

TUGBOAT

Volume 16, Number 3 / September 1995
1995 Annual Meeting Proceedings

	222	Robin Fairbairns / <i>Production notes</i>
Opening Address	223	Michel Goossens / <i>President's words</i>
Fonts	227	Jiří Zlatuška / <i>When METAFONT does it alone</i>
	233	Richard J. Kinch / <i>MetaFog: converting METAFONT shapes to contours</i>
	244	Alan Hoenig / <i>The Poetica family: fancy fonts with T_EX and L^AT_EX</i>
	253	Michel Goossens / <i>Using Adobe Type 1 Multiple Master fonts with T_EX</i>
	259	Jeremy Gibbons / <i>Dotted and dashed lines in METAFONT</i>
	265	Sergey Lesenko / <i>Printing T_EX documents with partial Type 1 fonts</i>
L^AT_EX	269	Matthew Swift / <i>Modularity in L^AT_EX</i>
	276	Dennis Kletzing / <i>A multienumerate package</i>
Hyphenation	280	Petr Sojka and Pavel Ševeček / <i>Hyphenation in T_EX — Quo Vadis?</i>
	290	Petr Sojka / <i>Notes on compound word hyphenation in T_EX</i>
Literate programming	297	Wlodek Bzyl / <i>Literate Plain source is available!</i>
	300	Bart Childs, Deborah Dunn and William Lively / <i>Teaching CS/1 courses in a literate manner</i>
Methods	310	T.V. Raman / <i>An audio view of (L^A)T_EX documents — part II</i>
	315	Sebastian Rahtz / <i>Another look at L^AT_EX to SGML conversion</i>
	325	Robin Fairbairns / <i>Omega — Why bother with Unicode?</i>
	329	Gabriel Valiente Feruglio / <i>Modern Catalan typographical conventions</i>
News & Announcements	339	TUG'96 Announcement
	340	Calendar
TUG Business	341	TUG'95 — List of Attendees
	344	Institutional members
Advertisements	345	T _E X consulting and production services

T_EX Users Group

Memberships and Subscriptions

TUGboat (ISSN 0896-3207) is published quarterly by the T_EX Users Group, Flood Building, 870 Market Street, #801; San Francisco, CA 94102, U.S.A.

1996 dues for individual members are as follows:

- Ordinary members: \$55
- Students: \$35

Membership in the T_EX Users Group is for the calendar year, and includes all issues of *TUGboat* 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: \$70 a year, including air mail delivery.

Second-class postage paid at San Francisco, CA, and additional mailing offices. Postmaster: Send address changes to *TUGboat*, T_EX Users Group, 1850 Union Street, #1637, San Francisco, CA 94123, U.S.A.

Institutional Membership

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

TUGboat © Copyright 1995, T_EX 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 T_EX 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 T_EX-arcana*[†]

Michel Goossens, *President**

Judy Johnson*, *Vice President*

Mimi Jett*, *Treasurer*

Sebastian Rahtz*, *Secretary*

Barbara Beeton

Karl Berry

Mimi Burbank

Michael Ferguson

Peter Flynn

George Greenwade

Yannis Haralambous

Jon Radel

Tom Rokicki

Norm Walsh

Jíří Zlatuška

Raymond Goucher, *Founding Executive Director*[†]

Hermann Zapf, *Wizard of Fonts*[†]

*member of executive committee

[†]honorary

Addresses

All correspondence,
payments, etc.

T_EX Users Group
1850 Union Street, #1637
San Francisco,
CA 94123 USA

Parcel post,
delivery services:

T_EX Users Group
Flood Building
870 Market Street, #801
San Francisco,
CA 94102, USA

Telephone

+1 415 982-8449

Fax

+1 415 982-8559

Electronic Mail

(Internet)

General correspondence:

TUG@tug.org

Submissions to *TUGboat*:

TUGboat@AMS.org

T_EX is a trademark of the American Mathematical Society.

Production Notes

Robin Fairbairns

University of Cambridge Computer Laboratory
Pembroke St
Cambridge, CB2 3QG
UK
Email: rf@cl.cam.ac.uk
URL: <http://www.cl.cam.ac.uk/users/rf/>

The Papers

There were more papers presented at the conference than could be accommodated within the page count available to us. As a result, the production team has had to move Jonathan Fine's paper *New Perspectives on T_EX Macros* and Jerry Marsden's (*et al.*) paper *Introduction to FasT_EX to TUGboat 16(4)*. Since that issue is being produced in parallel with the present one, readers will not have a serious wait. A transcript of Prof. Knuth's question and answer session is being prepared, and will (subject to his approval) be published in some future issue of *TUGboat*.

Of his two papers in this issue, Petr Sojka only presented the second (*Notes on Compound Word Hyphenation in T_EX*) at the conference. Since that paper was awarded the prize for best paper (at Donald Knuth's recommendation), Michel Goossens suggested that the first paper (*Hyphenation in T_EX — Quo Vadis?*), which sets the scene for the second paper, should also be presented here¹.

Macros

For the whole of the period of editing these proceedings, I've been working in parallel on macros to use with L^AT_EX 2_ε for producing *TUGboat*. I hope to report the state of this work in a paper for a future issue of *TUGboat*, but there's still much to do before the work is complete. For one paper (Childs, Dunn and Lively) I developed a separate (small) L^AT_EX 2_ε package. This package (*variline*) hasn't yet been released to CTAN, for lack of various sorts of testing; the omission on my part proves to be fortuitous, since the package doesn't work with the December 1995 release of L^AT_EX 2_ε (the incompatibility is easily dealt with, but I shan't be able to do so until the new year).

All but three papers were submitted as L^AT_EX source; the others were submitted as plain T_EX

¹ It had appeared in the preprints, due to an organisational error on my part.

marked up with the usual macros. Of the three plain papers, I converted one (Bzyl's) to L^AT_EX, left one (Valiente's) as it was, and am no longer responsible for the remaining one (Fine's).

Fonts

Most of this issue has been set in Computer Modern (or DC, version 1.1) fonts — in Malyshev's BaKoMa PostScript Type 1 versions; the only exception is the paper by Goossens, Rahtz and myself on the use of Adobe Multiple Master fonts. My original aim was to use Adobe Minion Multiple Master (which that paper uses) as the default font family for the whole of the issue, but planned further work on the metrics, and work on using Minion with papers submitted in plain T_EX, was not completed in good enough time.

Three of the other papers (Gibbons, Hoenig and Zlatuška) used 'exotic' fonts of one sort or another, but both Gibbons and Hoenig's samples were supplied as encapsulated PostScript. (Gibbons' paper talks about a bug in 'some printers' firmware that affects one of his diagrams; that bug affects his printer and mine, but not the one used for production at SCRI.)

Zlatuška's paper imposes slight difficulties on the production system — it requires some reasonably modest use of METAFONT to produce the logo font that he uses as the example of his technique. However, as he notes in the paper, some DVI drivers are offended by the behaviour of the fonts he produces; naturally, the driver that the production team is one of those.

Output

Though individual articles were worked on by members of the production team on their local computer systems, the final output was prepared (by Mimi Burbank) at SCRI on an IBM RS6000 running AIX, using the *Web2C* implementation of T_EX. Output was printed on a QMS 680 print system at 600 dpi.

TUG and You—Together We Can Do It

Michel Goossens, TUG President

CERN European Laboratory for Particle Physics

CH-1211 Geneva 23

Switzerland

Email: president@tug.org

On Sunday July 23rd the new TUG Board (after the spring elections) convened formally for the first time in Saint Petersburg Beach, Florida.

As incoming President I therefore have the pleasure of thanking outgoing President Christina Thiele for her continued dedication and efforts with which, during the last two and a half years, she ran the T_EX Users Group. They have not been easy years and several difficult problems had to be faced and resolved. Christina, with the help of members of the outgoing Board, laid the basis for deciding where improvements can be made. I consider it the main task of the new Board for the coming year(s) to build upon the experience and efforts of the past to find better and more efficient ways to serve the T_EX user community in general and the TUG membership in particular.

I wish to congratulate Karl Berry, Judy Johnson, and Jiří Zlatuška (new members) and Barbara Beeton (who has been a Board member for many years) for their election by acclamation last spring. To fill vacancies due to resignations and shortfalls of candidates in the elections, I nominated Mimi Jett, Tom Rokicki and Norman Walsh. I welcome all of them to the Board and hope that, together with the remaining members Mimi Burbank, Michael Ferguson, Peter Flynn, George Greenwade, Yannis Haralambous, Jon Radcliff, and Sebastian Rahtz, we can, by our enthusiasm, dedication, and team-work ensure that TUG remains strong and healthy.

I also want to express thanks to the outgoing Board members Jackie Damrau, Luzia Dietsche, and Nico Poppelier, who all three had to resign for professional reasons, and to Michael Doob, whose term came to an end, for all the time they have dedicated to their work relating to TUG.

Next a word about the Special Directors, who joined the TUG Board as representatives of the local T_EX User groups in 1989 to increase the awareness of problems non-North American users face when using T_EX. Having only five “international” representatives, although historically correct in 1989, no longer reflects the real situation, and in order to allow TUG to find ways of proposing a better

representation for all T_EX User Groups worldwide, the representatives of DANTE, GUTenberg, NTG, Nordic TUG, and UK TUG resigned so that the title of Special Director could be abolished. I want to thank Joachim Lammarsch, Bernard Gaulle, Johannes Braams, Dag Langmyhr, and Chris Rowley (and their predecessors) for their many valuable contributions, and I sincerely hope that we shall be able to count on the continued support of all these and the other user groups in the future.

At this year’s Board meeting, a new Executive Committee was elected: Judy Johnson was elected as Vice-President, Sebastian Rahtz as Secretary and Mimi Jett as Treasurer. I look forward to working with this new team and the Executive Director Patricia Monohon, and hope that the fact that both Judy and Mimi are located on the West Coast will prove a great plus for the Office, which has had to work somewhat in isolation the last year or two since all officers were somewhat remote. At this point it is a pleasure to say thank you to the outgoing Secretary Peter Flynn, who acted as the “TUG scribe” for all these years, and to George Greenwade, as outgoing Treasurer, who gave some extremely valuable input but has been practically unavailable for many months due to an extremely busy professional schedule.

Change in continuity is to be the theme of this new Board. We must find ways to take advantage of new technologies to provide tools that our present-day working environments need. We no longer are using the large and costly mainframes or limited PCs of ten years ago, but many of us now have powerful personal computers with large disks and a CD-ROM reader on our desks at home or at work. We should take this into account, together with the fact that T_EX is a “tool” for most people, not an end in itself. Physicists, mathematicians, scientists in all fields, writers of all kind—even I, for my personal letters—want to have the most appropriate and user-friendly tool that gets the job done. Therefore we should look around and live in symbiosis with progress in other text-processing areas that are

evolving every day. HTML/SGML, hypertext, Acrobat, multi-media, colour, multilingualism, multi-byte encodings are only a few of the items on the shopping list of every “writer” in a modern environment, and we should be ready to face these issues and come with working extensions that address these questions.

Above I mentioned the issue of a better representation of the various user groups in the workings of TUG. During TUG95 we discussed various scenarios to build an equitable structure giving each User Group that wants to join forces the chance to influence decisions related to T_EX. Such a structure should be based on mutual trust and sharing of responsibilities, work and support. TUG should act as catalyst for ideas and should provide a framework to coordinate global developments, to eliminate wasteful duplication of efforts and to concentrate on agreed points of action. But we should be careful not to fall victim to committee-itis. Small technical working groups with complete autonomy and led by one or two dedicated individuals should provide the main thrust of our efforts. TUG’s rôle should be concentrated on coordinating funding and assigning of priorities.

But this is all music for the (near) future. Today we are facing a falling membership (2400 TUG members in 1994, about 1750 so far in July 1995) and one of the main problems is the non-appearance of our flagship publication *TUGboat*, which is now three issues behind schedule. I made it my first task as new President to get *TUGboat* back onto schedule by the end of this year, promising four issues before Christmas 1995. Since April of this year the TUG Publications Committee has discussed the best way to remedy this situation and we have come up with the following detailed plan.

We have set up a core team, and a production environment at SCRI (the Supercomputer Computations Research Institute at Florida State University) is now in place allowing this team to work on the various articles in a coherent and coordinated way. Barbara Beeton remains in charge of the overall process, but with the more practical production tasks she will be assisted by Mimi Burbank, Robin Fairbairns, Sebastian Rahtz, Christina Thiele and myself. All these people are confident that they have free time available during the next year or so and they have shown in the past that they can produce high-quality output in a timely way as proceedings editors or editors of other T_EX magazines. At the Business meeting I clearly stated that I take it upon myself to do everything possible, together with this core team, to get four *TUGboat*’s

on our member’s desks before Christmas 1995, i.e., these four issues of *TUGboat* have to go to the printers before the end of each month between August and November. *TUGboat* 15(4) and *TUGboat* 16(1), each of about 100 pages, will be regular issues containing articles submitted since mid-1994, *TUGboat* 16(2), with Malcolm Clark as editor, will be a theme-issue, contain articles related to electronic documents, in particular SGML, HTML, hypertext, and Acrobat, while *TUGboat* 16(3), the proceedings issue, edited by Robin Fairbairns, will contain not yet published papers presented at the conference, including all prize-winning articles mentioned in my report on the TUG95 Conference.¹

On a longer time scale, we will have to break the various tasks down into smaller assignments, with very detailed and clear instructions. This will take some time and coordination, and Barbara Beeton has circulated to those who stepped forward at St. Petersburg an outline of *TUGboat* production procedures. At the same time, I would like to repeat what Barbara already said, namely, that there are other important tasks that have little to do with editing, correcting or running articles through T_EX. We need authors of good specialist, introductory, and tutorial-level articles. So everybody can help us find amongst their colleagues or friends potential writers who can provide material for publishing in *TUGboat*. I am confident that a lot of developments and experience are still hidden in the grey matter of potential contributors. With a little prompting it can be made to see the daylight, materialize into sentences and drawings, and thus be shared in all its splendour with all T_EX users of the world.

Let me now give an overview of some of the major developments of the recent past. As many of you will already know, the TUG Office has moved to San Francisco (the new address is: *T_EX Users Group, 1850 Union Street, Suite 1637, San Francisco CA 94123, USA*. Phone: (+1) 415 982 8449, fax: (+1) 415 982 8559. The email address remains unchanged as tug@tug.org). There were many reasons for this move, the main ones being that our lease was coming to an end in December 1995 and the owners offered to pay the complete moving expenses if we moved before the end of June. So we gracefully took on the offer (knowing that in any case the lease would not be renewed after 1995). Moreover, San Francisco, being a much larger town than Santa Barbara, offers vastly improved connectivity to the Internet, has

¹ Since you are reading this now, we have kept our word, and, indeed, we are now up-to-date with the *TUGboat* issues, and, moreover, well underway to also get the last *TUGboat* of 1995 to the printer before Christmas.

cheaper rates for telephone, rent, personnel charges, and, with all the universities in the vicinity, it will be much easier to find teachers and rooms for organizing courses or volunteer effort for other TUG activities. Let me take this occasion to congratulate the Office staff for the perfect planning and efficiency with which they have handled this major enterprise.

To make the Office more visible to the outside world and hence to promote \TeX , and to provide better service to TUG members, I have asked our Executive Director to take the necessary steps to work closely together with the PR-Committee consisting of Jon Radel, Tom Rokicki and others, to define actions (via the Internet, direct mailing, publishing in magazines, targeting specific groups, like math teachers, etc.) to increase the awareness of \TeX among the public. The Office will also be responsible for running courses, selling \TeX related material (like books, diskettes, CD-ROMs), provide infrastructure, act as liaison and otherwise support in any possible way the organizers of the yearly TUG Conference, plus other topical ones, if a need is felt. With the active participation of TUG Board members and other volunteers we are planning to run mail servers for various TUG-related discussion lists and develop and maintain the TUG WWW pages on the Web. All in all the operations in the Office will be streamlined so that it can respond better to the needs of \TeX users of the middle nineties.

A joint membership arrangement with the Dutch (NTG) and British (UK TUG) \TeX user groups allows their members to also become TUG members with a discount of 10%—and the reverse is also true. This possibility is greatly appreciated, since bank transfers between Europe and the States are not always efficient and can, moreover, be quite expensive. We hope to continue this arrangement and extend it in one form or another to other user groups if there is interest.

It has been decided that the publications of most user groups will be published on CD-ROM in the near future. NTG volunteered to coordinate this effort.

As further evidence that TUG takes its international rôle seriously it was decided to hold the TUG 1996 annual meeting in Dubna, Russia, where we shall be the guests of the Joint Institute for Nuclear Research, an international Laboratory where scientists of many countries have been doing basic research for many decades. Dubna is a small town on the Volga river some 100 miles north of Moscow. The proposed dates are July 28th to August 2nd. This will be a unique opportunity not only to meet \TeX users from several less familiar countries, to

exchange ideas and discuss local developments, but also to visit a few of the treasures of Russian art and civilization, and to get to know the famous Russian hospitality. Information is given elsewhere in this *TUGboat*; more details will be published in *TUGboat* 16(4) and 17(1).

Since the beginning of the year our magazine *\TeX and TUG News (TTN for the initiated)* has a new editor: Peter Flynn succeeds Christina Thiele. Since June 1991, when *TTN* number 0 came out, Christina has made sure that all TUG members could read at regular three-month intervals news, short non-technical articles, announcements, book reviews, all nicely complementary to *TUGboat's* more academic style. Many thanks Christina, for the hard work. I am confident that Peter will do his best to make *TTN* as interesting and fun to read as before, at the same time adding his own typographic style and experience to give *TTN* his own personal touch.

The year 1994 had a balanced budget, essentially thanks to supplementary income from courses and the sale of books, and the membership fees of 2400 members.

For 1995 we foresee a deficit of between \$20,000 and \$30,000, essentially due to the fall of the membership numbers by about 20%. Most non-renewers state that they will only renew when they get *TUGboat* on schedule again. So, with the actions relating to the publication of *TUGboat* outlined above, plus a publicity campaign planned for the end of this year, we hope to get back most of the members we have lost since 1994 and also get new ones. Therefore we proposed a balanced budget for 1996. Of course we hope to do better, since an efficient organization is one with some funds to spend on interesting developments or for sponsoring conferences or other activities.

But attracting new members is not merely a matter of printing *TUGboat* on time; we must also offer other services that \TeX users want and cannot find easily somewhere else. Therefore we should try and involve as many \TeX users as possible with TUG, and that is why it is so important to set up a new structure for TUG, that would allow all members of the other \TeX user groups to fully participate in and benefit from TUG activities. This way we can probably double our membership, and approach again the number of 4000 members or so that TUG had in 1989–1990. By using the expertise and availability of all those potential contributors we can all together develop plug-and-play CD-ROMs for Unix, Mac, and other platforms, publish manuals about interesting software, sponsor new developments like $\varepsilon\text{-}\text{\TeX}$, Omega, and $\mathcal{N}\mathcal{T}\mathcal{S}$ (see these proceedings, or

my reports on the TUG94 and Euro \TeX 94 conferences on CTAN).

As I stated in my program statement accompanying the ballots, I consider it my main goal to make TUG into the real international home for all \TeX users in the world, an organization that can help new groups form, advise existing groups or individuals about where they can find help, coordinate common developments in the area of text processing in its widest sense, represent the interest of \TeX in the (ISO, ANSI, ...) standards groups, foster ideas and bring people together by making them aware of one another's existence.

But this ambitious program can only succeed when I have the full support of everybody. Therefore, I shall go and talk to the representatives of the various user groups, to find out which is the best way to work together, and to determine possible scenarios to form the basis of a collaboration.

I hope that I have convinced you that it is in the interest of all of us, \TeX users of the world, to unite, and work together harmoniously to transform \TeX , METAFONT and friends, graciously offered to humanity by Knuth, into even better performing tools

for the 21st century. We must not sin by conservatism nor by overzealous revolutionary actions, but we must take into account that the world around us is continuously changing, and that we should use these changes to our advantage by including useful extensions into \TeX , \LaTeX or other programs.

