Fax: (909) 625-1342
Complete line of TeX, LaTeX, and METAFONT services including custom LaTeX style files, complete book production from manuscript to camera-ready copy; custom font and logo design; installation of customized TeX environments; phone consulting service; database applications and more. Call for a free estimate.

Richert, Norman
1614 Loch Lake Drive, El Lago, TX 77586; (713) 326-2583
TeX macro consulting.

TeXnology, Inc., Amy Hendrickson
57 Longwood Ave., Brookline, MA 02146; (617) 738-8029
TeX macro writing (author of MacroTeX); custom macros to meet publisher's or designer's specifications; instruction.

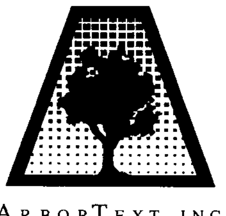
Type 2000
16 Madrona Avenue, Mill Valley, CA 94941; (415) 388-8873; Fax: (415) 388-8865
$2.50 per page for 2000 DPI TeX camera ready output! We have a three year history of providing high quality and fast turnaround to dozens of publishers, journals, authors and consultants who use TeX. Computer Modern, Bitstream and METAFONT fonts available. We accept DVI files only and output on RC paper. $2.25 per page for 100+ pages, $2.00 per page for 500+ pages.

Outside North America

TypoTeX Ltd.
Electronical Publishing, Battyány u. 14. Budapest, Hungary H-1015; (036) 11152 337
Editing and typesetting technical journals and books with TeX from manuscript to camera ready copy. Macro writing, font designing, TeX consulting and teaching.

Information about these services can be obtained from:

TeX Users Group
P. O. Box 869
Santa Barbara, CA 93102-0869
Phone: (805) 963-1388
Fax: (805) 963-8538
Email: tug@tug.org

NEW!  NEW!

- Silicon Graphics Iris or Indigo
- Solaris 2.1
- DVILASER/HP3
- Motif and OPEN LOOK Preview

Complete TEX packages
Ready to use, fully documented and supported.

Also Available For: Sun-4 (SPARC), IBM RS/6000, DEC/RISC-Ultrix, HP 9000, and IBM PC's

Call us for more information on our exciting new products!

1000 Victors Way ▲ Suite 400 ▲ Ann Arbor, MI 48108 ▲ (313) 996–3566 ▲ FAX (313) 996–3573

YOUR ONE STOP SOURCE FOR TEX MATERIALS

Includes LaTEX2ε.
We have a complete selection of books on TEX and document publishing, all at 10% savings.
A MicroTEX full system is $350.00 - Ask for details to upgrade your system.

TEX TOOL BOX

AmSpell Checker (FREE when ordered with MicroTEX) reg. $10
TEXHelp $49.95 DEMACS Editor (FREE when ordered with MicroTEX) reg. $10 Voyager $25.00
Adobe Type-on-Call $99.00 AMS-TEX $50.00 TEXpic $79.00 Capture Graphics $75.00

Micro Programs Inc., 251 Jackson Ave., Syosset, NY 11791 (516) 921-1351

Complete and return this form with payment to:

TEX Users Group
Membership Department
P. O. Box 869
Santa Barbara, CA 93102 USA

Telephone: (805) 963-1338
FAX: (805) 963-8358
Email: `tug@tug.org`

Membership is effective from January 1 to December 31 and includes subscriptions to *TUGboat, The Communications of the TEX Users Group* and the TUG newsletter, *TEX and TUG NEWS.* Members who join after January 1 will receive all issues published that calendar year.

For more information ...

Whether or not you join TUG now, feel free to return this form to request more information. Be sure to include your name and address in the spaces provided to the right.

Check all items you wish to receive below:

- ☐ Institutional membership information
- ☐ Course and meeting information
- ☐ Advertising rates
- ☐ Products/publications catalogue
- ☐ Public domain software catalogue

Individual Membership Application

Name ______________________________

Institutional affiliation, if any ______________________________

Position ______________________________

Address (business or home (circle one)) ______________________________

____

____

City ____________________ Province/State ____________

Country ____________________ Postal Code ____________

Telephone ____________________ FAX ____________

Email address ______________________________

I am also a member of the following other TEX organizations:

Specific applications or reasons for interest in TEX:

There are two types of TUG members: regular members, who pay annual dues of $60; and full-time student members, whose annual dues are $30. Students must include verification of student status with their applications.