All that does not move is dead, and the last thing we want is that \TeX should die. All living things evolve and so must \TeX . Together, in a well-determined and agreed way, we should define how much change is needed, desirable and implementable. Then we can assign the necessary resources and get the job done, once, everybody working in unison. That is important, since we do not want a cacophony of rival versions. Therefore a world-wide collaboration in a global forum is so important, and it is my sincere conviction that only TUG can offer it. I count on your continued support to make it all happen.

If you have comments, suggestions, or just want to say hello, I can easily be reached by email as president@tug.org

When METAFONT Does It Alone

Jiří Zlatuška

Faculty of Informatics

Masaryk University

Burešova 20

602 00 Brno

Czech Republic

Email: `zlatuska@informatics.muni.cz`

Abstract

Combining METAFONT and T_EX when typesetting text and graphics together has been shown on several occasions to bring very impressive results. A. Hoenig presented a method for communication between T_EX and METAFONT in order to solve two problems otherwise difficult to handle within T_EX or METAFONT alone: label placement for diagrams generated by METAFONT, and curvilinear typesetting. We show that the method for curvilinear typesetting (involving three passes in Hoenig’s approach) can be considerably simplified by using the extended ligature mechanism of T_EX 3, and that a single METAFONT pass is actually sufficient, with quite a simple interface on T_EX’s side. Institutional seal text placement can be realized as a simple METAFONT application using this method. While PostScript offers ready-to-use easy solutions to this class of problems, METAFONT solutions can still be preferable to PostScript because of the ability of adding META-ness, e.g., by introducing second-order magnitude corrections/distortions to the letters and/or logos in order to enhance legibility when used in smaller sizes.

Introduction

There are several methods available for including graphical information into T_EX documents. Some of them rely on the `\special` primitive of T_EX and consists in combining pictures created by tools independent of T_EX on the level of `dvi` drivers. Within the T_EX world, the METAFONT program can be used for defining graphic objects by using its capabilities as in the case of defining letterforms, resulting in a “font” containing graphic images as “letters” which can be typeset within a T_EX-composed document.

There are interesting possibilities arising from combination of METAFONT and T_EX especially when it comes to typesetting text material along curved baselines and/or combined with other pieces of graphical information. Effects of this kind can also be prepared using PostScript transformations as prepared by, e.g., the `pstricks` collection by Timothy van Zandt. Nonetheless, reasons can be found for preferring a solution using just the combination of METAFONT and T_EX, excluding effects caused by combination of the `dvi` driver and the underlying printing language. One of them can be the necessity of using either a printer or a previewer which does not understand PostScript.

Another may be the need to use non-linear effects within the generated pictures, e.g., scaling the proportions of letters used within them similarly as they change when changing design sizes for METAFONT-generated fonts.

One of the problems of using METAFONT easily for creating pictures involving also text parts, is the lack of ‘typesetting’ capabilities (solved in John Hobby’s `metapost` generating PostScript output from an input formulated in a language extending METAFONT) which would allow efficient incorporation of typeset text into METAFONT-generated figures. Alan Hoenig (Hoenig, 1991; Hoenig, 1992) defined a scheme for bidirectional communication between T_EX and METAFONT allowing T_EX to submit requirements for special effects under which METAFONT would generate particular instances of the letters (e.g., rotated and/or scaled) and T_EX would place these letters onto the appropriate place within the typeset material. One particular application of this T_EX and METAFONT “working together” was curvilinear typesetting when typesetting centred text around the circumference of a circular area as used for institutional seals or logos. In this paper we show an approach for tackling

this problem within a simpler scheme than the three-step method described by Hoenig. We use the capabilities of the ligature programs of METAFONT to create composite pictures which can be then invoked from within T_EX documents with a certain level of “intelligence” built into them. This can be simpler to use than the three-step method and the composition steps embedded into the font definition corresponding to the particular piece of graphics.

Typesetting along curved baselines with METAFONT

When typesetting parts of a text using METAFONT in non-standard ways such as placing the text along a curve and/or combined with other graphic objects, it is often necessary to break the picture into separate parts stored as individual characters within a font which METAFONT generates as its output. There are several reasons for doing this. The resulting METAFONT picture comprising the picture as a whole may be too large for METAFONT’s memory limitations. We may also want to be able to use parts of the picture independently of the others, or to select just a few of them in particular cases. On the level of typesetting the pictures in T_EX, it is necessary to be able to typeset the fragments of the picture (characters from the font representing it) at the proper places in the typeset material.

We can illustrate some of the requirements which should be handled by a METAFONT-based definition of an institutional logo for the author’s home institution—a logo of the Faculty of Informatics of Masaryk University consisting of an Escher-like graphic based on a design by Petr Sojka, encircled by a pair of Latin inscriptions typeset around the circumference with different orientation each. The logo as such looks as follows:



The basic variations we may have in mind may be typesetting just the graphic drawing inside the seal, typesetting just the inscription alone, skipping out the shaded parts—hence obtaining variants of the picture looking as follows:



Hoenig’s method A. Hoenig proposed a method for combining METAFONT and T_EX in such a way that a sequence of three steps of communication takes place between METAFONT and T_EX. First, T_EX makes basic measurements of the text parts to be typeset. Second, METAFONT reads this information, generates the pictures and/or transformed letterforms and passes this back to T_EX as a font together with numeric information (e.g., positions onto which the characters should be typeset) encoded as kerns between pairs of special structure. Third, T_EX reads the metric information associated with the font, extracts any encoded data which are needed and then typesets the generated characters onto specified positions.

Although the communication between METAFONT and T_EX is solvable in this way, the resulting process is rather complicated. It is hard to imagine the technique becoming so easy to use that the resulting graphics could regularly be invoked in non-expert users’ documents.

Leaving the placement to METAFONT The final composition of the picture is left to T_EX in Hoenig’s “METAFONT cooperates with T_EX” method, and this is also the reason that communication between METAFONT and T_EX is introduced.

There is a simpler possibility of leaving the whole job of placing the parts of the final picture to METAFONT alone. METAFONT can generate characters which are placed correctly with respect to the resulting picture and use a common point of the resulting graphic composition as the reference point of each of the characters generated as parts of it. METAFONT knows this information in any case, so it can just use it for changing the `currenttransform` transformation in order to move the character to the desired place. (Note that METAFONT will not exceed its memory limits if it just moves the picture within the coordinate system without actually setting on pixels far away from it.)

The resulting font METAFONT generates consists of characters which should be superposed one on top of another. The point where this should occur from T_EX’s point of view is the common reference point of the generated characters. A T_EX loop independent of the structure of the picture can be used for this—just reserving space for the picture within the typeset document and overprinting all the characters from the font within the loop. In order for this to work, the widths of the individual characters in such a font are set to zero so that sequencing the characters on T_EX’s input actually means printing them on top of each other.

The first four characters needed to typeset the upper part of the curved inscription above are (the dot indicating the reference point):

Overprinting them on top of each other yields:

Character definitions In order to generate these characters, we have to modify the METAFONT program files so that the letterforms are properly transformed and to add the code for computing their parameters.

The basic change in the METAFONT programs for characters can be done following the way A. Hoenig used, with just a few extra parameters added because the placement of the calculations should be based on them.

The code defining letters of the form

```
cmchar "The letter F";
beginchar
  (n,11.5u#-width_adj#,cap_height#,0);
  ...
endchar;
```

will be replaced by METAFONT macros of the form:

```
width.F:=11.5u-width_adj;
def F_(expr n, rotation_angle,
  position_shift) =
  currenttransform:=identity
    rotated rotation_angle
    shifted position_shift;
def t_=transformed
  currenttransform enddef;
cmchar "The letter F";
beginchar
  (n,11.5u#-width_adj#,cap_height#,0);
  ...
endchar;
```

In this transformation we extracted the width information concerning the character (which will be needed for proper character placement) and defined a macro generating an instance of the letter as slot number n in the generated font consisting of the letter rotated by angle `rotation_angle` and moved to position given by vector `position_shift`.

Note that `currenttransform` in this definition may be further modified by other transformations needed. When typesetting texts in circular logos, it is for example good to stretch the letters a bit when the size of the logo becomes smaller. This can be achieved by introducing a global parameter `taller_letters` (e.g., to depend on the sec-

ond order of logo size change), and modifying the `currenttransform` setting to

```
currenttransform:=identity
  yscaled taller_letters
  rotated rotation_angle
  shifted position_shift;
```

Computing character positions For character position calculations it is enough to incrementally move the reference point of the text characters along the circle and to compute the positions and angular shifts of the letters to be typeset. These calculations can be carried out analytically, and use of the `solve` macro is not needed (in contrast with Hoenig's method).

For the upper arch of the circular text, the character position calculations are based on the widths of the characters only, and for the lower arch also on the height of the caps (because the characters should be shifted out of the basic circle by this distance).

The essential piece of information is the widths of the characters (including any kerning which follows them—as Hoenig notes (Hoenig, 1992), it is better not to rely on the default kerning used for linear text). We define an array for this,

```
numeric c[];
and fill in the width information including kerning
for the circular text such as
c[1]:=width.F+kkk;
c[2]:=width.A+kk;
c[3]:=width.C;
c[4]:=width.U;
c[5]:=width.L+kk;
c[6]:=width.T+kk;
c[7]:=width.A;
c[8]:=width.S;
...
c[chars_placed_up]:=width.AE;
```

```
c[first_down]:=width.U;
...
c[last_down]:=width.A;
```

Now three arrays will be defined,

```
numeric centering[],
  rot_angle[];
pair pos_shift[];
```

for recording the information concerning rotation angle and position shift of each of the individual instances of the letter, and an auxiliary array used for centering the texts along the vertical axis.

Now based on the character widths in the `c` array we are ready to calculate the co-ordinates of each of the characters `c[1]` up to `c[chars_placed_up]`

placed on the upper arch. Note that two passes are done here. The first one starts typesetting at 180 degrees, calculates the overall angle length, and sets `centering[0]` to the actual angle where centered text should start from. The second pass then recalculates the positions and angles starting from this corrected initial setting.

```
centering[0] := 180;
for j:=1,2:
pos_shift[1] := radius*dir(centering[0]);
for i=1 upto chars_placed_up:
  half:=1/2 c[i];
  halfdist:= radius +-+ half;
  centering[i] := centering[i-1]
    - 2 * angle (halfdist, half);
  pos_shift[i+1]:=radius*dir centering[i];
  rot_angle[i] := angle (pos_shift[i+1]
    - pos_shift[i]);
endfor;
centering[0] := 180 - 1/2 centering
  [chars_placed_up];
endfor;
```

Parameters of the characters placed into the lower arch are calculated in the opposite direction using the same approach. We just need to align the upper parts of each of the characters and to move the reference point out of the base circle—hence the difference in calculating `pos_shift[i]`:

```
centering[last_down+1] := 0;
for j:=1,2:
pos_shift[last_down+1] :=
  (radius + cap_height
    * taller_letters)
  * dir(centering[last_down+1]);
for i=last_down downto first_down:
  half:=1/2 c[i];
  halfdist:= radius +-+ half;
  centering[i] := centering[i+1]
    - 2 * angle (halfdist, half);
  pos_shift[i] := radius*dir(centering[i])
    + (radius + cap_height
    * taller_letters)
  * (dir(centering[i+1]
    - angle (halfdist, half)))
  - radius * (dir(centering[i+1]
    - angle (halfdist, half)));
  rot_angle[i] := angle (pos_shift[i]
    - pos_shift[i+1]) + 180;
endfor;
centering[last_down+1] :=
  centering[last_down+1] - 1/2
  * (180 + centering[first_down]);
endfor;
```

Generating the characters Now we are ready to generate the actual instances of the characters according to arrays `rot_angle[]` and `pos_shift[]`. We just need to pass the information to the appropriate procedures:

```
F_(1,rot_angle[1],pos_shift[1]);
A_(2,rot_angle[2],pos_shift[2]);
C_(3,rot_angle[3],pos_shift[3]);
U_(4,rot_angle[4],pos_shift[4]);
L_(5,rot_angle[5],pos_shift[5]);
T_(6,rot_angle[6],pos_shift[6]);
A_(7,rot_angle[7],pos_shift[7]);
S_(8,rot_angle[8],pos_shift[8]);
...
```

This font can now be used from within \TeX by saying, e.g.,

```
\char1\char2\char3\char4
\char5\char6\char7\char8
```

in order to generate the following fragment:



Mounting the pieces together using METAFONT It would still be clumsy to use METAFONT in order to generate the pieces of the picture, but still to have to compound them together manually within \TeX as the example above suggests. Fortunately we can do better, using the ligature mechanism of \TeX fonts. A similar trick is used within F. Sowa's `bm2font` or K. Horák's (Horák, 1994) method for decomposition of big METAFONT pictures.

Combinations of at least two letters from a font occurring adjacent to each other in the \TeX source suffice for invoking METAFONT's ligature program. Unlike the common ligatures used in ordinary Latin alphabet fonts, ligatures employed for this purpose make use of the fact that ligature handling is defined as a simple rewriting system rewriting pairs of codes into results consisting of inserting a new character and either leaving the source characters in, or removing them. Moreover, \TeX inserts a special "boundary" character before and after each word, including points where the font changes. Hence a simple way to define the full picture composed of the individual pieces is to define a ligature program combining the boundary character with a single letter triggering generation of the full picture. There can be several such triggers defining several parts of the picture.

Suppose for example that we want to be able to print three parts of the logo separately—the inscription, the Escher-like drawing inside of the logo, and

the colour areas inside of the drawing. Let us select three identifiers for this purpose – “S” standing for “seal”, “L” standing for “logo”, and “C” standing for “colour”. In order for the ligature mechanism to work, we add them as empty characters with zero dimensions:

```
beginchar("S",0,0,0); endchar;
beginchar("L",0,0,0); endchar;
beginchar("C",0,0,0); endchar;
```

Before designing the ligature program, let’s consider one more feature of the resulting picture. So far all the characters generated had zero width so that composing them did not change the position of the reference point within T_EX. This works for every character *inside* of the composition of the picture except for the first and the last – half of the “bounding box” of the resulting picture should be inserted there. Using slot 254 for the half of the bounding box we can define one additional character with non-trivial dimensions:

```
beginchar(254,radius#+cap_height#,
          radius#+cap_height#,
          radius#+cap_height#);
endchar;
```

Now the ligature program capable of starting everything off would have the form of:

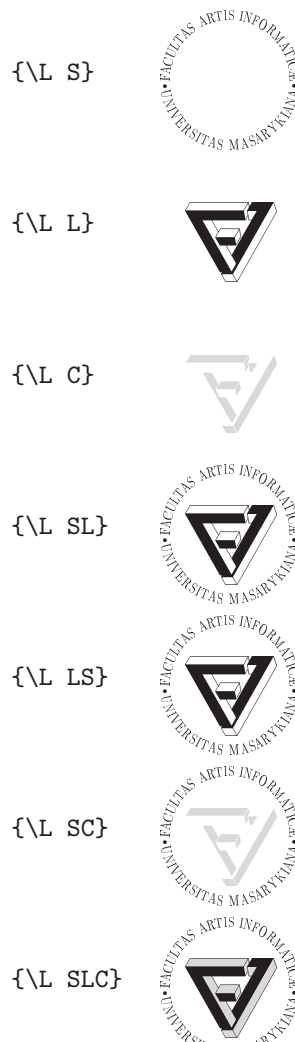
```
boundarychar:=255;
ligtable
||: "S"  =:| 254,
    "L"  =:| 254,
    "C"  =:| 254,
254: "S" |=:|> "S",
254: "L" |=:|> "L",
254: "C" |=:|> "L",
"S": "S" =:| 1,
  1: "S" |=:| 2,
  2: "S" |=:| 3,
  3: "S" |=:| 4,
  4: "S" |=:| 5,
  5: "S" |=:| 6,
  6: "S" |=:| 7,
  7: "S" |=:| 8,
  8: "S" |=:| 10,
10: "S" |=:| 11,
11: "S" |=:| 12,
12: "S" |=:| 13,
13: "S" |=:| 14,
14: "S" |=:| 16,
...
chars_placed_up: "S" |=:| first_down,
...
last_down-1: "S" |=: last_down,
last_down: "L" |=:|> "L",
```

```
"C" |=:|> "C",
255 |=:|> 254,
"L": "L"  =:| 254,
254: "L" |=: "A", %logo char
"A": "S" |=:|> "S",
    "C" |=:|> "C",
    255 |=:|> 254,
"C": "C"  =:| 254,
254: "C" |=: "B", %colour char
"B": "S" |=:|> "S",
    "L" |=:|> "L",
    255 |=:|> 254,
```

The font is intelligent enough to be used in such a way that after saying

```
\font\L=our-logo at 2cm
```

we can use the following input in order to define pictures of the form:



Driver problems The scheme outlined above works fine except for a minor problem with certain dvi drivers which may slightly distort the resulting

appearance of the complete picture. As a general rule, resetting `max_drift` to zero may be a good idea with most drivers, or else the first component may be slightly mis-aligned (alternatively one can add an empty character with zero dimensions to the beginning of every ligature chain in order to compensate for the drift with a harmless character first).

With `dvips` there's one more problem: it rejects empty characters with non-trivial dimensions. Before this gets fixed, the remedy may be including one pixel into the 254 character so that it's no longer empty. The pixel should be placed in a position that is set in any case. In our case there is no pixel shared by all the possible variants, hence the 254 character had to be split into some six other ones which are used depending on the context within the activated ligature chain.

Output drivers within the `emtex` family exhibit even more peculiar behavior: The characters to be overprinted are off-placed by positive horizontal skips so that the resulting picture gets completely distorted. Note this is not a problem with the `emtex` implementation which does generate a correct `dvi` file in this case, but purely a problem with the driver handling the somewhat unusual font rather unfaithfully.

Conclusion

We have described a method for composing images containing typesetting of circular texts and pictures with sufficiently rich functionality using just the possibilities offered by definitions in METAFONT alone. The ideas used mostly derive from A. Hoenig's ideas (Hoenig, 1992), yet are enough to locate all the necessary mechanism into a single METAFONT pass instead of invoking iterative processes involving communication between METAFONT and T_EX.

Acknowledgement This work has been supported by GA ČR grant 201/93/1269.

References

- A. Hoenig. "When T_EX and METAFONT talk: typesetting on curvilinear paths and other special effects". *TUGboat* **12**(4), 554–557, 1991.
- A. Hoenig. "When T_EX and METAFONT work together". In *Proceedings of EuroT_EX 92*, edited by J. Zlatuška, pages 1–19, Prague. 1992.
- K. Horák. "Fighting with big METAFONT pictures when printing them reversely or landscape". In *Proceedings EuroT_EX 94*, edited by W. Bzyl and

T. Plata-Przechlewski, pages 105–107, Gdańsk. 1994.

MetaFog: Converting METAFONT Shapes to Contours

Richard J. Kinch

6994 Pebble Beach Court
Lake Worth FL 33467 USA

Email: kinch@holonet.net

URL: <http://www.emi.net/~kinch>

Abstract

The Computer Modern Typefaces have their original specification in terms of the METAFONT language. The individual glyph programs rely on the sophisticated algebra and marking methods of METAFONT. Many of the METAFONT primitives, such as stroked pens and overlapping ink, are not directly expressible in outline typeface formats such as Type 1 and TrueType, which support only topographic contours expressed as non-overlapping Bézier curves.

We explain the computational geometry involved in the conversion from METAFONT shapes to outlines, why this is a difficult problem, and why previous efforts have fallen short. We describe MetaFog, a set of programs written to post-process METAFONT output to complete such conversions, and the algorithms implemented to solve the mathematical problems. The two most significant problems are (1) finding the envelope of an elliptical pen stroked along a Bézier curve (an algebraic problem), and (2) reducing overlapping paths to an equivalent, non-overlapping contour (a topological problem). We propose a scheme to embed Type 1 and TrueType hint technology into METAFONT sources to reduce the duplication of effort to produce well-hinted fonts. We compare the accuracy of MetaFog's analytic conversions to approximations based on auto-tracing of METAFONT's bit-mapped output, and show examples of errors in the Computer Modern Typefaces which are hidden in METAFONT proofs but visible in MetaFog proofs.

T_EX and its Fonts

Modern implementations of T_EX like TRUET_EX[®] have eliminated bit-mapped meta-fonts in favor of outline formats such as TrueType or Type 1. T_EX did an admirable job of producing its own font bit-maps in the days before operating systems supported fonts. But today the most popular operating systems and print engines require outline fonts. These scalable formats facilitate previewing and printing T_EX documents in a powerful, portable, and flexible fashion which bit-mapped fonts cannot achieve.

While pure T_EX is independent of any particular fonts, T_EX is nevertheless just as dependent today on Computer Modern and other METAFONT-based fonts as ever. Thus arises the need for conversion of METAFONT programs into equivalent outline forms.

While METAFONT programs can describe a glyph in terms of complex, overlapping paths, the outline formats require that we specify glyphs as a

set of *contours* (non-overlapping outlines). Herein lies the most difficult aspect of conversion: METAFONT's primitive shapes are built from third-degree parametric curves modulated by third-degree paths, and such shapes can overlap, add and subtract in arbitrarily devious ways.

Conversions: analytic versus approximate.

MetaFog is a system for exact, analytic conversion of METAFONT shapes to contours. That is, MetaFog always store shapes in terms of their pure, parametric curves. By “analytic” we mean that the methods we use analyze and solve the underlying equations for the parametric curves. We use no intermediate approximations such as converting curves to polygons, so that every result curve is a direct derivation of an input curve and every input point is unchanged in the output.

By “exact” we mean that the result curves follow the METAFONT shapes to within one pixel in the 1024 or 2048 pixels/em grid used in typical outline font formats. In some cases METAFONT design

envelopes cannot be represented exactly by Bézier curves, and we use this metric to determine the degree of curve-fitting needed. We use a METAFONT `mode_def` for a “perfect” output device needing no corrections for fill-in or overshoot.

Automating an analytic conversion of METAFONT shapes requires a major effort in both mathematics and software. It requires solutions to problems which Knuth managed to avoid in METAFONT by using numerical tricks and simplifications. Earlier projects have attempted the task, but either fall short of or approximate the full solution (Yanai and Berry, 1990; Carr, 1987; Henderson, 1989).

Outline conversions of meta-fonts have also been done before using approximation techniques, thus avoiding the difficulty of an exact, analytic conversion. For example, autotracing attempts to fit an outline to a high-resolution bit-map. With enough skilled labor, autotracing yields an aesthetically pleasing result, although the shapes will tend to have certain artifactual deviations from the precise METAFONT originals. The BlueSky-Y&Y conversions of Computer Modern and other meta-fonts show that careful autotracing and hand-tuning can produce a result equal to that of a conventionally-designed commercial font.

More recently Malyshev (1994) has published the BaKoMa fonts, which contain very precise outline conversions of Computer Modern. Malyshev’s publication is limited to the results (that is, the outline fonts themselves); he has not revealed the details of his technique, although he claims that it is analytic and not an autotraced or otherwise a digitized approximation. We will show below examples of font details which an analytic conversion would preserve, but which are missing from the BaKoMa fonts. Malyshev’s claim of analytic perfection could nevertheless be true, if such errors were introduced, for example, by bugs in his conversion software. On the other hand, if a hidden approximation is involved somewhere in the BaKoMa conversion process, the result would not meet our strict definition of being both “exact” and “analytic”. This is not to say that the BaKoMa fonts are poor conversions; it is evident that the shapes are excellent in every way important to font designers and that they are generally faithful to the METAFONT originals.

The Nature of the Conversion

Let us consider the nature of the conversions involved. METAFONT can actually do more sophisticated things than we are about to describe, but we will restrict our consideration to those META-

FONT features that are actually used in typefaces like Computer Modern.

Bézier curves. We will consider Bézier (Glassner, 1990) contours to be our target format. A Bézier *curve* (Figure 1) is a parametric curve governed by the equation:

$$z(t) = (1-t)^3 z_1 + 3(1-t)^2 t z_2 + 3(1-t)t^2 z_3 + t^3 z_4$$

Parameter t is called the *time* along the curve and ranges over the interval $[0, 1]$; the point at time t is $z(t)$. A Bézier *path* is a set of Bézier curves which connect in a chain at their endpoints to form a more complex curve. A closed path which does not overlap describes a complete circuit and encloses an area. A set of such paths make a Bézier *contour*, which can describe the outlines of a glyph. The paths in the contour of a well-formed glyph do not intersect each other, and as well they do not intersect themselves. This is the representation used in the Type 1 font format (Adobe Systems, 1990). Conversion of Type 1 glyphs to TrueType glyphs (which use lower-order parametric curves) is a straightforward conversion. In METAFONT (as documented in the literate source code), Knuth calls the Bézier paths *cubic splines* (an equivalent mathematical term), and uses a data structure consisting of knot locations and control points to specify paths. This is the terminology we use in MetaFog. In Figure 1, points z_1 and z_4 are knots, and z_2 and z_3 are control points.

The goal of MetaFog conversion is to produce Bézier outlines which accurately represent the METAFONT designs. This will be close to the minimal set of knots needed to fit the design, because both METAFONT and Computer Modern are economical in their use of reference points, and the reference points in a METAFONT program generally expand into the minimal set of knots to implement a fitted curve. Because METAFONT divides curves into octants, METAFONT’s curves tend to have control points every 45 degrees or so, versus Type 1 fonts which often subtend curves of 90 or 180 degrees per control point. So in this sense METAFONT designs have *more* control points than good Type 1 designs. On the other hand, the Type 1 format mandates rules for tangents and extrema points that tend to add redundant control points to designs, so in this sense METAFONT designs have *fewer* control points than good Type 1 designs. MetaFog preserves the pure METAFONT design, such as the addition of 45 degree control points and the absence of redundant extrema points versus a likely implementation in Type 1. The final conversion code will optionally add redundant points to meet the Type 1 mandates.

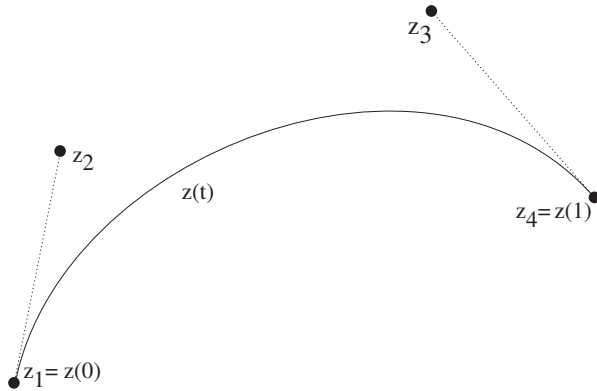


Figure 1: Bézier curve, starting at z_1 and ending at z_4 . The outgoing control point is z_2 , incoming is z_3 .

Before METAFONT digitizes a glyph into a bit map, it represents the glyph as a collection of shapes. Each shape can be an outline determined by a set of Bézier curves or the envelope of an ellipse stroked along a path. Each shape also can add or subtract ink. This is the internal representation which we wish to reduce to an equivalent set of Bézier outlines, which are the shapes which a Type 1 font uses directly or which can be easily converted into the shapes for a TrueType font.

METAFONT shapes also have color; in practice this means that we can think of each shape as either additive black ink or “white” ink that subtracts black ink already drawn. We can see that the order of drawing shapes in a glyph must therefore be preserved.

Within MetaFog we use the winding-number convention (like PostScript’s) for controlling color (black versus white), while METAFONT stores an explicit color for each shape. METAFONT shapes usually, but not always, follow a consistent winding direction for the associated color. MetaFog is careful to check shapes on input so that the winding number and color are consistent. When MetaFog discovers an inconsistency, it reverses the input path.

The different models treat edges differently when rendering bit maps. We have yet to take this into account in our conversions.

Bézier tools. MetaFog uses quite a few algebraic tools to manipulate curves. Some are re-implemented or generalized algorithms from METAFONT, and some are entirely new concepts:

- Find the coordinates of a given time value on a curve.
- Find closest time on a curve to a given point.

- Audit a curve, path, or contour data structure for consistency.
- Test whether a path is degenerate (zero winding number).
- Test whether a path is redundant (contained within) with respect to another.
- Test whether a path (possibly pivoted) duplicates another.
- Test whether two paths overlap (that is, have a common segment).
- Find all intersections between two curves, associating mutual intersecting locations.
- Find all intersections in a contour, associating them with each appropriate curve in terms of time.
- Sort intersection times associated with curves in a contour.
- N-sect a curve into N curves given a set of times.
- Given an ellipse, generate a four-curve approximation.
- Given an ellipse, find the point on the ellipse at a given angle from the major axis.
- Given an ellipse and an angle of rotation, find the maximum point (horizontal tangent) on the ellipse.
- Test if two line segments cross.
- Given the parallel curves of a stroked path, stretch or shrink the endpoints to fit a given ellipse with arbitrary rotation.
- Given a curve and a rotated ellipse, return the 6 to 8 curves fitting the envelope.
- Given the positions and tangents of a curve endpoints, and a midpoint position, locate the endpoints which fit the constraints.
- Test a curve for “simplicity” (that is, turning angle $\leq \frac{\pi}{2}$ and no inflections).
- Given an open path and an ellipse, return the envelope, reducing overlapping segments.
- Test whether a given point is on a curve (with given tolerance).
- Test whether a given point is inside a closed path.
- Test whether a given path is interior to another path.
- Find all circuits in a contour.

The above algorithms, plus syntactical and data-structure chores, make up about 12,000 lines of C program code.

Loading Shape Information from METAPOST. METAPOST (Hobby, 1989) relieves us of the difficult task of running METAFONT and extracting the Bézier curve information relevant to a character. We chose to have MetaFog interpret the PostScript output from METAPOST and to construct the MetaFog contour data structures during this interpretation, rather than trying to modify METAPOST to make output in a more convenient form. This allows us to stay current with METAPOST improvements.

METAPOST outputs outline curves in PostScript by first defining the path with `newpath`, `moveto`, `curveto`, `lineto` and `closepath` commands, followed by a zero-pen-width `stroke` and a `fill`. For “white” ink METAPOST uses `setgray` before stroking or filling. For elliptical pens and slanted coordinate output transformations METAPOST uses `dtransform`'s to apply affine transformations. MetaFog contains an input interpreter that converts METAPOST output to internal data structures.

Rendering ellipses stroking paths. One of the problems which Knuth sidestepped in METAFONT was computing the envelope of an ellipse stroking along a Bézier curve. Knuth here chose to use Hobby's method to compute the envelope in terms of the raster instead of scalable curves; the computational geometry then reduces to a matter of manipulating line-segments and polygons instead of polynomial curves (*The METAFONTbook*, §524).

We instead want to compute a Bézier curve outline for stroked-ellipse envelope. Algebra tells us that stroking a 3rd degree polynomial curve (the ellipse approximated by Bézier curves) along a 3rd degree polynomial curve (the Bézier curve of the stroked path) results in a 6th degree envelope curve. We will have to approximate these 6th degree exact envelope curves with 3rd degree (Bézier) curves.

Figure 2 shows how an ellipse contour may be approximated by a contour made from Bézier curves. This is similar to the four-curve approximation to a circle cited by Knuth. The Bézier control points for a unit circle are located symmetrically $\frac{4}{3}(\sqrt{2} - 1) \approx 0.552$ units away from the end points. (This quantity does not appear explicitly in METAFONT, but we can solve for it by substituting the known angles and locations at the ends and midpoints of the curves.) The affine properties of Bézier curves permit us to linearly distort the Bézier control points of the unit circle in proportion to the eccentricity of a unit ellipse to fit a Bézier contour to that ellipse. We can also apply linear

transformations of rotation, scaling, and translation to tilt, size, and place a unit ellipse as desired.

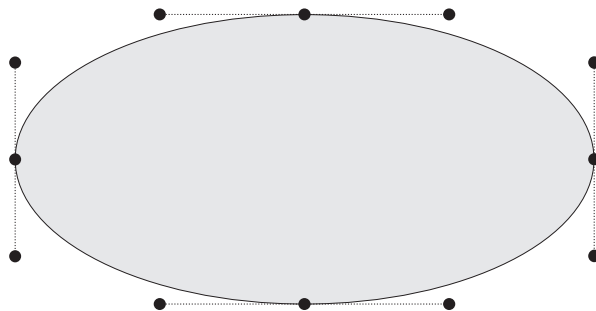


Figure 2: Contour of four Bézier curves which approximate an ellipse.

To proceed to the envelope problem, let us assume that the situation looks like Figure 3, where (without loss of generality) we have rotated and translated the coordinates such that the start of the stroking path is at the origin and has zero slope there.

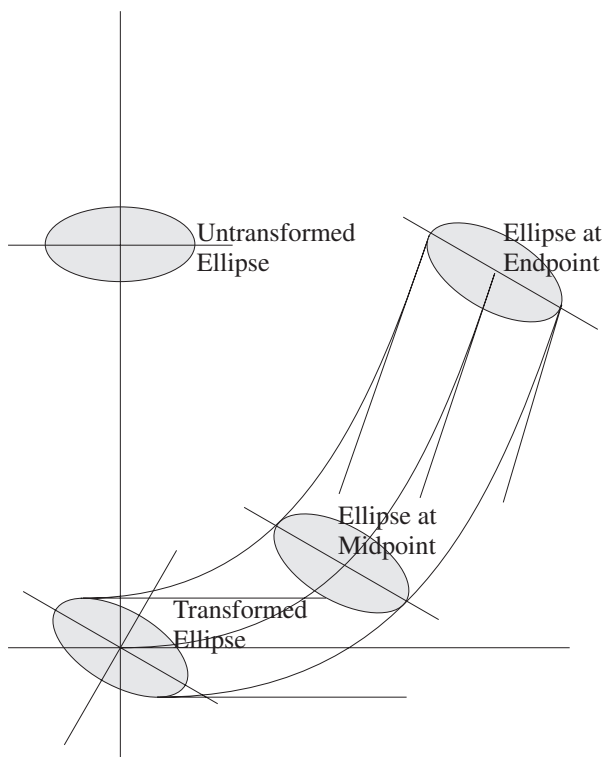


Figure 3: Stroking an Elliptical Pen on a Bézier Path.

We will fit an envelope consisting of two ends and two sides. The sides are “parallel” to the

stroking path, and the ends are subsets of the ellipse at the start and finish of the stroke. We use a set of boundary conditions for the approximation which will be natural and visually appealing: The slopes of the side curves start at zero and end with the same angle at which the stroke curve ends. We fix the midpoints and angles of the side curves based on the location of the ellipse at the midpoint of the stroke, using the tangent points of the ellipse matching the angle of the stroke at its midpoint. This approximation is quite good when curves are not too “sharp”; that is, they do not turn through more than 90 degrees, and are not too “tight”; that is, they do not have a high 2nd spatial derivative. We can always bisect sharp and/or tight curves to improve the accuracy of the approximation as needed; in practice the curves are almost always so gentle as to be well-fitted without bisection.

To compute the envelope curves, we must find their endpoints and their control point locations. We first translate and rotate the coordinates of the problem to the normalized coordinate system to fit the model. Using the boundary conditions—namely the endpoint locations, endpoint tangent angles, and the midpoint locations—a bit of polynomial algebra and a solution of simultaneous equations yields a closed-form solution to where to put the endpoints and control points of the envelope curves. Given these two curves, we can compute the subset of the ellipse curves as a maximization problem in another transformed coordinate system. Inverting the rotation and translation of coordinates yields the desired solution.

Figure 4 shows some examples of envelopes computed with this method. Careful attention to generality and numerical domains yields a robust algorithm, which is crucial to the wild data characteristic of graphical shapes.

METAFONT usually uses circles (of course, a circle is a special case of an ellipse) to stroke pens. The exceptions where METAFONT uses elliptical pens are the calligraphic capitals and a few math symbols. Knuth also used circular pens quite liberally in Computer Modern. For example, circular pens draw the rectangular stems, since the technique makes parameterization of stem widths and rounding of corners somewhat easier, and the serif programs take care of squaring off the round corners for Roman faces.

The logical shape primitives OR and NOT.

Once MetaFog expands any stroked paths to envelopes, it can proceed to intersect overlapping paths. MetaFog must compute all possible multiple intersections of each pair of curves in a path,

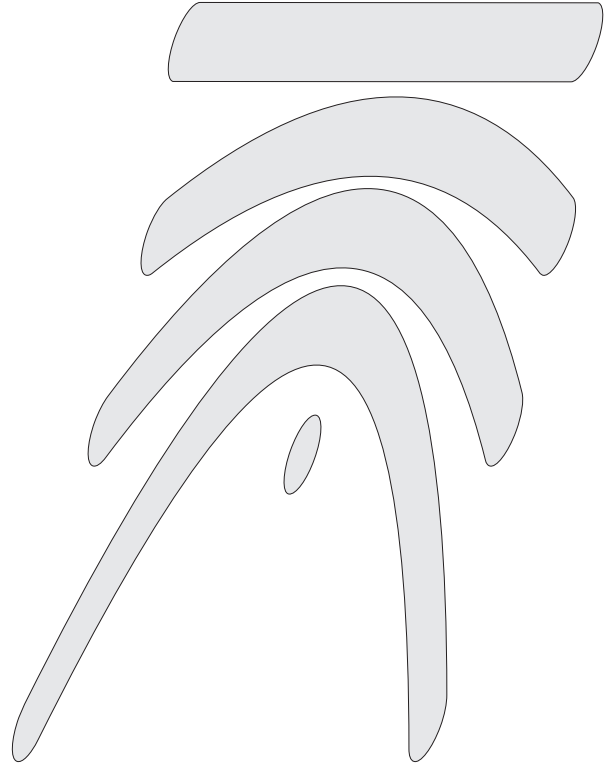


Figure 4: Envelopes computed for various Bézier strokes of an elliptical pen.

instead of assuming only one possible intersection as in METAFONT. (Two Bézier curves can have 8 intersections—try to find an example using your favorite drawing program!). MetaFog computes all intersections using an exhaustive extension to the recursive, numerical solution Knuth used in METAFONT; a closed-form solution employing zeroes to cubic polynomials is also possible but not implemented. Computing all such intersections and reconstituting the shapes with new knots at all intersections in the general case is a difficult problem which consumes most of MetaFog’s running time.

Weeding. The MetaFog weeder is a visual tool which allows a human operator to examine and hand-correct the output from automatic conversions. Manual input to the conversion process is vital, because METAFONT output often has degenerate shapes and intersections that defy an automatic solution. In such cases, MetaFog cannot determine which shapes are overlapping, and so outputs a partial solution to the topological problem; the weeder allows the human designer to choose the proper Bézier shapes from intermediate METAFONT elements.

Figure 5 shows the weeding display for character ‘m’ of `cmti10`. The display shows each of the Bézier curves of the input shapes, intersected and broken into separate pieces. The user has invoked MetaFog in “minimal” mode (which is guaranteed to succeed), which means that all curves used in computing envelopes of strokes are retained; an “intermediate” mode (which does not always succeed) reduces each envelope to the exterior curves. Note how MetaFog has stroked a circular pen along Bézier paths and produced curves for the envelope. MetaFog has also inserted new knots where curves intersect; this computation can be quite complex since a given curve can have arbitrarily many intersection points, resulting in a repeated bisection of the curve. The human operator uses the mouse to observe and toggle Bézier segments which make up the correct envelope of the character; each segment changes color as it toggles on (blue) or off (red). Toggling proceeds quickly because the mouse click need only be near (not necessarily on) the desired curve, clicks are buffered when the operator outpaces the CPU, and a second click will toggle off any inadvertently erroneous selections.

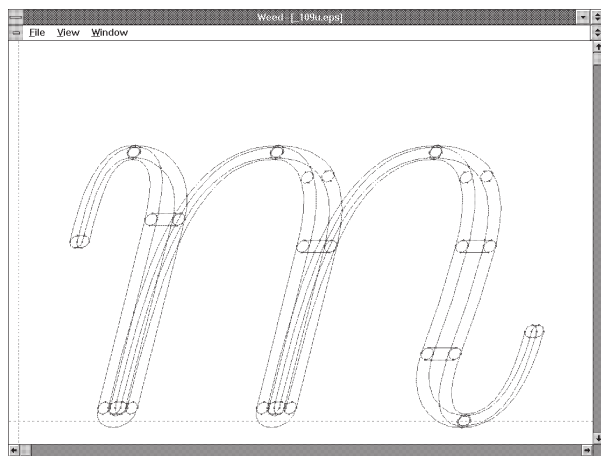


Figure 5: Weeder display for `cmti10` ‘m’

The weeder’s user interface is optimized for speed. The PC’s numeric keypad provides convenience functions, so that the operator keeps one hand on the mouse and the other on the function keys. The operator can quickly switch between displaying all curves and displaying selected curves only. Previewing selected curves only gives an accurate check that no segments are missing and that no extra segments are selected. Zoom-in and zoom-out allow the operator to pick through “busy” areas where many curves lie very close together. After toggling all the exterior edges of the glyph, the

operator visually checks the glyph for proper construction, and finishes by exporting the character. During export, the weeder takes care of optimizing the output curves by removing redundant control points. Keypad functions allow the user to flip quickly through all the glyphs in a font. This allows careful previewing and weeding of any glyphs that need touch-up from MetaFog. A checkplot program produces a printout of all the glyphs in a font for checking and documentation.