Please indicate the type of membership for which you are applying:

☐ Regular at $60 ☐ Full-time student at $30

Amount enclosed for 1994 membership: $ __________

☐ Check/money order payable to TEX Users Group enclosed
(*checks in US dollars must be drawn on a US bank; checks in other currencies are acceptable, drawn on an appropriate bank*)

☐ Bank transfer:

TEX Users Group, Account #1558-816,
Santa Barbara Bank and Trust, 20 East Carrillo Street,
Santa Barbara, CA 93101 USA

your bank ______________________________

ref # ______________________________

☐ Charge to MasterCard/VISA

Card # ____________________ Exp. date ________

Signature ______________________________

Institutional Membership Application

Institution or Organization ______________________________

____

Principal contact ______________________________

Address ______________________________

____

City ______________ Province/State ______________

Country ______________ Postal Code ______________

Daytime telephone ______________ FAX ______________

Email address ______________________________

Complete and return this form with payment to:

TEX Users Group
Membership Department
P. O. Box 869
Santa Barbara, CA 93102
USA

Membership is effective from January 1 to December 31. Members who join after January 1 will receive all issues of *TUGboat* and *TEX and TUG NEWS* published that calendar year.

Each Institutional Membership entitles the institution to:

- designate a number of individuals to have full status as TUG individual members;
- take advantage of reduced rates for TUG meetings and courses for *all* staff members;
- be acknowledged in every issue of *TUGboat* published during the membership year.

Educational institutions receive a $100 discount in the membership fee. The three basic categories of Institutional Membership each include a certain number of individual memberships. Additional individual memberships may be obtained at the rates indicated. Fees are as follows:

Category	Rate (educ. / non-educ.)	Add'l mem.
A (includes 7 memberships)	$ 540 / $ 640	$50 ea.
B (includes 12 memberships)	$ 815 / $ 915	$50 ea.
C (includes 30 memberships)	$1710 / $1810	$40 ea.

Please indicate the type of membership for which you are applying:

Category _____ + _____ additional individual memberships

Amount enclosed for 1994 membership: $ ________

☐ Check/money order payable to TEX Users Group enclosed (*payment in US dollars must be drawn on a US bank; payment in other currencies is acceptable, drawn on an appropriate bank*)

☐ Bank transfer: your bank ______________________

ref # ______________________

TEX Users Group, Account #1558-816,
Santa Barbara Bank and Trust, 20 East Carrillo Street,
Santa Barbara, CA 93101 USA

☐ Charge to MasterCard/VISA

Card # ______________________ Exp. date ________

Signature ______________________

Please attach a list of individuals whom you wish to designate as TUG individual members. Minimally, we require names and addresses so that TUG publications may be sent directly to these individuals, but we would also appreciate receiving the supplemental information regarding phone numbers, email addresses, and TEX interests as requested on the TUG Individual Membership Application form. For this purpose, the latter application form may be photocopied and mailed with this form.

For more information ...

Correspondence

TEX Users Group
P. O. Box 869
Santa Barbara, CA 93102
USA

Telephone: (805) 963-1338
FAX: (805) 963-8358
Email: `tug@tug.org`

Whether or not you join TUG now, feel free to return this form to request more information.

Check all items you wish to receive below:

☐ Course and meeting information

☐ Advertising rates

☐ Products/publications catalogue

☐ Public domain software catalogue

e-MATH: Now on World Wide Web

Your Internet connection to the mathematical community

WWW access:
http://e-math.ams.org/

Telnet access:
telnet e-math.ams.org
Login and password are both e-math (lowercase)

For more information contact: eps@math.ams.org

Electronic Products
American Mathematical Society

Index of Advertisers

419	Addison-Wesley
417	American Mathematical Society
414	ArborText
Cover 3	Blue Sky Research
414	Micro Programs, Inc.
417	Springer-Verlag
418	Y&Y