In the worst case, MetaFog can always produce a fully-intersected set of shapes with elliptical pen envelopes (if any) already expanded. The operator of the weeder then has a more detailed pointing job, but the result will be just as perfect as an automatic solution.

Figure 6 shows the MetaFog weeder view of `cmsy10`’s calligraphic `A`. This illustrates how a tilted elliptical pen strokes a Bézier path in slanted coordinates.

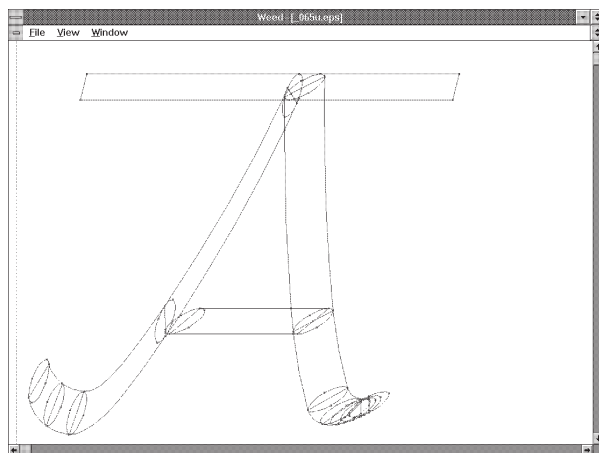


Figure 6: Weeder display for `cmsy10` calligraphic `A`, illustrating envelopes of elliptical pen strokes

A few cases of the calligraphic capitals contain tightly turning curves which require hand-corrections using the weeder.

Hinting. Rendering fonts on low-resolution devices like video displays and laser printers requires heuristic help to yield a pleasing result. Without such help, the bit maps will have unnatural bumps, stems will be of uneven width, and drop-outs will occur.

METAFONT handles these matters in the chapter “Discreteness and Discretion” of the *The METAFONTbook* and via `mode_def` items like `fillin` and `o_correction`. The Type 1 font language allows the designer to add “hinting” for the same purpose. TrueType calls the same notion “instructions”. Since most of the industry effort in this regard has

been expended on Type 1 font hinting, “hinting” has become the generic term for this aspect of font design.

METAFONT has significant modeling differences from the outline font hinting methods, so there is no translation possible to automatically make a hinted outline font from a METAFONT design.

Hinting can be applied after the translation, either automatically by auto-hinting software (yielding a poor to modest result) or by a skilled programmer (yielding the best results, given enough expert effort). The big problem with adding hints in this “post facto” manner, is that the hints become detached from the original METAFONT programs, and any change to the meta-fonts will require repeating the manual hinting effort. The biggest loss here is that the meta-ness of METAFONT programs does not carry over to post-facto hand-hinting; one METAFONT character program makes variants like bold, italic, sans serif, etc., but each variant must be independently hand-hinted. Another important example is that most of the METAFONT programs for the DC fonts repeat programs from Computer Modern, but such redundancies would not be usable after translation to outline formats. Outline-format fonts do suffer from an inability to exploit these redundancies, and serious font designers typically have in-house tools to overcome this problem.

Since METAFONT hinting does not carry over to Type 1 or TrueType hinting, the ideal solution would be to enhance the Computer Modern METAFONT programs to contain new hinting information suitable for translation to other forms. Type 1 and TrueType hinting employ a limited number of techniques, which depend on the exact coordinates and design of each particular character. A programmer could add each hint and the associated coordinates to each character’s METAFONT program in the form of pseudo-comments. A hint-translator program would convert the METAFONT pseudo-comments into Type 1 hint programs or TrueType instructions. Making the shape translation independent of the hint translation would allow adjusting shapes or hints independently, without having to re-run both aspects of the translation. The pseudo-comment language would be designed to represent the various hinting technologies and to exploit any commonality between them. The METAFONT language is well-suited to an extension of this sort.

For example, Type 1 “flex” hinting needs to know the size and position of what is called the “dish” concavity in the Computer Modern serifs. Addition of this information to the Type 1 fonts

improves the rendering of serifs. While this information is present in the METAFONT programs, it is lost in the process of translation to output shapes. The proposed method of pseudo-comments and hint-translator would preserve and translate this information. Hints are typically applied to stems, bowls, bulbs, and other character features, and METAFONT is quite aware of the pertinent coordinates of these items.

This is also a database problem. One of the difficult tasks of translating T_EX fonts is the surfeit of them. Just between Computer Modern and the DC fonts, spread across various optical sizes, there are several hundred fonts each having 128 or 256 glyphs. Given that the typical glyph outline contains dozens of endpoints, each having 3 pairs of coordinates, one can see that the translation enterprise involves millions of coordinates. Organizing this information into glyph data, character names, fonts, character metrics, encodings, accent composition rules, version controls, kerning pairs, ligature rules, font families, output formats, hinting data, and so on is a substantial database problem. Since we want to exploit redundancies like common subsets between OT1 and T1 encodings, we especially need a capable database approach to managing this information.

MetaFog uses more of a rapid-prototype approach. Shell scripts manage the various steps in translating a given font: running METAPOST to get intermediate conversions; running MetaFog itself to convert all or part of a given font to outlines, assembling various files for a C program `makefont` which assembles individual character data into complete Type 1 fonts, including insertion of extrema points, initial production of an AFM file, and a T_EX virtual font file. Tables keep track of redundancies between characters and fonts so that a given METAFONT glyph need only be translated once. Tables such as encoding vectors are typically kept in ASCII form and look-ups are performed by shell scripts. Glyph information is kept in PostScript or pseudo-PostScript form and rapidly manipulated by C programs built from common function libraries.

To finish the fonts, we use several outside utilities. The programs of Hetherington’s `t1utils` collection take care of the details of conversions to and from the encrypted Type 1 font format, so that MetaFog need be concerned only with ASCII Type 1 output. We also test the fonts with all the commercial font editors currently available: *Fontographer*, *Fontmonger*, *Type Designer*, and *FontLab*; we use

Fontmonger to convert the Type 1 fonts to TrueType form.

If we were to repeat the implementation, one might consider using a relational database to store the information, with query scripts and C programs doing the detailed work.

Optical Overkill. Fonts as they are used in operating systems today do not favor the optical scaling which \TeX is adept at exploiting. For example, \TeX uses eight optical sizes of the Computer Modern Roman font (5–10, 12, and 17 points). This is too many optical sizes—do we really need every step from 5 to 10 points? No doubt this was encouraged by the METAFONT facility at optically scaling with a simple parameter change. But with the various embellishments of bold, italic, and so on, a minimally complete Computer Modern font set yields over 100 discrete fonts.

Users today are not accustomed to seeing so many fonts associated with an application. \TeX has a distinctly archaic atmosphere in this regard. Operating systems that manage fonts are taxed to handle the plethora of tiny variations in \TeX fonts.

Lately this overkill of optical sizes has worsened with the NFSS, which does a good job of hiding optical sizes from the user, but encourages the style designer to multiply them.

The pain is excruciating with regard to outline translation, where essentially identical problems with slight variations are repeated many times. We would urge restraint on \TeX experts when it comes to selecting optical sizes.

Comparisons of Various Approaches

Let us compare a typical Computer Modern glyph as translated to outline form by various methods. Figures 7 through 12 show the output for ‘R’ of *cmr10* from various conversions.

Note that Figure 7 and Figure 8 show extra, relocated, missing, or artifact control points which have lost the symmetry of the METAFONT control points. The autotracing method used is evident in these examples.

Figure 9 has retained most of the METAFONT control points but also inserted artifacts. Figures 10 and 11 show the set of true METAFONT pieces from an intermediate step, where MetaFog has expanded the circular pen strokes into their Bézier envelopes. Figure 12 shows how MetaFog retains the METAFONT control points exactly, including all the octants and all the symmetry; there are no extra or artifactual control points. In comparing Figures 9 and 12, note that the tip of the leg in the BaKoMa

conversion develops an asymmetry, that the flat top of the tip has narrowed, and that the 45 degree control points are missing from the bowl and serif curves (which will underspecify these curves). The MetaFog conversion retains the proper symmetry, flatness, and precision, which are all aspects of this character readily observed in the METAFONT proof in *Computer Modern Typefaces*.

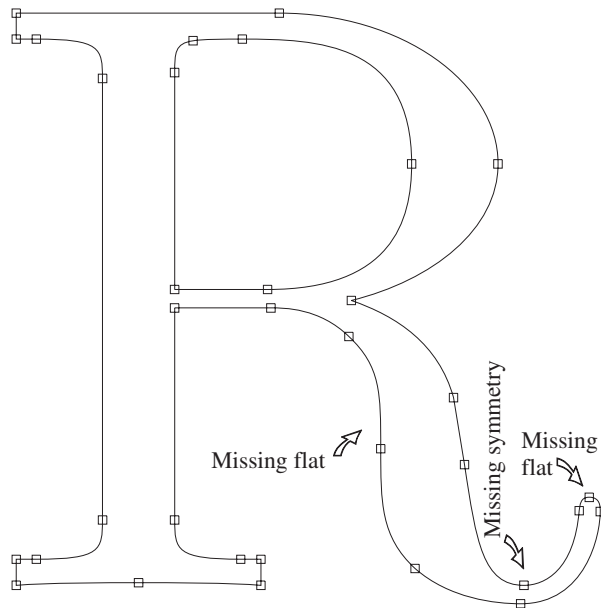


Figure 7: Blue Sky Research autotraced conversion.

X-Rays reveal bugs in Computer Modern. MetaFog allows more detailed visualization of character designs than METAFONT proofs. While the proofs show reference points and marked areas, they cannot show most of the relevant geometric information. Indeed, few of the knots, not all the outlines, and none of the stroked pen envelopes are accessible in METAFONT. Since MetaFog converts and manipulates all these items, it can also plot them in a convenient form. This yields a new and sometimes surprising “x-ray” view of a character—a view unavailable in METAFONT. MetaFog’s output files use a PostScript format so that proof pictures plot on any PostScript output device. The weeder is also a convenient visual tool for such views.

Surprising aspects to some of the Computer Modern designs show up in the “x-rays”. The parametric nature of meta-designed features becomes visually apparent, and bugs in the design are clear where they were not before. Figure 13 shows how the serif subroutine has introduced an unexpected

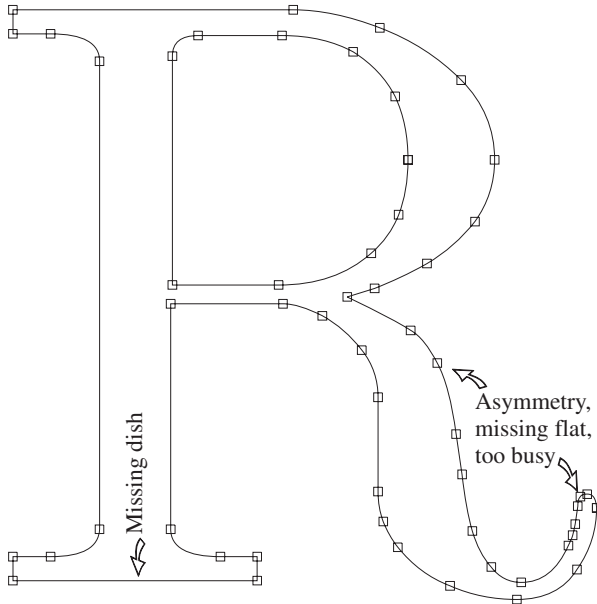


Figure 8: PC TeX autotraced conversion.

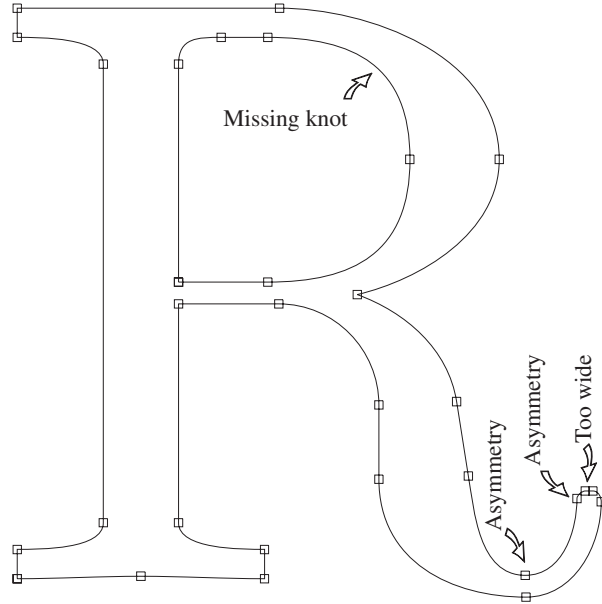


Figure 9: BaKoMa conversion.

inflection in `cmbx5` letter ‘x’. Since the loss removes just a few pixels likely to be filled in physically by most marking engines and optically by the human eye, the error is not obvious in normal usage or on METAFONT proofs. It becomes clear, however, during the MetaFog weeding.

Figure 14 shows how joining of the “beak” to the arm on digit ‘7’ becomes distorted at smaller sizes. This error is easily missed on proofs but is visible under magnification. If you magnify and carefully examine the actual-size proofs in the *Computer Modern Typefaces* (Volume E of *Computers & Typesetting*), this error is visible as a row or two of extra pixels at the top of the character.

A frank error in `dcr10`’s ‘thorn’ was easily discovered in this way, although it had escaped all the proof checks and actual usage for several years (Figure 15). The bottom serifs have an extra “step”, which on bit-mapped proofs looks like a purposeful fillet. On the MetaFog conversion it appears clearly as an error. (This error has been corrected in the autographs pursuant to this discovery, and does not appear on more recent versions.)

Is There an Exact Translation?

Is an exact translation possible? We used the standard of 1 pixel on a 2048 pixel/em grid. No doubt the “noise” of digitization and hinting creates many more varying pixels than this standard of error. There is no outline that will render the same bit map in a Type 1 or TrueType engine as METAFONT

would render for a METAFONT program. The font format itself requires that we must approximate sixth-degree pen strokes with third-degree pieces. METAFONT cannot draw proof picture of the underlying curves, it can only produce a high-resolution bit map. And finally, device-specific mode definitions in METAFONT result in significant changes to the “proof mode” device.

So there is no such thing, in a practical sense, as an exact translation, because there is no exact shape to what a METAFONT program describes! Perhaps we should instead speak in terms of an “ideal” translation.

Sample Output

A sample of MetaFog conversion, namely `cmr10` in Type 1 format, is available at:

[FTP://ftp.netcom.com/pub/Tr/TrueTeX](ftp://ftp.netcom.com/pub/Tr/TrueTeX)

This is an *unhinted* font suitable for viewing in a font editor, but not suited for textual use. MetaFog itself is a proprietary product, and is not in the public domain.

Colophon

We drafted this paper using the TRUE TeX implementation of L^AT_EX 2_ε for Windows, which allowed WYSIWYG previewing and printing, including all graphic images. We used three kinds of figures, and processed them all through Corel Draw 5: ordinary drawings, MetaFog imports, and screen snapshots.

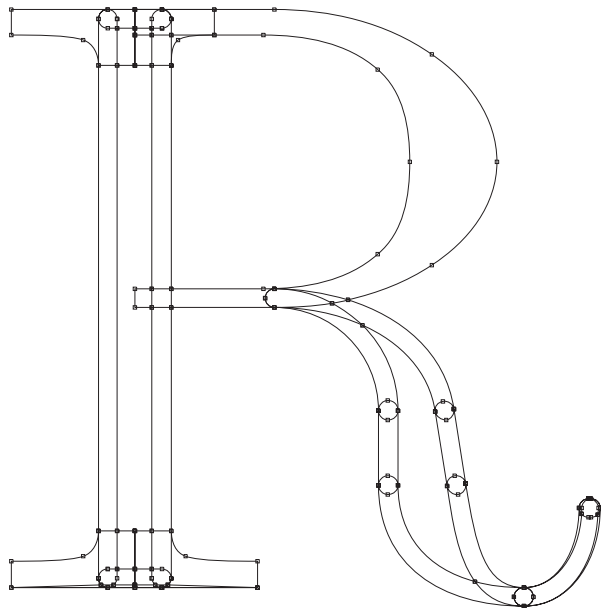


Figure 10: MetaFog intermediate step (overlaid), showing both explicit METAFONT shapes and Bézier envelopes of circular pen strokes.

We created figures like the Bézier curve and ellipse examples using Corel Draw’s drawing tools; MetaFog-output figures by importing MetaFog’s PostScript-like output into Corel Draw; and screen snapshots by capturing with Corel Capture and pasting into Corel Draw via the Windows clipboard. We used the figures in Corel Draw to export Encapsulated PostScript (EPSF) files and inserted the files as \TeX figures using the `epsfig` package for \LaTeX . The TRUETEX `dvips`-compatible special-handlers allowed both screen previews and printing of the EPSF figures, including printing on a non-PostScript laser printer. We used Corel Draw to print overhead transparencies of the figures. Draft copies and transparencies were imaged on an HP 4M Plus laser printer. The Proceedings editors use \LaTeX and `dvips`, so that no conversions were necessary between the author’s submission and the final production.

References

- Adobe Systems. *Adobe Type 1 Font Format, version 1.1*. Addison Wesley, 1990.
- L. Carr. “Of METAFONT and PostScript”. *TeXniques* 5, \TeX Users Group, 1987.
- A. Glassner, editor. *Graphics Gems*. Academic Press, Cambridge, MA, 1990.
- D. Henderson. “Outline fonts with METAFONT”. *TUGboat* 10(1), 36–38, 1989.

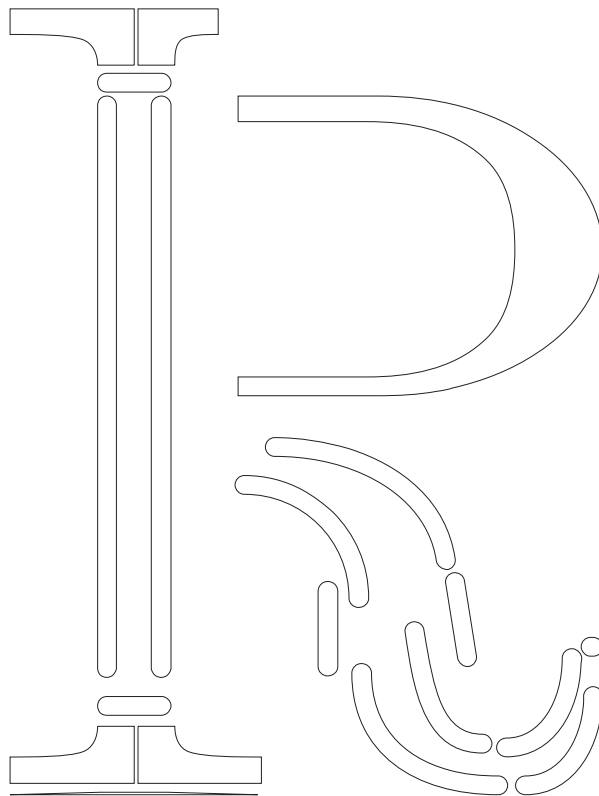


Figure 11: MetaFog intermediate step (exploded). The “dish” at the bottom is a white shape which subtracts from the serifs.

- J. D. Hobby. “A METAFONT-like system with PostScript output”. *TUGboat* 10(4), 505–512, 1989.
- B. Malyshev. “Automatic conversion of METAFONT fonts to Type1 PostScript”. *TUGboat* 15(3), 200–200, 1994.
- S. Yanai and Berry, Daniel M. “Environment for translating METAFONT to PostScript”. *TUGboat* 11(4), 525–541, 1990.

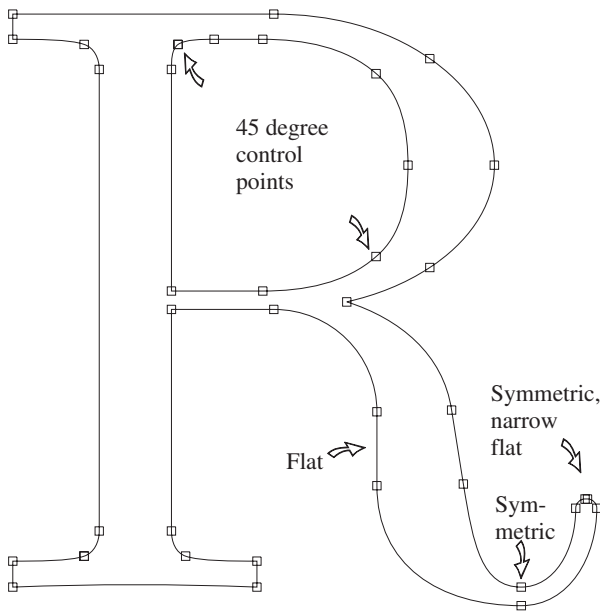


Figure 12: TRUEType conversion via MetaFog.

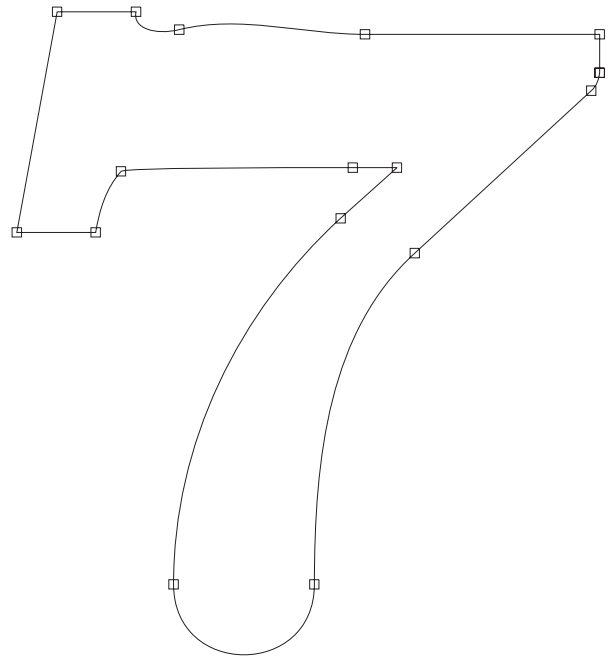


Figure 14: Error in CM digit 7 (cmbx5)

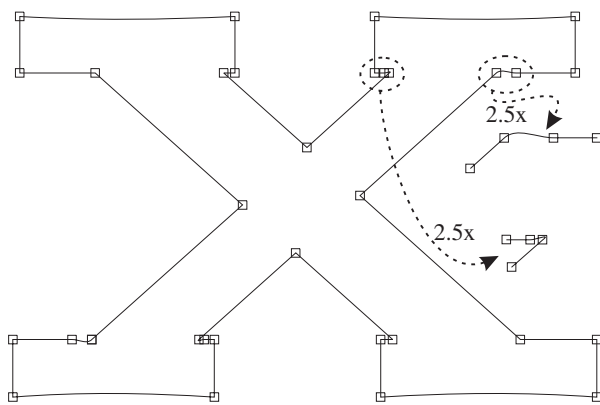


Figure 13: Error in CM serifs (cmbx5)

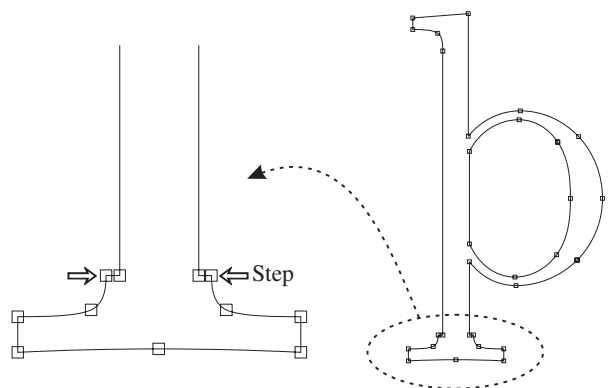


Figure 15: Error in DC thorn (dcr10)

The Poetica Family: Fancy Fonts with T_EX and L^AT_EX

Alan Hoenig
17 Bay Avenue
Huntington NY 11743
USA
Email: ajhjj@cunyvm.cuny.edu

Abstract

The Adobe Poetica family comprises fonts which have widely varying degrees of ornamentation, but which are designed to be used together. A document that employs them must change fonts regularly to provide appropriate ornamentation in different parts of words. Techniques for using Poetica fonts in (L^A)T_EX are presented, using a macro package written by the author, and metrics for the fonts derived using the `fontinst` package.

In the beginning of the ‘desktop publishing era’, digital foundries concentrated on making all old favorite fonts available in digital form. When that task was well along, they then began to turn their attention to enhancements to these fonts, the first group of which were expert fonts which contained things like small capitals and all double-f ligatures. Now that an impressive array of expert fonts has become available, a third wave may be under way — the development of beautiful fonts which break out of common font schemes altogether. A prime example is the Poetica family of fonts from Adobe. (Another is the Mantinia faces from the Carter and Cone foundry.) The purpose of this presentation is to suggest ways of typesetting with these beautiful fonts without walking around with numerous font tables in hand.

The Poetica Package

As delivered from Adobe Systems, the Poetica fonts comprise 21 fonts in two families. The main font is plainly modeled after the Chancery scripts of the Renaissance:

*Sonata Number 29 in B flat major,
opus 106,
Hammerklavier*

Why so many fonts? There is a wide variety of plain and fancy variants for many glyphs, odd ligatures, and special forms word endings or beginnings. One font contains ornaments, and another contains nothing but different ampersands. Using some of these fancy characters, the above phrase might appear

*Sonata Number 29
in B flat major,
opus 106,
Hammerklavier*

How can we use T_EX and L^AT_EX to typeset with these fonts, using convenient input conventions?

Yannis Haralambous and John Plaice (1994) have already demonstrated that the best way is to use a T_EX that handles 16-bit input. In that way, all the raw Adobe fonts can become part of a single, huge font associated with which would be a complex system of ligature rules to automatically select various glyphs in appropriate situations. Their Omega system is one such 16-bit system, currently freely available, which will handle such fonts, and which will handle a Poetica super-font. However, the fonts are so attractive that many people will not want to wait to implement Omega, and so I present the following discussion. Alternatively, the clumsiness of the arrangements I suggest may convince authors of the need to upgrade to Omega right away!

The Raw Package: A Closer Look. The main package consists of four main chancery fonts. Each has similar lowercase alphabets, but the uppercase characters are increasingly fancy. An expert font contains some ligatures, superior and inferior figures, and miscellaneous glyphs. A small caps font and an alternate SC font contains upright capitals that match the Chancery fonts.

Chancery I *Abc Def GHI Jklmn*
 Chancery II *Abc Def GHI Jklmn*
 Chancery III *Abc Def GHI Jklmn*
 Chancery IV *Abc Def GHI Jklmn*
 Expert *fffi fl fi fl*
 Small Caps *ABCDEFGHIJKLMN*
 SC Alternates *ABCDEFGHIJKLN*

ff fff fff fi fl ffi ffl
À Á Â Ã Ä Å
ß Œ Š Š Š

Finally, separate fonts contain batches of ornaments and bunches of ampersands.

** † ‡ § ¶ · ¸ ¹ º » ¼ ½ ¾*
& & & & & & & &

The second Poetica family is more interesting. There are four swash caps fonts, each of which contains two increasingly fancy uppercase alphabets.

A B C D E A B C D E
A B C D E A B C D E
A B C D E A B C D E
A B C D E A B C D E

An initial swash font contains one very fancy uppercase font, appropriate only for word beginnings.

A B C D E F G H I J

There are two lowercase alternate fonts, and each of these contains several groups of alternate forms for many lowercase characters.

g k p y z
g k p y z
g k p y z
g k p y z
g k p y z

There are two each of lowercase beginnings and endings fonts. Each of these fonts contains at least two forms (of certain letters only) appropriate for word boundaries.

baa end hah ill ton
baa end hah ill ton
baa end hah ill ton
baa end hah ill ton

A special ligature font contains fancy forms of the familiar f-ligatures, as well as many more ligatures not normally used.

It's clear that there are many ways to represent a single character, depending on its location in a word, whether it's upper- or lowercase, and the degree of swash that an author desires. The first example also makes clear that a little swash goes a long way, and a convenient font selection scheme would make it convenient to typeset in some single 'background' font from which it would be easy to ascend or descend to fancier or plainer fonts for isolated characters. We'd also like to be able to do this without having to lug around sheaves of font tables with us.

The next section details the font scheme that I propose for these fonts. I will then show how to use TeX's virtual font mechanism to create these fonts.

Fonts, Fonts, Fonts

Poetica contains a total of sixteen — 16! — uppercase alphabets. These include:

- four alphabets which match the four original chancery fonts;
- eight increasingly fancy swash alphabets (these appear in four fonts so that each font contains a pair of uppercase alphabets, one of which is in the lowercase position);
- a super-fancy swash alphabet suitable only for initial letters (if then);
- two small caps alphabets; and
- a small caps alternate alphabet (although this is a sparse set — only 15 letters are represented).

I felt able to organize these in twelve fonts:

- four Chancery fonts;
- four swash fonts, each incorporating two uppercase alphabets;
- one super-swash font;
- two small caps fonts; and
- two titling fonts.

This is still a formidable array of fonts, and I'll say more later on about ways of dealing with them all. But at this point, I'll indicate that I shoehorned

two swash uppercase alphabets into each font by virtue of T_EX's ligature mechanism. Most of the time, uppercase glyphs appear only at the beginning of a word, so I created the fonts so that * followed by a capital letter generates the alternate capital. For example, if I type

A B C D E

I might get

A B C D E

but if I type

*A *B *C *D *E

in the same font, I get

A B C D E

instead.

The uppercase alphabets dictate the nature of their fonts. They quite clearly become increasingly fancy, so it makes sense to apportion some of the other special characters to these fonts in order of increasing fanciness. It's straightforward via Alan Jeffrey's `fontinst` package to add these characters to the fonts.

Word Boundaries. Many of the characters provided by Adobe belong specially to the beginnings or endings of words, and the `boundarychar` mechanism of T_EX3 makes this easy to implement, but not as easy as I expected for the following interesting reason.

Human readers are quite specific in what constitutes a word boundary. Most often it would be a space or punctuation, but T_EX3 is more restrictive: essentially any consecutive string of characters is a word. This means that `te\it st` is two words from T_EX's point of view—that is, a font change in the middle of a word creates two word boundaries. Typesetting with fonts containing fancy word boundary glyphs requires dealing with this fact.

Here is an example of automatic boundary glyph selection. Notice here how the forms of the 'm' and 't' change depending on their positions within a word. With the proper fonts selected, I simply typed `mat tom-tom` to get

mat tom-tom

Supporting Macros. These fonts are beautiful, but I needed some input conventions that would allow me to increase or decrease the amount of fanciness in some easy way.

I began by appropriating from mathematics the characters `^`, `_`, `+`, and `-`; this is no loss, as I felt it

unlikely that I'd be doing math in conjunction with Poetica. (However, some of the Chancery capitals do make a good candidate as a math calligraphic alphabet. That is a different and easier problem.)

Typesetting is done within the Poetica environment:

```
\begin{Poetica}
...
\end{Poetica}
```

(I am assuming the conventions of L^AT_EX 2_ε, so I have access to the New Font Selection Scheme) which automatically switches to the Poetica family. All the fonts are in the medium series `m`, selected automatically, and the fonts themselves are divided into three groups of shapes. (Adobe provides no bold face fonts in this family.) 'Normal' fonts comprise four fonts, with font shapes of `n0`, `n1`, `n2`, and `n3`. Two groups of five swash fonts apiece, with shape designations `f0` through `f4` and `F0` through `F4` (`f` or `F`=fancy) encompass the ornate fonts I set up. The `F`-shapes incorporate word boundary glyphs, while `f`-shapes do not. The higher the number, the fancier the font. There is also a small caps font (shapes `c` and `c1`) and two titling fonts (shapes `t` and `t1`). The default font has the shape `n3` at an eighteen-on-twenty-two point size:

Alpha-Betic Constants
Demand Emphasis.
 12345 67890

Although these fonts can be accessed by the usual NFSS commands, the usual `\fontshape` and `\selectfont` commands are discouraged in favor of a single `\Fontshape` command which combines `\fontshape` and `\selectfont` together with some bookkeeping, the reason for which will shortly become clear. It will be necessary to do any font sizing with `\fontsize` (in the usual way) before calling `\Fontshape`.

But even `\Fontshape` is too verbose. Most of the time, we are content to typeset virtually everything in a piece of text in the same font, except from time to time we may want to make one or two characters more or less fancy than the default. Although the usual font changing could be invoked, it's a bit messy to do that for a single character here and there. I implemented a scheme which seemed to me ideal from the point of view of making these spot changes, and for that reason different meanings were assigned to `^`, `_`, `|`, `\+`, and `\-`. The control sequences `\+` and `\-` take the next character (or group) and raise the level of fanciness up or down by one

font. If that is not sufficient, simply add additional +s and -s. The symbols ^ and _ now mean go up to the fanciest and plainest fonts respectively (that is, shapes n0 and f4),

but the + and - convention also holds here. The vertical bar is now equivalent to the `\noboundary` command. There are also two additional commands: `\wordbounds` and `\nowordbounds`, which select the F-shape or f-shape fonts respectively.

Thus, if we type

```
\begin{Poetica}
For every action there is a reaction
\end{Poetica}
```

we get

For every Action there is a Reaction

But if we type

```
\begin{Poetica}
^For every \+A^{ct}{\i}on\ \
^there i^s a
^---{*R}ea^{---ct}{\i}o^n
\end{Poetica}
```

we get instead

For every Action there is a Reaction

Actually, the markup here is almost as intrusive as normal T_EX markup would be, but normal Poetica markup would not be this excessive. I had great fun generating this sample, adding and subtracting +s and -s until there was sufficient demonstration of these conventions as well as a demonstration of several different glyphs. Note the several ct ligatures, and other alternate letterforms.

Let's see why special treatment of word bounds is necessary. If we re-typeset this example with `\wordbounds` in effect, we get

For every Action there is a Reaction

Notice the unfortunate appearance of certain boundary glyphs in the middle of real words due to the word boundaries formed every time there is a font shift.

Incidentally, to get an idea of the possibilities of swash I first typed

```
\begin{Poetica}
^For every action
^there is a Reaction}
\end{Poetica}
```

to get

For every action there is a Reaction

Let me include two more examples. If we type

```
\newsavebox{\mybox}
\newlength{\mywd}\newlength{\myht}
\newlength{\mydp}
\setlength{\fboxrule}{1.2pt}
\savebox{\mybox}
{\fbox{\begin{minipage}{.5\textwidth}
\begin{center}\begin{Poetica}
\fontsize{26}{34}\selectfont
^---{*A}nd if \+{y}ou wi^{ll} con-\
sider a\++{ll} t\++h{\i}ngs,
{\wordbounds\++y}ou\
will find that ^{th}ose\
whi^{ch} are ^goo^{---d}an^d\
use\++ful a\++lways ^have\
^{th}e grace of beaut^y\
in ^{th}em as we^{---ll}.\
{\renewcommand\.{\hspace{1.8pt}}}%
\fontshape{t1}\fontsize{16}{24}
\selectfont
c\.\a\.\s\.\t\.\i\.\g\.\l\.\i\%
\.\o\.\n\.\e}\}[1pc]
\fontsize{30}{38}\orn{78}
\end{Poetica}\end{center}
\end{minipage}}
\settowidth{\mywd}{\usebox{\mybox}}
\settoheight{\myht}{\usebox{\mybox}}
\settodepth{\mydp}{\usebox{\mybox}}
\noindent\rlap{\vrule width1.1\mywd
height1.1\myht depth1.1\mydp}%
\hskip.05\mywd%
{\White{\usebox{\mybox}}}
```

we get figure 1. Notice that the titling fonts have to be accessed explicitly (ditto for the small caps fonts); they are not part of the bump up scheme elsewhere in use. We selected a special ornament via the `\orn` command; there is a corresponding `\amp` command to select ampersands for the special ampersand font. I guess you will need to have access to the ornament and ampersand font tables to know which characters to choose. Since I am using Tom Rokicki's `dvips` post-processor, I use the `colorvdi` package. I get the fancy effect here by setting a big, black rule box, and overprinting the text in `\White` ink. Actually, only the indented material sets type; the remaining lines set things up to print white on black.

And a final example. To get figure 2, I typed

```
\newdimen\W
\newcommand{\dropcap}[1]{\setbox0=
\hbox{\fontsize{44}{48}
\selectfont#1\ }%
```

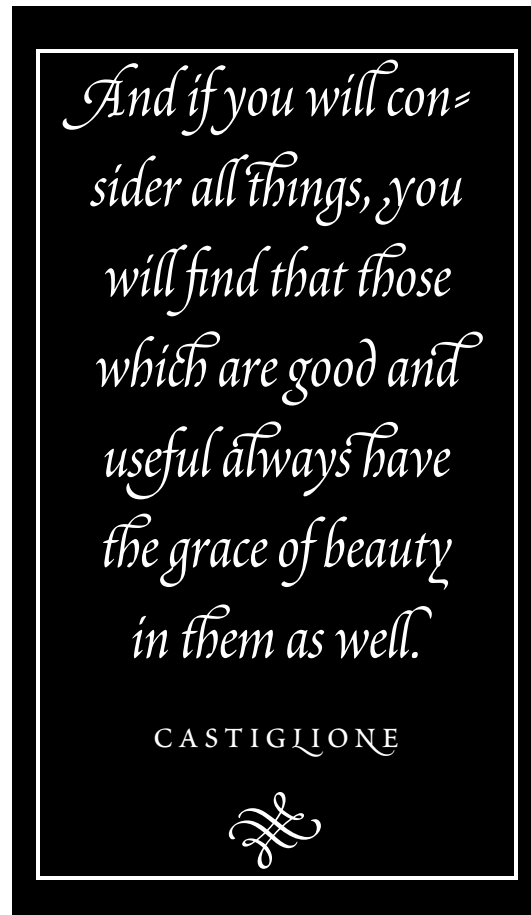


Figure 1: One example.

```

\setbox0=\hbox to.5\wd0{\hss\box0}%
\W=\wd0 \gdef\.{\noindent\hskip\W}
\noindent\vbox to10pt{\box0 \vss}
\begin{center}
\begin{Poetica}\fontsize{23}{31}
\selectfont
\fontsize{30}{32}\selectfont
_{A Sonne}\{\wordbounds^--{|t}}
\[\3pt]
{\fontsize{14}{21}\fontshape{c}
\selectfont
William Shakespeare}
\end{Poetica}\end{center}
\begin{verse}\begin{Poetica}
\dropcap{^W}hen, in disgrace with
\+{*F}ortune and men's eye^{s},\
\I a^--{\ll} alone beweeep my
outca\+{st} \ ^{st}ate,\
And trouble deaf heaven with my
bootle^--{ss} cries,\
\+And look upon myself, and
curse my fate,\
Wishing me like to one more
ri^{ch} in ^-{\h}ope,\
Featured like him, like him
with ^{f}riends posse^--{st},\
Desiring this man's ar^---{t}
and that man's sco^---{p}e,\
\+With what I most enjoy
contented least;\
Yet in these thoughts myself
almost de^--{sp}{\i}sing^---\
Haply I think on ^{th}ee:
and then my \+{st}ate,\
Like to the \+Lark at break
of day arisin{\wordbounds^g}\
\+From su^--{\ll}en earth,
sings hymns at \+Heaven's
gate;\[6pt]
\quad ^For ^{th}y sweet \ \ \
{\wordbounds^--l}ove rememb'red
suc^--{\h} weal^{th} brings\
\quad ^{Th}at t^--{\h}en \+{I}
scorn to ^{ch}ange my ^-{\st}ate
with ^-{*K}ings.\
\end{Poetica}
\end{verse}

```

The Poetica macro file, very short, appears in Appendix A.