DVIWindo [newtugad.dvi] page: 1

File Preferences Previous! Next! Unmagnify! Magnify! Fonts

Bitmap-free TEX for Windows

Powerful, fast, flexible TEX system for Windows

TeX Package

Y&Y TEX Package includes:

- DVIWindo
- DVIPSONE
- Y&Y big TEX
- Adobe Type Manager
- Acrobat Reader
- PostScript Type 1 fonts

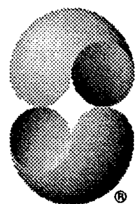
Unique Features

- Uses Type 1 *or* TrueType fonts
- Provides *partial font downloading*
- Can use *any* Windows printer driver
- Big TEX runs in Windows *or* DOS
- Commercial grade, fully hinted fonts
- *Complete* flexibility in font encoding
- Support for EPS *and* TIFF images

Why Y&Y?

Mature products. Years of experience with Windows, PostScript printers and scalable outline fonts. We *understand* and know how to *avoid* problems with Windows, ATM, 'clone' printers, and problem fonts.

Y&Y — the experts in scalable outline fonts for TEX



Y&Y, Inc. 45 Walden St., Suite 2F, Concord, MA 01742 USA — (800) 742-4059 — (508) 371-3286 — (508) 371-2004 (fax)

DVIWindo and DVIPSONE are trademarks of Y&Y, Inc. Windows is a registered trademark of MicroSoft Co. Adobe Type Manager is a registered trademark of Adobe Systems Inc.

LaTeX2ε . . . Knuth. . . *Mathematica*®

Leading the way in scientific computing. Addison-Wesley.

When looking for the best in scientific computing, you've come to rely on Addison-Wesley. Take a moment to see how we've made our list even stronger.

The LaTeX Companion
Michael Goossens, Frank Mittelbach, and Alexander Samarin
This book is packed with information needed to use LaTeX even more productively. It is a true companion to Leslie Lamport's users guide as well as a valuable complement to any LaTeX introduction. Describes the new LaTeX standard.
1994 (0-201-54199-8) 400 pp. Softcover

LaTeX: A Document Preparation System, Second Edition
Leslie Lamport
The authoritative user's guide and reference manual has been revised to document features now available in the new standard software release—LaTeX2ε. The new edition features additional styles and functions, improved font handling, and much more.
1994 (0-201-52983-1) 256 pp. Softcover

The Stanford GraphBase: A Platform for Combinatorial Computing
Donald E. Knuth
This book represents the first fruits of Knuth's preparation for Volume 4 of *The Art of Computer Programming.* It uses examples to demonstrate the art of literate programming, and provides a useful means for comparing combinatorial algorithms.
1994 (0-201-54275-7) 600 pp. Hardcover

The CWEB System of Structured Documentation (Version 3.0)
Donald E. Knuth and Silvio Levy
CWEB is a version of WEB for documenting C and C++ programs. CWEB combines TEX with two of today's most widely used professional programming languages. This book is the definitive user's guide and reference manual for the CWEB system.
1994 (0-201-57569-8) approx. 240 pp. Softcover

Concrete Mathematics, Second Edition
Ronald L. Graham, Donald E. Knuth, and Oren Patashnick
With improvements to almost every page, the second edition of this classic text and reference introduces the mathematics that supports advanced computer programming.
1994 (0-201-55802-5) 672 pp. Hardcover

Applied *Mathematica*®: Getting Started, Getting It Done
William T. Shaw and Jason Tigg
This book shows how *Mathematica*® can be used to solve problems in the applied sciences. Provides a wealth of practical tips and techniques.
1994 (0-201-54217-X) 320 pp. Softcover

The Joy of *Mathematica*®
Alan Shuchat and Fred Shultz
This software product provides easy-to-use menus for Macintosh versions of *Mathematica*®. Its accompanying book is an exploration of key issues and applications in Mathematics.
1994 (0-201-59145-6) 200 pp. Softcover + disk

Look for these titles wherever fine technical books are sold.