Creating Poetica Virtual Fonts. The tool of choice for creating virtual fonts is the Alan Jeffrey's `fontinst` package. In the presence of ASCII files containing font information, running the installation

file through T_EX creates the `.vpl` files from which the actual `.vf` virtual fonts are rendered. The font information should be in three types of files.

1. Metric files — files giving information about the sizes and kernings of each glyph. typically, these are files with extensions `.pl` (T_EX fonts) or `.afm` (type 1 outline fonts). The `fontinst` package reads these files and creates its own metric `.mtx` files. Other metric information needs to be supplied in additional `.mtx` files using commands following the standard `fontinst` syntax.
2. Encoding files, which say how the glyphs should be arranged in the font. In addition to this *encoding* information, ligature information is also found here.
3. Miscellaneous additional files, usually metric in nature.

The `fontinst` installation file has the following structure.

```

\input fontinst.sty
\installfonts
\installfamily{OT1}{poet}{}
\installfont{pof3}{pos10,unpos13,
poslaii0,poslai0,
unlai1,
possciv0,unsc,
setfont1,
pociii0,latinpoet}
{OT1swa}%
{OT1}{poet}{m}{f3}{}
...
...
\endinstallfonts
\bye

```

The lines of ellipses represent the (many) additional `\installfont` commands not shown here. These instructions provide for a font family called `poet` which uses OT1 (original T_EX) encoding. One font in that family is called `pof3` and corresponds to medium series `m` and font shape `f3` within this family.

Of the parameters of the `\installfont` command, the second and third present a list of metric and encoding files that `fontinst` will need to construct the virtual fonts. The cryptic nature of these file names is imposed, as is so often the case, by the 8 + 3 file name structure of MS-DOS. Here's a brief description of these files.

- All file names `po*` are of `.afm` files containing metric information about one of the 21 Poetica raw fonts. The names `pos10`, `poslai0`, `poslaii0`, `poslbii0`, `posleii0`, `poslei0`, `possciv0`, and `pociii0` belong to the ligature,

the second lowercase alternates, the first lowercase alternates, the second lowercase beginning letters, the second lowercase endings, the first lowercase endings, the fourth supplementary swash caps, and Chancery font number 3.

- Because the order in which the information is read by `fontinst`, we need some mechanism for removing superfluous information from its memory. The `un*.mtx` files perform this function. See below for an extensive description of this process. Remember that the `fontinst` macros have been crafted so that information once read is not over-written by later information.

Crafting a Font

Let's consider in greater detail the construction of the font we called `pof3`. The first thing `fontinst` does is read the glyph information pertaining to the ligatures (`pos10`), whose glyphs have been named according to the standard Adobe encoding vector. For example, position 65—A—of the ligature raw font `pos10` is occupied by the ligature 'Ch' and is called 'A' in the `.afm` file.

This is bad, and a violation of Adobe's own standards! Such a glyph should, by rights, be named 'Ch' since that's what the letterform looks like and not 'A'. If something is not done, the real 'A' (in `possiv0`) will never be typeset and every 'A' in the source document will appear as 'Ch' in the typeset output. (Do you see why? `fontinst` pays attention only to the first definition of a letterform. Since the default nomenclature creates an 'A' out of a C-h ligature, this becomes the definition of 'A' and a proper definition of A later on will be ignored.) Therefore, we must read a file `unpos13.mtx` which saves the ligature information under a more meaningful name and frees up the 'A' slot for the real glyph. Two lines of this file might read

```
\resetligglyph Ch A
\unsetglyph A
```

where we have previously entered the definition

```
\setcommand\resetligglyph#1#2#3{
  \setleftkerning{#1#2}{#1}{1000}
  \setrightkerning{#1#2}{#2}{1000}
  \resetglyph{#1#2}
  \glyph{#3}{1000}
  \endresetglyph
}
```

In this case, a new glyph called 'Ch' is defined to be equivalent to 'A'. This new glyph is also given some appropriate kerning information. Once that

definition has been fixed, the 'A' glyph has been made free for later use.

This is the philosophy behind the next several files. Special alternate forms, beginning forms, and ending forms are carefully ingested, and various `un*` files save glyphs under more appropriate names and free up poorly named slots. Finally, the uppercase letters are taken from a swash font, some of the alternate characters are declared to be equivalent to letter glyphs (in file `setfont1.mtx`), the remaining lowercase and other characters are taken from the Chancery 3 font `pociii0`, and `latinpoe.mtx` lists the characters in the font. This concludes the metric portion.

Anything not involving measurement is by definition the province of an encoding file. The file lays out the order of glyphs in a font—the encoding vector—and arranges for ligature formation. For example,

```
\setslot{c}
  \Ligature{h}{ch}
  \Ligature{k}{ck}
  \Ligature{l}{cl}
  \Ligature{t}{ct}
  \atendofword{cend}
\endsetslot
```

arranges things with \TeX so that 'c' followed by 'h' are replaced by the special 'ch' ligature if the 'ch' glyph exists. If a 'c' appears at the end of a word, it is replaced by a special final c glyph, here called 'cend'. Special forms for the beginning of a word are set up by

```
\setslot{boundarychar}
  \atstartofword{b}{bbeg}
  \atstartofword{e}{ebeg}
  ...
  \atstartofword{w}{wbeg}
  \atstartofword{y}{ybeg}
\endsetslot
```

We use the `fontinst` definitions

```
\setcommand\Ligature#1#2{% cond'l lig
  \ifisglyph{#2}\then
  \ligature{LIG}{#1}{#2}\fi}
\setcommand\atendofword#1{%
  \Ligature{boundarychar}{#1}}
\setcommand\atstartofword#1#2{%
  \Ligature{#1}{#2}}
```

to control these special ligatures.

The `fontinst` files are to be posted on CTAN for anonymous file transfer.

A Sonnet

WILLIAM SHAKESPEARE

When, in disgrace with Fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself, and curse my fate,
Wishing me like to one more rich in hope,
Featured like him, like him with friends possess'd,
Desiring this man's art and that man's scope,
With what I most enjoy contented least;
Yet in these thoughts myself almost despising—
Haply I think on thee: and then my state,
Like to the Lark at break of day arising'
From sullen earth, sings hymns at Heaven's gate;

For thy sweet love rememb' red such wealth brings
That then I scorn to change my state with Kings.

Figure 2: A second example.

The Poetica Macros

```

%% Package File poetica.sty
\newcount\poetic
\newcount\poetbound \poetbound=4
\newcount\poetceiling \poetceiling=8
\newcount\poetfloor \poetfloor=0

\def\wordbounds{\def\fancyshape{F}}
\def\nowordbounds{\def\fancyshape{f}}
  \nowordbounds % default

\def\parsefontshape#1#2{\poetic=-1
  \if f#1\poetic=\poetbound
    \advance\poetic by#2 \fi
  \if n#1\poetic=\poetfloor
    \advance\poetic by#2 \fi
}
\def\setshape{% input is \poetic
  \ifnum\poetic<0 \else
    \ifnum\poetic<\poetbound
      \edef\fshape{n\the\poetic}%
    \else\advance\poetic by
      -\poetbound
      \edef\fshape{\fancyshape
        \the\poetic}%
    \fi
  \fi
}
\newcommand{\Fontshape}[1]{%
  \parsefontshape#1%
  \fontshape{#1}\selectfont}
\newenvironment{Poetica}{%
  \begingroup\fontencoding{OT1}
  \fontfamily{poet}\fontsize{18}{22}
  \fontseries{m}\Fontshape{n3}
  \poetic=3 \setshape{\endgroup}
\let\dhyph=\- \let\mytabs=\+
\let\oldhat=\^ \let\oldsub=_
  \let\oldvert=|
\catcode'\^ \active \catcode'\_ \active
\catcode'\| \active \def\|{\oldvert}
  \let|= \noboundary
\newcount\INC \INC 1
\def^{\bgroup \let\compare=-
  \let\bump=\bumpdown \INC-1
  \poetic=\poetceiling
  \afterassignment\getnextchar
  \global\let\nexttok= }
\def_{\bgroup \let\compare=+
  \let\bump=\bumpup \INC 1
  \poetic=\poetfloor
  \afterassignment\getnextchar
  \global\let\nexttok= }

```

```

\def-\{\bgroup \let\compare=-
  \let\bump=\bumpdown \INC-1 \bump
  \afterassignment\getnextchar
  \global\let\nexttok= }
\def+\{\bgroup \let\compare=+
  \let\bump=\bumpup \INC 1 \bump
  \afterassignment\getnextchar
  \global\let\nexttok= }

\def\getnextchar{%
  \if\compare\nexttok
    \bump \let\nextact\grabchar
  \else
    \edef\nextact{%
      \noexpand\typeset
      \noexpand\nexttok}%
  \fi \nextact
}
\def\grabchar{\afterassignment
  \getnextchar \let\nexttok}
\def\bumpdown{\advance\poetic \INC
  \ifnum\poetic<\poetfloor
  \poetic\poetfloor \fi}
\def\bumpup{\advance\poetic \INC
  \ifnum\poetic>\poetceiling
  \poetic\poetceiling \fi}
\def\typeset#1{\setshape\fontshape
  {\fshape}
  \selectfont #1\egroup}

%% ornaments and ampersands
\newcommand{\orn}[1]{%
  {\fontshape{orn}\selectfont
  \symbol{#1}}}
\newcommand{\amp}[1]{%
  {\fontshape{amp}\selectfont
  \symbol{#1}}}

\endinput

```

References

- Y. Haralambous and Plaice, John. "First applications of Ω : Adobe Poetica, Arabic, Greek, Khmer". *TUGboat* 15(3), 344–352, 1994.

Using Adobe Type 1 Multiple Master Fonts with T_EX

Michel Goossens

CN Division, CERN
CH-1211 Geneva 23
Switzerland
Email: m.goossens@cern.ch

Sebastian Rahtz

Elsevier Science Ltd
The Boulevard, Langford Lane
Kidlington, Oxford OX5 1GB
UK
Email: s.rahtz@elsevier.co.uk

Robin Fairbairns

University of Cambridge Computer Laboratory
Pembroke St, Cambridge CB2 3QG
UK
Email: rf@cl.cam.ac.uk

Abstract

Adobe's Multiple Master font format has some of the properties that METAFONT pioneered to express many fonts of the same family from the same (set of) sources. The advent of multiple master fonts could offer a significantly better choice of fonts to users of T_EX; however, there are problems integrating them with T_EX, and the paper presents a first solution to those problems.

The paper is derived (by Robin Fairbairns) from an article written by Michel Goossens and Sebastian Rahtz for the UKTUG magazine Baskerville volume 5, number 3. As a demonstration of the effectiveness of the techniques described, it is being typeset using Adobe Minion and Myriad multiple master fonts.

Introduction

The multiple master font format is an extension of the Type 1 font format, which allows the generation of a wide variety of typeface styles from a single font program. This capability allows users and applications control over the typographic parameters of fonts used in their documents, in a manner reminiscent of Knuth's ground-breaking METAFONT. This article describes the multiple master system in some detail, and describes the procedures needed to make instances, and create the appropriate font metrics for use with T_EX.

Multiple Master overview

A multiple master font program contains two or more outline typefaces called *master designs*, which describe one or more *design axes*. The master designs that constitute a design axis represent a dynamic range of one typographic parameter, such as the weight or width. This range of styles is defined in a multiple master font program by specifying one master design to represent each end of an axis, such as a *light* and

extra-bold weight, as well as any *intermediate master designs* that are required. The maximum number of master designs allowed is sixteen.

A *font instance* consists of a font dictionary derived from the multiple master font program (or from another font instance). It contains a `WeightVector` array with k values that sum to 1.0 and which determine the relative contributions of each master design to the resulting interpolated design.

All derived font instances share the `CharStrings` dictionary and `Subrs` array of the main multiple master font program, making it relatively economical to generate a variety of font instances. Multiple master fonts can be made compatible with the installed base of PostScript language interpreters by including several PostScript language procedures and a new set of `OtherSubrs` routines in the font program. The procedures include the new `makeblendedfont` operator, the interpolation procedure `$Blend` and a new definition of the `findfont` operator.

Multiple Master Design Space It is possible to think of the master designs as being arranged in a 1, 2, 3, or 4 dimensional space with various font instances corresponding to different locations in that space. The entries in the `FontInfo` dictionary specify what this space is and where the master designs are located in it. This information is necessary for interactive programs that allow users to create new font instances, and should be included in the font's Adobe Multiple Font Metrics (AMFM) file.

Figure 1 illustrates an example of the design space of a three axis multiple master font. In this example, the axes are *weight*, *width*, and *optical* size. It is recommended that a font program be organized to have the lightest weight, narrowest width, and smallest design size mapped to the origin of the blend space.

Multiple master coordinates are of two types: those which represent the design space and those which represent the blend space. *Design coordinates* are integers whose range for a particular typeface is chosen by the designer. They are used in font names and in the user interface for software which creates and manipulates multiple master font programs. The theoretical range for a weight or width axis is from 1 to 999 design units; however a typical typeface, with styles ranging from light to black, might have a dynamic range of from 200 (for light) to 800 units (for black).

Another type of optional axis would be for optical size, in which the character design changes with the point size. The design coordinates for the optical size axis might have a dynamic range of from 6- to 72-point, which represents the practical extremes of sizes for typefaces designed for publishing purposes.

Blend coordinates are normalized values, in the range of 0 to 1, which correspond to the minimum and maximum design space coordinates. They are used by the Type 1 rasterizer because they are more convenient for mathematical manipulations. The linear space of blend coordinates is related to the (potentially) non-linear space of the design coordinates by the `BlendDesignMap` entry in the font dictionary.

A four axis design might also be considered; an example of a fourth axis would be having an axis for a typographic style (serif/sans serif) or contrast (high/low: the ratio of thick to thin stem widths). If four axes are defined, sixteen master designs are required. Also, since sixteen is the maximum number of designs allowed, there can be no intermediate designs with four axes.

Multiple Master Font Programs

Multiple master typefaces may contain from two to sixteen master designs, organized as having from one to four design axes. Since the maximum number of master designs allowed in a multiple master font is 16, the number x of intermediate masters is subject to the restriction $2^n + x \leq 16$, where n is the number of design axes.

The values used for calculating the weighted average are stored in the font dictionary in an array named `WeightVector`. The multiple master font program, as shipped by the font vendor, can have a default setting for the `WeightVector`; it is recommended that it is set so the default font instance will be the normal roman design for that typeface.

Multiple Master Keywords `BlendAxisTypes` is a (required) array of n PostScript language strings where n is the dimension of the design space and hence the number of axes. Each string specifies the corresponding axis type. In the case of the Minion 3-axis example, this value would be:

```
/BlendAxisTypes [/Weight /Width /OpticalSize]
```

`BlendDesignPositions` is a (required) array of k arrays giving the locations of the k master designs in the design space. Each location subarray has n numbers giving the location of the design in the n dimensions of the blend space, with a minimum value of zero and a maximum value of one. Table 1 with eight master designs is based on the example shown in Figure 1. It corresponds to the blend space of a 3-axis multiple master font like Minion.

For the MinionMMfont, the `BlendDesignPositions` array becomes:

```
/BlendDesignPositions
[[0 0 0][1 0 0][0 1 0][1 1 0]
[0 0 1][1 0 1][0 1 1][1 1 1]] def
```

`BlendDesignMap` is a required entry consisting of an array of n arrays where n is the dimension of the design space. Each array contains m subarrays that describe the mapping of design coordinates into normalized coordinates for that design axis. The minimum value allowed for m is two, and the maximum is twelve. The order of the subarrays corresponds to the order of design axes in `BlendAxisTypes`. In the case of the Minion font this array is three dimensional ($n = 3$) and has the following form:

```
/BlendDesignMap [
[[345 0] [620 1]] [[450 0] [600 1]]
[[6 0] [8 0.35] [11 0.5] [18 0.75] [72 1]] ]
```

The first number in an individual subarray is in design coordinates with a minimum value of 1 and a maximum value of 999. The second number in the subarray is in normalized coordinates, that is, in the range of 0 to 1. In the above example, the weight ranges from 345 to 620, while the width ranges from 450 to 600 in design space. The third axis, optical size, ranges from 6 to 72 (corresponding to the point sizes for which the typeface can be adjusted for optimal legibility).

The `makeblendedfont` Operator

```
blendedfontdict weightvector makeblendedfont blendedfontdict
```

This operator creates a font dictionary with blended entries. The *blendedfontdict* argument is a font dictionary of an existing multiple master font; it can be from either

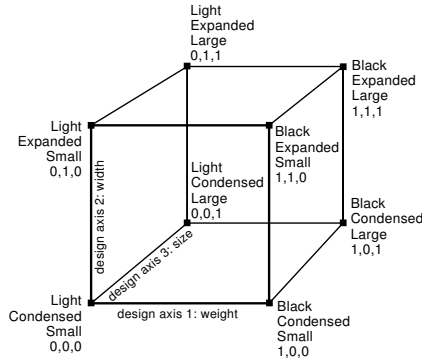


Figure 1: Multiple master typeface blend space arrangement

the original multiple master font itself, or from an interpolated font instance since any Blend dictionary contains all elements needed to derive additional font instances.

The *weightvector* argument is an array of numbers summing to 1.0 to be used as the weights for creating the new font instance. The value of *WeightVector* in *blendedfontdict* is set to the values in the array *weightvector*. Blended values are calculated for entries in the *Private* and *FontInfo* dictionaries. The result is a font dictionary that can be used as an argument to *definefont*. The resulting dictionary and its contents are still read-write, so the caller of *makeblendedfont* can make further modifications if necessary.

The Blend dictionary data structures provide the information needed by the *makeblendedfont* operator, without needing to have the specific list of entries to be blended built into the procedure. This allows a single copy of the procedure to be used even if the set of entries to be blended varies in future fonts.

Multiple Master findfont Procedure Multiple master font programs from Adobe include a procedure which redefines the *findfont* operator in *systemdict*. This is necessary because of the need to generate font instances on-the-fly to satisfy multiple master font references in a PostScript language document. The procedure creates all necessary font instances before calling the standard *findfont* procedure.

Two procedures, *NormalizeDesignVector* and *ConvertDesignVector*, which are referenced in *findfont*, must be configured for the number of axes and master designs in the font program in which they are used. The *NormalizeDesignVector* procedure must calculate the normalized equivalent of the design coordinates in the *FontName*, using the values in the *BlendDesignMap* array. These normalized coordinates must be left on the stack for the *ConvertDesignVector* procedure. This procedure should take the normalized coordinates, generate

<i>Design label</i>	<i>Blend space coordinates</i>
design 1: light condensed small	000
design 2: black condensed small	100
design 3: light expanded small	010
design 4: black expanded small	110
design 5: light condensed large	001
design 6: black condensed large	101
design 7: light expanded large	011
design 8: black expanded large	111

Table 1: Design labels and blend space values for the Minion 3-axis multiple master font

WeightVector values, and leave them on the stack for the *makeblendedfont* operator.

Using Multiple master fonts with \TeX

Multiple master fonts come with a set of multiple master AFM files, which are called “AMFM” (Adobe Master Font Metrics) files. This file contains information about the number of master designs, the number of axes, the *BlendDesignPositions* and *BlendDesignMap* arrays, as well as the names, and *weightvector* for the master designs, from which all font instances are derived.

To get the actual metric information for the characters in a font instance, one has to combine the metric information of the master designs (eight, in the case of Minion). To do this one needs to calculate the *weightvector* for the given instance. Starting from design-coordinate space one can use the *NormalizeDesignVector* operator to transform to the normalized coordinate space, and from there with the *ConvertDesignVector* operator one obtains the *weightvector*. These two operators are particular to a font (since they depend on the master designs), and are present in the multiple master font dictionary. One can decode the PostScript code for calculating the *weightvector* and translate it into another computer language, and then use the procedure to combine the values in the AFM files for the master designs to calculate the values needed for the font instance. For instance, in the case of the MinionMM font, the PostScript code defines the eight components of the *weightvector* as follows:

$$\begin{aligned}
 w_1 &= xyz & w_2 &= (1-x)yz \\
 w_3 &= x(1-y)z & w_4 &= (1-x)(1-y)z \\
 w_5 &= xy(1-z) & w_6 &= (1-x)y(1-z) \\
 w_7 &= x(1-y)(1-z) & w_8 &= 1 - \sum_{n=1}^7 w_n
 \end{aligned}$$

where x is the normalized weight, y the normalized width, and z the normalized optical size.

These eight numbers w_i allow the calculation of all needed parameters in an AFM file for a font instance. One

reads each parameter value in turn in the eight master design AFM files, applies the relevant weight, and the weighted sum thus obtained is the desired interpolated value of the given parameter for the font instance.

Myriad is a sans serif companion font to Minion. It has two design axes and four master designs. The weights for deriving font-instance parameters in normalized coordinate space in function of the four master designs are given by:

$$\begin{aligned} w_1 &= (1-x)(1-y) & w_2 &= (1-x)y \\ w_3 &= x(1-y) & w_4 &= xy \end{aligned}$$

where x is the normalized weight and y the normalized width. The corresponding mapping parameters between design space and normalized coordinates are:

```
BlendDesignPositions [ [0 0] [1 0] [0 1] [1 1] ]
BlendDesignMap [[ [215 0] [830 1] ] [ [300 0] [700 1] ] ]
BlendAxisTypes [ /Weight /Width ]
```

Now one can extract any of the boundingbox and kern entries for a given font instance by getting the element in question from the eight (or four, in the case of Myriad) master files and calculating the interpolated value. To make matters simpler an explicit example will be given for the Myriad font, since it involves only four numbers in each case. Figure 5 shows some parts of the four master-design AFM files

When the instance AFM file has been created, a suitable metric for \TeX can be built with `afm2tfm` or the `fontinst` package.

In practice

We have instantiated the ideas outlined above by developing Unix shell scripts, and adapting an AFM-parsing program distributed by Adobe. The main script takes the following actions:

1. create a small PostScript file to invoke multiple master operators with values passed to the script;
2. run GhostScript on this file to derive normalized weights, and write them to a temporary file; note that this must be version 3.33 or later of Aladdin GhostScript — earlier versions of the program did not have the code to realize multiple master fonts;
3. run our “`mmafm`” program to read master AFM files, write a new instance AFM file, and create a \TeX metric (our initial setup uses `afm2tfm` to create 8r base-encoded metrics, and EC-encoded virtual fonts for actual use);
4. write a `dvips` map entry and header file to tell the driver about the new font.

Thus a call to our scripts consists of the parameters

```
MinionMM zmn18ac6 360 460 6
```

This creates a metric file called `zmn18ac6`, using Karl Berry’s scheme to name “Minion, light weight, 8a-encoded, condensed, at 6pt design size”. The entry in the map file reads

```
zmn18rc6 zmn18ac6 "TeXBase1Encoding \
ReEncodeFont" <8r.enc <MinionMM.pfb \
<zmn18ac6.pro
```

where the prologue file `zmn18ac6.pro` contains instructions to the PostScript interpreter as to how the given font instance should be generated from the multiple master font code in `MinionMM.pfb`: `zmn18ac6.pro` contains the code:

```
/zmn18ac6 /MinionMM findfont
dup begin [
  360 460 6   NormalizeDesignVector
  ConvertDesignVector
] end makeblendedfont definefont
```

Note the presence of the `NormalizeDesignVector`, `ConvertDesignVector` and `makeblendedfont` PostScript operators described earlier.

In addition, we hand-wrote “`fd`” files to tell \TeX how to match up the various weight and width instances we created to its notions of series and shape. The only complication here was that the Minion font has an optical size axis, and we built four instances which we wanted \TeX to use at different user sizes:

```
\DeclareFontShape{T1}{zmn}{lc}{n}{%
  <-7>zmn18tc6 %
  <7-10>zmn18tc8 %
  <10-15>zmn18tc11 %
  <15->zmn18tc18}
{}
```

The effect of the optical sizes is demonstrated by Figure 6 which shows the 6pt and 18pt instances scaled to the same size. The differences in design are as apparent as the corresponding examples from Computer Modern would be.

The tools we developed served to test the ideas, and build a set of metrics; they are available from us on request, but users should beware that they are neither intuitive in use, nor necessarily robust. It is to be hoped that a more functional, portable, solution will be developed in time. The keen \TeX ie may be interested in developing a `MakeTeXTFM` script for Unix `web2c` systems to apply the programs on the fly from within \TeX .

```
FontName MyriadMN-LightCn      FontName MyriadMN-BlackCn      FontName MyriadMN-LightSemiEx  FontName MyriadMN-BlackSemiEx
FamilyName Myriad MN          FamilyName Myriad MN          FamilyName Myriad MN          FamilyName Myriad MN
Weight Light                   Weight Black                   Weight Light                   Weight Black
ItalicAngle 0                  ItalicAngle 0                  ItalicAngle 0                  ItalicAngle 0
IsFixedPitch false            IsFixedPitch false            IsFixedPitch false            IsFixedPitch false
FontBBox -52 -250 970 818     FontBBox -64 -250 970 843     FontBBox -58 -250 1100 825     FontBBox -48 -250 1432 867
...
StartKernPairs 974            StartKernPairs 974            StartKernPairs 974            StartKernPairs 974
KPX A z 10                    KPX A z 10                    KPX A z 25                    KPX A z 7
KPX A y -31                   KPX A y -10                   KPX A y -10                   KPX A y -44
KPX A x 4                     KPX A x 0                     KPX A x 0                     KPX A x -6
KPX A w -36                   KPX A w -10                   KPX A w -10                   KPX A w -47
KPX A v -42                   KPX A v -10                   KPX A v -25                   KPX A v -62
KPX A u -9                    KPX A u 0                     KPX A u -10                   KPX A u -22
KPX A t -17                   KPX A t 0                     KPX A t 0                     KPX A t -32
KPX A s 0                     KPX A s 10                    KPX A s -10                   KPX A s -6
KPX A r -4                    KPX A r 0                     KPX A r 0                     KPX A r -10
KPX A quoteright -90         KPX A quoteright -20         KPX A quoteright -30         KPX A quoteright -90
KPX A quotedblright -90     KPX A quotedblright -20     KPX A quotedblright -30     KPX A quotedblright -90
KPX A q -9                    KPX A q 0                     KPX A q -10                   KPX A q -18
KPX A p -4                    KPX A p 0                     KPX A p 0                     KPX A p -10
KPX A o -12                   KPX A o 0                     KPX A o -10                   KPX A o -18
...
EndKernPairs                  EndKernPairs                  EndKernPairs                  EndKernPairs
```

Figure 5: The four AFM files for the Myriad master designs



Figure 6: Minion instances from opposite ends of the optical size axis set at the same size (exaggerated)

Dotted and Dashed Lines in METAFONT

Jeremy Gibbons

Department of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand.
Email: jeremy@cs.auckland.ac.nz

Abstract

We show how to draw evenly dotted and dashed curved lines in METAFONT, using *recursive refinement* of paths. METAPOST provides extra primitives that can be used for this task, but the method presented here can be used for both METAFONT and METAPOST.

Introduction

Knuth's METAFONT has powerful facilities for manipulating and drawing curves or 'paths'. These facilities are generally sufficient for METAFONT's primary intended purpose, namely drawing letters. However, METAFONT is also very well suited to producing technical diagrams; for this secondary purpose, METAFONT lacks a valuable facility — that of drawing evenly dotted and dashed curves. In this paper we show how to remedy this shortcoming, using the facilities that METAFONT does have available.

John Hobby's METAPOST is an adaptation of METAFONT for producing PostScript output rather than bitmaps. METAPOST *was* primarily intended for producing technical diagrams (Don Hosenk reports Hobby as saying, 'Well, you *could* use it for generating characters, but I wouldn't recommend it'). METAPOST therefore provides an ingenious scheme for drawing dotted and dashed lines: an arbitrary picture can be used to generate a dash pattern for drawing paths. This scheme is not very general — there are reasonable dashed-line-like applications for which it does not work — but METAPOST also provides lower-level primitives `arclength` and `arctime` that are quite general. These primitives make the approach presented in this paper largely redundant for METAPOST, but it remains necessary for the 'core METAFONT' language.

Throughout this paper, we use the term 'METAFONT' to refer to both Knuth's METAFONT and Hobby's METAPOST; we use the term 'METAPOST' to refer just to Hobby's METAPOST.

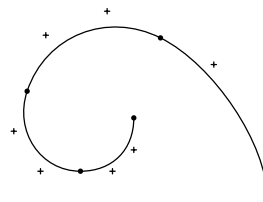
Cubic Bézier curves

METAFONT represents curved lines by *piecewise cubic Bézier curves*, which it calls 'paths'. These are discussed in Chapter 3 of the METAFONTbook; we summarize here all that is needed for the purposes

of this paper. For most of this paper, we consider only *non-cyclic* paths; we discuss cyclic paths briefly at the end of the paper.

A path is specified by a sequence of *knots* and *control points*. The path runs from the first knot to the last knot, passing through each knot in turn. Between each pair of consecutive knots, there are two control points; the path leaves one knot in the direction of the next control point, and enters the next knot in the direction of the previous control point. A path with $n + 1$ knots is said to have *length* n , and can be considered as a function from 'time' (i.e., the real numbers) between 0 and n inclusive to points in the plane; times that are natural numbers correspond to the knots, with time 0 the start of the path and time n the end. (Throughout this paper, we use the term 'length' as a measure of the number of knots in a path; it is always a natural number. In contrast, we use the term 'arc length' for the spatial distance covered in travelling along the path.)

For example, here is a spiral path of length 4, and the knots (dots) and control points (crosses) used to generate it.



This path was drawn by the METAFONT code

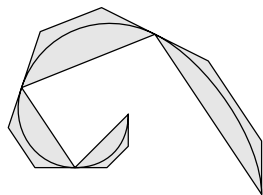
```
p := (90,0) .. controls (90,20) and (70,50) ..  
      (50,60) .. controls (30,70) and (7,61) ..  
      (0,40) .. controls (-5,25) and (5,10) ..  
      (20,10) .. controls (32,10) and (40,18) ..  
      (40,30);
```

`draw p`

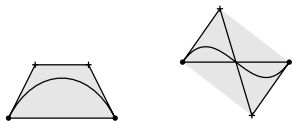
METAFONT actually has sophisticated algorithms for choosing 'nice' control points given just the knots and possibly some other information, but that does

not concern us here; whichever way the path is specified, METAFONT represents it internally as knots and control points.

An important property of piecewise cubic Bézier curves is that each *segment* of a path (i.e., a part between two consecutive knots) lies entirely within the ‘convex hull’ of—that is, the smallest convex polygon surrounding—the knots at either end and the control points in between. For example, here is the same spiral, with the convex hull surrounding each path segment shown shaded grey.



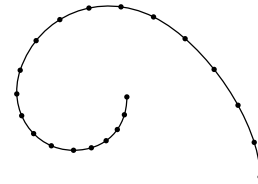
There are two general forms for a segment of a path: either the control points between the two knots are both on the same side of the ‘chord’ between the knots, or they are on different sides. These two cases are illustrated below.



Notice that, in either case, the arc length of the chord between the knots is no greater than that of the path, and the arc length of the ‘control polygon’ (consisting of three straight lines, from the first knot to the second via the control points) is no less than that of the path; this fact is important in what follows. Clearly, this property holds of paths as well as path segments. Also, the degenerate case in which both control points are on the same line as the knots yields a path that also lies entirely on that line; the arc lengths of the chord and control polygons again form lower and upper bounds on the arc length of the path.

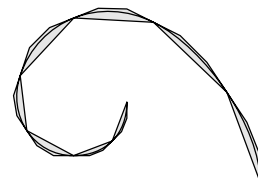
Evenly-spaced points on a path

The essential problem when it comes to drawing dotted or dashed lines is that points evenly separated in *time* along a path are not necessarily evenly separated in *space*. For example, here is the same spiral path as above, with 21 dots spaced evenly in time—that is, placed every $\frac{1}{5}$ th of a time unit. Notice how the dots get closer as the curve gets tighter; a point moves at different ‘speeds’ in space as it progresses along the path evenly in time.



Producing evenly-spaced dots basically involves finding the arc length of a cubic Bézier curve. This is a difficult mathematical problem; it involves integrating the square root of a degree-four polynomial, which in turn can only be done analytically using ‘elliptic integrals’—not one of the primitives provided by METAFONT.

Fortunately, there is a simple approximation method for finding the arc length; this method is the subject of this paper. The basis of the method is recursive *refinement* of the path, picking more and more knots on the path and hence using control points that are closer to the curve. For example, if we pick an extra knot half way (in time) between each pair of consecutive knots on the above spiral path, we get the following picture:



This spiral path is generated by the METAFONT code

```
path q;
q := subpath (0,0.5) of p
    for i := 1 step .5 until length p:
        & subpath(i-0.5,i) of p
    endfor;
```

The path itself has not changed (although it is now travelling at half of its original speed), but the chord and control polygons are much closer approximations to the curve. How good are these approximations? Gravesen (1993) shows that, under repeated recursive refinements, the average of the arc lengths of the chord and control polygons converges very quickly to the arc length of the path¹. We use recursive refinement to get a sufficiently-close polygonal approximation to a path, and then divide that polygonal approximation up evenly in space. This yields (for example) the result below.

¹ In fact, the average of the arc lengths of the chord and control polygons under k recursive refinements converges to the arc length of the path as 16^{-k} ; that is, the error decreases by a factor of 16 on every iteration. Gravesen also gives a general result for degree- n Bézier curves.

```

vardef chordpoly expr p =
  save i; numeric i;
  point 0 of p
  for i := 1 upto length p:
    -- point i of p
  endfor
enddef;

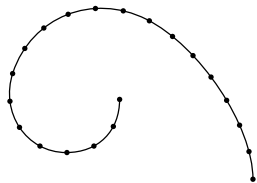
vardef controlpoly expr p =
  save i; numeric i;
  point 0 of p
  for i := 1 upto length p:
    -- postcontrol (i-1) of p
    -- precontrol i of p
    -- point i of p
  endfor
enddef;

vardef chordlen expr p =
  save i; numeric i;
  0
  for i := 1 upto length p:
    + abs(point i of p - point (i-1) of p)
  endfor
enddef;

vardef controllen expr p =
  chordlen (controlpoly p)
enddef;

```

Figure 1: The functions `chordlen` and `controllen`



Notice now that the dots are evenly spaced, even as the curve gets tighter.

Recursive refinement

In order to refine a path, we need first to be able to compute lower and upper bounds to its arc length. The functions `chordlen` and `controllen`, which return the arc lengths of the chord and control polygons, are defined in Figure 1. (The METAFONT expression `point i of p` returns the position of the path at time i ; for natural i , `postcontrol i of p` returns the first control point after knot i , and similarly, `precontrol i of p` returns the last control point before knot i .)

Given the lower bound `chordlen p` and upper bound `controllen p` to the arc length of a path p , recursive refinement is straightforward. If the

bounds are sufficiently close, we return just p ; otherwise we split p into two halves (time-wise), independently refine the two halves, and join the results back together. We use a multiplicative rather than additive test for ‘sufficiently close’, so that if a number of subpaths are independently ‘sufficiently refined’ then their concatenation will also be. Note that splitting the path in two doesn’t double the length, as we did in our earlier spiral example; it only adds zero or one more knot (depending on whether or not `length p` is even). However, it does give the advantage of *adaptive* refinement—nearly straight parts of the path are not refined as much as very wiggly parts.

```

numeric tol; tol := eps;
vardef refine expr p =
  if (controllen p) <= (1+tol)*(chordlen p):
    p
  else:
    (refine (subpath(0, length p/2) of p)) &
    (refine (subpath(length p/2, length p) of p))
  fi
enddef;

```

Marking a path evenly

Having refined the path p , we still need to divide it into equal-sized chunks; that is, we need to find a sequence of times t_0, \dots, t_r such that the arc length between points t_i and t_{i+1} of p for each i is some fixed given distance d . However, we now have a polygonal path `chordpoly(refine p)` which very closely approximates p . It is straightforward (if a little messy) to find the times that divide this polygonal approximation into chunks of arc length d ; we simply use those same times for the curved path p .

The code for the function `markedevery` is given in Figure 2. The function takes in a path p and a distance d , and returns the sequence of times that divide the chord polygon of p evenly into chunks of arc length d . This function should be called only on a path which is ‘very nearly’ polygonal—one that is actually polygonal, or perhaps the result of refining another path.

The program maintains variables t , the ‘current time’, which increases from 0 to the length (in time) of p and is always at an integer value (in fact, equal to `knot`) at the start of the outer loop body, and d_{next} , which is the distance from the ‘current point’ (point t of p) until the next mark. Each path segment is considered in turn, the time counter t advancing as required along it. Note that the first and last ‘chunks’ taken from a segment may be shorter than d .

```

secondarydef p markedevery d =
  begingroup
    save q; path q; q := (0,0); % for result
    save dnext; numeric dnext; dnext := d;
    save seglength; numeric seglength;
    save knot; numeric knot;
    save t; numeric t; t:=0;
    save dt; numeric dt;
    for knot := 0 upto length p - 1:
      seglength := abs (point (knot+1) of p
        - point knot of p);
      % arc length of this segment
      if seglength > 0:
        forever:
          dt := dnext / seglength;
          % time to next mark (if this segment)
          exitif t+dt > knot+1;
          % exit if next mark not on this seg
          t := t+dt; % move forwards...
          q := q -- (t,0); % ... & put mark here
          dnext := d; % next mark is d away
        endfor
        % now t <= knot+1 < t+dt
        dnext := dnext - (knot+1-t)*seglength;
        % put leftover towards next mark
        t := knot+1;
      else: % empty seg (coincident knots)
        t := t+1;
      fi
    endfor;
    q % return time sequence we've built up
  endgroup
enddef;

```

Figure 2: The function `markedevery`

```

def drawdotted (expr p, d) =
  save refined, marks, i;
  path refined, marks; numeric i;
  refined := refine p;
  marks := refined markedevery d;
  for i := 0 upto length marks:
    drawdot
      point (xpart(point i of marks))
        of refined;
  endfor
enddef;

```

Figure 3: The procedure `drawdotted`

METAFONT has no ‘list’ type that can be used to return the sequence of times t_0, \dots, t_r , so we return the path $(t_0, 0) \dashrightarrow \dots \dashrightarrow (t_r, 0)$ instead; this path is built up in the variable `q`.

Dotted and dashed lines

The function `markedevery` does the work of marking a path evenly in space; as described above, to draw a dotted path we first refine it, and mark the refined polygonal approximation evenly instead. The procedure `drawdotted` is defined in Figure 3.

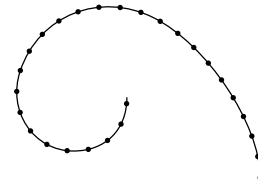
For example, we can place dots every eight units in space on our spiral path as follows:

```

pickup pencircle scaled 0.5;
draw p;
pickup pencircle scaled 2;
drawdotted (p, 8);

```

This yields the picture



Notice that there is no guarantee that the last dot will be at the end of the path. To ensure that this is the case, we must choose the dot spacing to divide evenly into the arc length of the path. Fortunately, the average of the arc lengths of the chord and control polygons makes a very good estimate of the arc length, as discussed above. The evenly-dotted spiral on page 261 with the 21st dot exactly at the end of the path was drawn with the commands

```

pickup pencircle scaled 0.5;
draw p;
pickup pencircle scaled 2;
path q; q := refine p;
drawdotted (p,
  .5[chordlen q,controllen q]/(20*(1+eps)));

```

(Notice that we have to scale down the ‘ideal’ dot spacing by a factor of `1+eps`, to ensure that the last dot is just on rather than just off the end of the path in case of rounding errors.)