Addison-Wesley Publishing Company
1-800-447-2226

TUG '95

— *St. Petersburg Beach, Florida* —

July 24–28, 1995

The TeX Users Group is proud to announce that the **sixteenth** annual meeting will be held at the TradeWinds Hotel, in St. Petersburg Beach, Florida, July 24–28, 1995. We would like to extend a warm invitation to TeX users around the world—come join us at one of the largest and most beautiful resort beaches in Florida, as we explore where TeX is to be found and how its users are going far beyond—or are diverging from—its initial mathematical context.

The theme of the meeting will be "Real World TeX" and we plan to have demonstrations of pre- and post-processors, and the active participation of developers and vendors, in hopes that **you**, the user, may discover "hands-on" just what can be done with TeX, METAFONT, PostScript, and other utilities!

Commercial users of TeX are particularly encouraged to attend. The meeting will feature papers of interest to publishers and TeX vendors, a panel discussion addressing commercial users' needs and wants, and a gallery for displaying samples of TeX work.

There will be the usual courses associated with the meeting: *Intensive Courses* in LaTeX2ε and TeX, PostScript, Graphics, and perhaps other topics. The meeting itself will have excellent speakers, panel discussions, workshops, poster displays, Birds-of-a-Feather sessions (BoFs) and technical demonstrations.

Getting Information

A preliminary schedule will be available in February of 1995, so be sure to look for updates in *TeX and TUG NEWS* and *TUGboat*, on the World Wide Web, at `http://www.ucc.ie/info/TeX/tug/tug95sched.html`, as well as on the CTAN archives in `tex-archive/usergrps/tug/`.

Nearer the time of the conference, there will be an on-line form for registration (`tug95form.html`) located on the WWW cited above.

Send suggestions and requests to `tug95c@scri.fsu.edu`.

The TUG95 committee will be working with individuals who wish to share accommodations, to help defray expenses. The Bursary Fund is also available to assist TeX users who demonstrate need. All members are encouraged to consider contributing to the fund. To obtain more information about contributing to or applying for the Bursary Fund, please contact the TUG office by email to `tug@tug.org` or by post to the address
TeX Users Group
P.O. Box 869
Santa Barbara, CA 93102-0869 USA.

Deadlines

Submission of Abstracts	◇	January 31, 1995
Preliminary Papers Due	◇	April 30, 1995
Preprint Deadline	◇	June 23, 1995
Meeting Date	◇	July 24–July 28, 1995
Camera Ready Deadline	◇	August 25, 1995

Interactive TeX ✓
WYSIWYG TeX ✓
User-friendly TeX ✓

Textures ✓ It's not like any other TeX system.[1]

When Apple introduced the Macintosh and its graphic interface, we embarked on a long-term project of research toward a TeX system "for the rest of us," a TeX system that would be humanely interactive, and visibly responsive; an integrated TeX *system*, designed to be radically easy for those new to TeX, and engineered to be the best for expert TeX users. The research continues; the product is Textures.

Textures is something of a Copernican revolution in TeX interface. The paradigm shifts from the usual TeX "input–process–output–repeat" mode, to a wider frame wherein the TeX language describes a dynamic document that is continuously, quietly "realized" as you write it, with no process steps whatsoever.

This change in perspective must be experienced to be fully grasped. As you would expect, Textures is good for beginners. You might be surprised to know that it's also good for experienced TeX users and especially good for TeX programmers.[2] It's not a "front-end" or an imitation, it's a full-scale live TeX processor that's actually easy to use.

There's much more to Textures than a new perspective on TeX, of course; too much to list here but we'll mention custom menus, Computer Modern PostScript fonts, illustrations, inter-application communication, automatic installation, genuine support, ...

We don't claim perfection; nor do we believe in exaggerated marketing, odd as this may seem; and we do understand our finite abilities to provide all that one might wish. But we also guarantee that you will be satisfied with Textures and with the service you receive from Blue Sky Research, or we will refund the (very finite) price you pay.

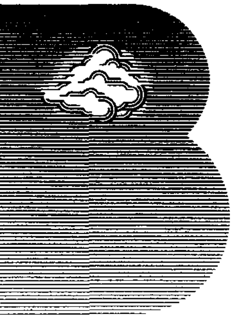
[1] On the other hand, Textures is *exactly* like every other TeX system. Its TeX engine is strictly standard and up-to-date; it runs LaTeX, AMS-TeX, and all standard TeX macros without change. But even here it's not ordinary, with hand-tuned assembler code for maximum performance, and a transparent memory model that you can't fill until you run out of disk.

[2] If you are a serious TeX user on another platform, it can be worth getting a Mac just to run Textures.

For all Macintosh processors and printers minimum 2.5MB memory and 5MB disk

Blue Sky Research
534 SW Third Avenue
Portland, Oregon 97204
USA

800 622 8398
503 222 9571
facsimile 503 222 1643
sales@bluesky.com



TUGboat

Volume 15, Number 3 / September 1994
1994 Annual Meeting Proceedings

166 Editorial and production notes
167 Acknowledgements and Conference Program: Innovation

Keynote
169 Charles Bigelow / *Lucida and TeX: lessons of logic and history*

Publishing, Languages Literature and Fonts
170 Frank Mittelbach and Michel Goossens / *Real life book production — lessons learned from The LaTeX Companion*
174 Yannis Haralambous / *Typesetting the holy Bible in Hebrew, with TeX*
192 Michel Cohen / *Adaptive character generation and spatial expressiveness*
199 Yannis Haralambous / *Humanist*
200 Basil Malyshev / *Automatic conversion of* METAFONT *fonts to Type1 PostScript*

Colour, and LaTeX
201 James Hafner / *The (pre)history of color in Rokicki's* `dvips`
205 Tom Rokicki / *Advanced 'special' support in a dvi driver*
213 Angus Duggan / *Colour separation and PostScript*
218 Sebastian Rahtz and Michel Goossens / *Simple colour design with LaTeX2ε*
223 Friedhelm Sowa / *Printing colour pictures*
228 Michael Sofka / *Color book production using TeX*
239 Timothy van Zandt and Denis Girou / *Inside PSTricks*
247 Jon Stenerson / *A LaTeX style file generator*
255 Johannes Braams / *Document classes and packages in LaTeX2ε*
263 Alan Jeffrey / *PostScript font support in LaTeX2ε*

TeX Tools
269 Oren Patashnik / BibTeX *1.0*
274 Pierre MacKay / *A typesetter's toolkit*
285 Michael P. Barnett and Kevin R. Perry / *Symbolic computation for electronic publishing*
293 Minato Kawaguti and Norio Kitajima / *Concurrent use of interactive TeX previewer with an Emacs-type Editor*
301 Yannis Haralambous / *An Indic TeX preprocessor — Sinhalese TeX*
302 Jean-Luc Doumont / *Pascal pretty-printing: an example of "preprocessing within TeX"*

Futures
309 Joachim Schrod / *Towards interactivity for TeX*
318 Chris Rowley and Frank Mittelbach / *The floating world*
319 Don Hosek / *Sophisticated page layout with TeX*
320 John Plaice / *Progress in the Omega project*
325 Arthur Ogawa / *Object-oriented programming, descriptive markup, and TeX*
331 William Erik Baxter / *An object-oriented programming system in TeX*
339 Norm Walsh / *A World Wide Web interface to* CTAN
344 Yannis Haralambous and John Plaice / *First applications of Ω: Adobe Poetica, Arabic, Greek, Khmer*
353 Philip Taylor / *ε-TeX & NTS: A progress report*

Publishing and Design
359 Maurice Laugier and Yannis Haralambous / *TeX innovations by the Louis-Jean Printing House*
360 Michael Downes / *Design by template in a production macro package*
369 Alan Hoenig / *Less is More: Restricting TeX's scope enables complex page layouts*
381 Jonathan Fine / *Documents, compuscripts, programs and macros*
386 Marko Grobelnik, Dunja Mladeniç, Darko Zupaniç and Borut Žnidar / *Integrated system for encyclopaedia typesetting based on TeX*
388 Henry Baragar and Gail E. Harris / *An example of a special purpose input language to LaTeX*

Appendix
397 Color pages
405 Participants at the Annual Meeting

News & Announcements
408 Calendar
409 1995 Knuth Scholarship

TUG Business
411 Institutional members
415 TUG membership application

Advertisements
412 TeX consulting and production services
417 Index of advertisers