Dashed lines can be drawn using pretty much the same approach. Here is a simple-minded macro to do it.

```

def drawdashed (expr p, d) =
  save refined, marks, i;
  path refined, marks; numeric i;
  refined := refine p;
  marks := refined markedevery d;
  if (length marks) mod 2 = 0:
    marks := marks -- (length refined,0);
  fi

```

```

vardef refine expr p =
  if (controllength p)<=(1+tol)*(chordlength p):
    p
  else:
    (refine (subpath(0, length p/2) of p)) &
    (refine (subpath(length p/2,length p) of p))
    if cycle p: & cycle fi
  fi
enddef;

```

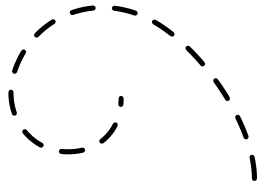
Figure 4: refine for cyclic paths

```

for i := 0 step 2 until length marks - 1:
  draw subpath(xpart (point i of marks),
    xpart (point (i+1) of marks)) of refined;
endfor
enddef;

```

For example, with the same spiral path and $d = 8$ we get



Notice that we add an extra mark at the end of the path (yielding a partial final dash) if the number of marks is odd.

A more elaborate approach would allow differing dash and gap sizes, and displacing the dashes. This could be done by using the greatest common divisor of the various distances to mark the path, or perhaps by altering the `markedevery` function to take several distances as arguments.

Cyclic paths

The same recursive refinement technique works just as well for cyclic paths; in fact, the only change that is needed is to the function `refine`. When we split a path into two halves, we need to remember whether the path is cyclic, recombining the halves as a cycle if so. The code for this version of `refine` is given in Figure 4.

Iterative non-adaptive refinement

The recursive refinement technique described here is quite elegant, but it can cause METAFONT to run out of stack space on very wiggly paths. An iterative approach can avoid this, at the cost of not easily providing adaptive refinement; we simply repeatedly double the length until the lower and upper bounds on the arc length are sufficiently close. The code is given in Figure 5.

```

vardef refine expr p =
  save q; path q; q := p;
  forever:
    exitif (controllen q)<=(1+tol)*(chordlen q);
    q := subpath (0,0.5) of q
    for i := 1 step .5 until length q:
      & subpath(i-0.5,i) of q
    endfor
    if cycle q: & cycle fi;
  endfor;
  q
enddef;

```

Figure 5: An iterative version of refine

An unusual application

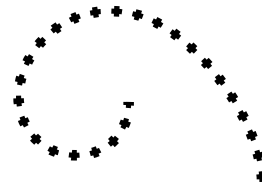
As we mentioned in the introduction, Hobby's METAFONT does provide primitives for drawing dotted and dashed lines. You can do some surprising things with dashes; for example, you can draw crosses along a path by drawing the path twice, once with long thin dashes and once with short fat dashes.

```

interim linecap:=butt;
draw p dashed dashpattern(on 6 off 6)
  withpen pencircle scaled 2;
draw p dashed dashpattern(off 2 on 2 off 8)
  withpen pencircle scaled 6;

```

(The assignment to `linecap` produces square, rather than rounded, ends to lines.) Unfortunately, although METAFONT generates good PostScript from such constructions, bugs in many PostScript interpreters make them come out wrong. For example, the following picture should consist of crosses, but on some PostScript interpreters some of the crosses turn out mushroom-shaped.



Still, there are some things that cannot be done with METAFONT's dash primitives, but can be done with the techniques described here. We conclude this paper with one such application. (In fact, this application was the original motivation for the author's interest in the topic.) Note that this problem can also be solved using METAFONT's `arclength` and `arctime` primitives, but then the solution is not portable to 'core METAFONT'.

Several years ago, while a PhD student, the author used to play in a jazz band called the *Mississippi Muskrats*, and he endeavoured to produce a

logo for posters for the band. This logo included a picture of a muskrat, for which a ‘furry’ effect was obtained by drawing diagonal lines across a path. The first attempt at drawing a furry muskrat used the same time interval for every path, and yielded a bushy throat and a threadbare back:

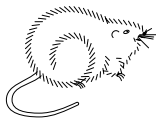


(The author makes no claims for the artistic merit of these drawings.)

The second attempt used different time intervals for different paths, and was much better; still however, the muskrat’s back suddenly gets hairier about halfway from the tail to the neck, and of course there’s the hassle of choosing all those different numbers.



The third attempt used the method described in this paper, and gave a much healthier-looking muskrat:



Acknowledgements

The author would like to thank Jens Gravesen for his very elegant paper (Gravesen, 1993), and Alan Hoenig, Geoffrey Tobin and several other people who have put up with discussions of this topic electronically and in person over the last few years.

References

- J. Gravesen. “Adaptive Subdivision and the Length of Bézier Curves”. Technical Report 472, The Danish Center for Applied Mathematics and Mechanics, Technical University of Denmark, 1993.

T1part: Printing T_EX Documents with Partial Type 1 Fonts

Sergey Lesenko

Institute for High Energy Physics
Scientific Information Department
Protvino, Moscow Region, Russia, P.O. Box 35
Email: lesenko@desert.ihep.su

Abstract

A large fraction of the scientific papers available on the Internet have been created by T_EX and dvips. Most of these papers use Computer Modern fonts at 300 dpi, and thus neither preview well with a typical screen resolution of less than 100 dpi, nor take advantage of the higher resolution of current laser printers. This paper presents T1part, a set of subroutines that decompose Type 1 PostScript fonts, including only those characters needed in a particular document. This package, in conjunction with a high-quality freely available set of Computer Modern fonts, can provide for the distribution of compact PostScript files that preview legibly and print beautifully. In addition, this package allows the printing of much more complex documents using more fonts than have previously been available in dvips. The T1part subroutines are modular and can be easily incorporated into other drivers.

Introduction

Using Type 1 PostScript fonts in T_EX documents has been problematic for a number of reasons. Not least among these problems is the requirement that either the fonts be included in the document itself, or that the fonts be available to all potential recipients of the document. Including such fonts in the document itself raises serious copyright issues, as well as causing the resulting file to be large and slow to print. In this paper, we present T1part, which allows such fonts to be included in PostScript output in a partial form. It is currently integrated with Tomas Rokicki's dvips program (Rokicki, 1994).

While commercial programs with this feature have been available for a number of years, this is the first freely available integrated implementation of this functionality.

The idea for this paper was suggested by Basil Malyshev's paper (Malyshev, 1994). The program is based on information available in the Adobe "Black Book" (Adobe Systems, 1990). In addition, I found work by Rajeev Karunakaran (1994), Chris B. Sears (1991), and Al Stevens (1994) to be useful.

This paper discusses the interface, algorithms, and finally the efficiency of using the T1part program.

Interface

T1part, as a subroutine, needs to be told what fonts and what characters in each font to include. The fonts are indicated by a file name pointing at a PFB or PFA Type 1 font. The set of characters to include can be specified either by a list of glyph names or by a set of character codes. In the latter case, the character codes must be translated into glyph names through the font's encoding vector.

The resulting output of the program is a partial font in PFA format. As integrated into dvips, the program inserts this font directly into dvips' output stream.

Algorithm

Before we present how T1part works, we must describe the format of a Type 1 font. We will first consider a font in PFB format; the PFA format is a simple modification of this. A Type 1 font in PFB format has the following structure and relevant keywords in each part:

- ASCII part
 - *keyword* /**Encoding**
- BINARY part (eexec encryption)
 - *keyword* /**lenIV**
 - *keyword* /**Subrs**
 - *keyword* /**CharStrings**
- ASCII part

By “ASCII”, we mean the portion of the file that is not eexec encrypted.

Since the font contains three different parts, each part has its own parsing process. The minimum set of keywords listed above allows us to quickly parse the input at a high speed and with a simple parser.

The parsing process is as follows. First, we read the initial ASCII portion and search for the **Encoding** keyword. After finding it we define its type with the help of the next token. If the next token is **StandardEncoding**, we will assume that the Adobe Standard Encoding is the default font encoding vector.

If a reencoding has been specified for this font in the dvips `psfonts.map` file, then we use that reencoding to translate the character codes to glyph names. Otherwise, we try to parse an encoding from the input stream by searching for index and glyph name pairs. If we do not find such an encoding, we search for and parse an AFM file associated with the font.

Next, we parse the BINARY portion of the font. We start by performing the eexec decryption and loading the result directly and entirely into memory. We then scan the decrypted section for the keywords `/lenIV`, `/Subrs`, and `/CharStrings`. Each of the sections separated by these keywords has a similar structure.

When initially parsing the **Subrs** section, we simply identify what subroutines exist by number and keep track of the number of tokens in each subroutine definition. We do not initially send out any subroutines.

When we parse the **CharStrings** section, we initially identify what subroutines are associated with each character. We parse the subroutines for each used character, recursively diving into the **Subrs** entries as necessary, marking which subroutines are actually used. If there are multiple **Subrs** sections, as is the case with some hybrid fonts, we consider all such sections.

If a required character is a composite character, then the component characters will be marked and processed as above.

For efficiency, all selected subroutines are decrypted only once; a flag associated with each subroutine is used to indicate whether that subroutine has already been decrypted.

After finishing the scanning phases, the size of the **Subrs** and **CharStrings** arrays must change to reflect the deleted subroutines. The new values are:

- **Subrs** size—largest selected subroutine plus one

- **CharStrings** size—number of selected charstrings

We retain the indices of the subroutines for simplicity (so we do not need to rescan and modify each subroutine) and to easily guarantee that each subroutine only refers to those with a lower index (an Adobe requirement that prevents recursive subroutines).

Finally, the selected portions of the scanned memory are eexec encrypted and directed to the output in hexadecimal (PFA) form.

To finish up the font, the final ASCII portion is sent to the output without changes.

If the input font is in PFA format, the keywords that define the beginning of the hexadecimal eexec section are **currentfile eexec**; a line of all zeros marks the end of the section.

Results

T1part was tested by integrating it into dvips and running it over a number of files using the BaKoMa collection of Computer Modern fonts, as well as the Acrobat Reader selection of Adobe fonts. The results of these tests are:

- *Size of output file.* The total size of the resulting PostScript file when using T1part and PS fonts was compared with that when using bitmap PK fonts under both 300 dpi and 600 dpi. Usually, the bitmapped font output was slightly smaller at 300 dpi, but slightly larger at 600 dpi. Thus, using partial Type 1 fonts is practical from the perspective of final file size.
- *Reduction in PS fonts.* In our tests, we found that usually less than half of the characters from body text fonts were used, and a very small fraction of special fonts were used. On average, the size of the partial PostScript font created by T1part was less than 30% of the size of the original font.

The figures are given in detail in tables 1–3.

The output of the program has been tested with GhostScript (Deutsch, 1993), and the time and memory required to view documents created with partial fonts was less than those with the whole fonts.

It is clear that the popular paradigm of \TeX —dvips—GhostScript will be more efficient when using the integrated T1part functionality.

Acknowledgements

I would like to thank Tomas Rokicki for his helpful discussion and suggestions.

References

- Adobe Systems. *Adobe Type 1 Font Format, version 1.1*. Addison Wesley, 1990.
- L. P. Deutsch. “Aladdin Ghostscript version 2.6.1”. Electronic distribution from `ftp.cs.wisc.edu` via CTAN, 1993.
- R. Karunakaran. “PFB2PFA program”. Electronic distribution via `comp.sources.postscript`, volume 03 issue 51, 1994.
- B. K. Malyshev. “Problems of the conversion of Metafont fonts to PostScript Type 1”. In *Proceedings of TUG94*, edited by M. Goossens, Santa Barbara, CA. 1994.
- T. Rokicki. “Dvips: A T_EX Driver”. Electronically distributed from `labrea.stanford.edu` via CTAN with `dvips`, version 5.58, 1994.
- C. B. Sears. “CHARS program”. Electronic distribution via `comp.sources.misc` volume 19 issue 94, 1991.
- A. Stevens. “Quincy: a C interpreter”. *Dr. Dobb’s journal* (09), 1994. Electronic distribution as `ftp://ftp.mv.com/pub/ddj/1994/1994.09/qnc41.zip`.

Table 1: Dvips’ output* with various modes of used fonts (`dvips.dvi`, version 5.58, used as input).

Font type	PS Type 1		PK				
	Full	Partial	300	600	1016	1200	1800
Mode of font embedding							
Output resolution (dpi)							
Output size (bytes)	762981	398358	424529	1012195	3082215	4487168	14933095
Output with compressed bitmap fonts (bytes)	–	–	274205	335981	418895	480083	622405
Disk space required for fonts** (bytes)	277579	277579	104036	251756	479080	588696	947684

* Dvips’ output size without embedding of fonts — 228918 bytes

** PFB format is used for PS Type 1 fonts

Table 2: Efficiency of partial font downloading for each used font (dvips.dvi used as input)

Font name	cmbx10	cmr10	cmr8	cmsl10	cmti10	cmitt10	cminch	cmmi10	cmsy10	cmsy7	logo10	Total
Font version	*	*	*	*	*	*	**	*	*	*	**	
Percentage characters used	55	75	14	27	28	72	19	6	7	7	99	32
Partial font (bytes)	28 757	40 675	7 314	14 903	18 382	37 908	4 438	3 628	3 862	3 929	5 301	169 097
Full font (bytes)	52 768	53 960	53 360	54 870	65 948	52 915	23 211	63 752	52 613	54 951	5 372	533 720

Table 3: Results of using DC instead of CM fonts

Font name	dcbx10	dcr10	dcr8	dcsl10	dcti10	dctt10	cminch	cmmi10	cmsy10	cmsy7	logo10	Total
Font version	***	***	***	***	***	***	**	*	*	*	**	
Percentage characters used	34	44	9	18	17	44	19	6	7	7	99	23
Partial font (bytes)	25 766	34 039	6 732	14 555	15 827	36 450	4 438	3 628	3 862	3 929	5 301	154 527
Full font (bytes)	75 311	77 211	76 871	80 687	92 105	83 261	23 211	63 752	52 613	54 951	5 372	685 345

* BaKoMa/CM Fonts Collection (1.3, (Level-C), January 95)

** Paradissa Fonts Collection (1.0-prerelease, 1993)

*** BaKoMa/DC Fonts Collection (1.0, (Level-B), 1994)

Modularity in L^AT_EX

Matt Swift

59 Brainerd Rd #202
Allston MA 02134-4564
USA
Email: <swift@bu.edu>

Abstract

The author surveys several kinds of desirable modularity in L^AT_EX and argues the advantages of a system in which sources and macros may inhabit modules called bits and features, respectively, that are independent of their context in a disk file and are identified by names independent of disk file names. The author discusses implementation, and sketches are given of solutions involving extensions, enhancements, front ends, and back ends to T_EX. The author has made available code which adds some new modular features to L^AT_EX.

This paper aims to clarify several issues in the use and development of L^AT_EX that belong under the general heading of modularity. The issues have arisen during attempts to solve particular problems.

Certain modular features are not worth implementing in L^AT_EX 2_ε and should instead be the concern of those who are designing and implementing L^AT_EX 3, ε-T_EX, $\mathcal{N}\mathcal{T}\mathcal{S}$, and the tools that will accompany them. Other modular features can be implemented in L^AT_EX without a large programming effort or extensive changes in user syntax. The author has implemented some modest modular features in L^AT_EX with encouraging results.

Some terms will be given a precise definition below, but *modularity* will be used only as a broad descriptive term.

Several concerns addressed here have been addressed earlier in the companion papers “An Object-Oriented Programming System in T_EX” (Baxter, 1994) and “Object-Oriented Programming, Descriptive Markup, and T_EX” (Ogawa, 1994) published in the T_EX Users Group ’94 proceedings (*TUGboat* 15, no. 3). I recommend these thoughtful papers to readers interested in this subject. Baxter describes a L^AT_EX-like markup that realizes many of the benefits of object-oriented program design. Its syntax could for the most part exist simultaneously with standard L^AT_EX. Ogawa enumerates some ways L^AT_EX falls short of an ideal object-oriented markup language.

What is a document?

The L^AT_EX processor and an accompanying suite of programs and hardware devices translate a com-

puter disk file called a *L^AT_EX source* (“source”) into a document. One can reasonably say, in our context, that a *document* is a text-dominated visual presentation of information whose canonical form is an ordered collection of pages.

We are used to calling many things documents that are not documents in this sense. A source is not a document — not a visual presentation of information — nor is it intended to be a complete description of a document (though it is such, trivially, when it is viewed or printed). A source is a partial, abstract description of a document, a meta-description of a document. The L^AT_EX processor interprets the minimal information in a source by means of implicit conventions and explicit rules, and the resulting interpretation (the *dvi* file) is a complete document description. A subsequent procedure transforms the document description into a real document, a real image on a screen or page.

There are several good reasons to create and maintain sources instead of document descriptions or documents themselves. Source files are much smaller than the files that result from them, since they contain less information. They are human-readable, whereas document descriptions, in the interests of efficiency, are difficult or impossible to read. Worrying about the large amount of extra information necessary to describe a document distracts authors, to whom it is superfluous. Sources are a versatile form because they can be used to produce many different kinds of documents or even presentations of the information that are not documents in the present sense, such as aural presentations (see Raman, 1995).

Our goal is to allow a computer program to present information to our senses in a manner that meets our needs, which can change from time to time and from user to user. Ideally, marked-up text in the source makes explicit all the information we would like the document in any form to convey. Marked text is a very efficient conventional form of information. Typically, we wish to present the information in a manner that most greatly facilitates the reader's understanding by presenting various aspects of its content efficiently to the eye, which can process it extremely quickly. But there is not a single superior scheme for accomplishing this. Also, because perceiving a document is a personal experience like any other, typesetting can aim to be beautiful, or, as in advertising, manipulative. To meet these various needs, a variety of documents can be derived from the source by reprocessing it with different parameters.

Practically, it is very hard to decide what information should be made explicit by markup, and what can be assumed. When we stretch the borders of the present definition of a document, we change our ideas of what elements must be differentiated in the document and consequently discover a need to mark structures that were handled implicitly for more familiar documents. In some languages, for example, the sequence of glyphs in the source must be presented right-to-left, or top-to-bottom, or both, not in the way of written English. The notion of pages is ill-adapted to a computer-screen. And when we change the presentation's form from visual to aural, we discover that some of our markup is visual and not as abstractly informational as perhaps we thought.

One can consider a \LaTeX source from several different points of view. Very basically, it is a program written in \TeX , which is a “list-based macro language with late binding” (NTS, 1995), and using the \LaTeX macro library. From another point of view, as mentioned, it is a meta-description of a visual presentation of text-dominated information. If a certain method of processing the source, such as standard \LaTeX , is agreed upon, then it can be agreed that a source is a complete description of a document.

Our present interest suggests another point of view that considers a source to comprise two kinds of elements identified by markup (not including comments). A *block element* is a list of atoms—irreducibles such as font glyphs and spaces—other block elements, and/or anchor elements. An *anchor element* is a signal to the formatter that it should

act as if certain atoms or block elements were in the source at that point.

Anchors are used when it is better that the formatter supply data from somewhere at runtime, rather than the user include it in the source document. There is a close correspondence between anchors and HTML entities (see Goossens and Saarela, 1995a). An anchor used for citation, for example, might generate two block elements such as (Hobson 1956) and an entry in a bibliography at the end of the chapter.

From a more abstract point of view, blocks and anchors are markup functions. The difference—no strict—is that blocks take an argument, often a long one, of text to be represented quite directly in the typeset image, whereas anchors do not take an argument of this kind.

A source is therefore itself a single block element. Once anchor references are resolved, it may be considered to be a single list of atoms. Any number of contiguous regions of the list are identified (“tagged”) as subordinate block elements. Block elements may nest but not overlap.

In the translation from source to document, the order of the list of atoms is for the most part preserved. Certain classes of block elements, however, are conceptually independent of their contexts in the source, and routinely appear in new contexts in the document. We might call this kind of block element a *float element* as opposed to a *fixed element* whose immediate context does not change in the translation. Float elements include footnotes, \LaTeX floats, marginalia, and entries in bibliographies and indices. The existence of float elements is possible exactly because block elements do not overlap in the source.

\LaTeX is both the language in which sources are written and the engine which creates a document description from a source. Users must distinguish these capabilities or confusion and inefficiency may result. Regions in the list of atoms that compose the source are ideally tagged by markup that describes what the region is; that is, markup that is not specific to any particular document description derived from it. Some “visual markup”, or markup that describes how the region should be actually presented, is inevitably necessary during the creation of complex documents, when the automatic procedures break down and require manual aid. What is most to be regretted, however, is the confusion and encouragement of inefficient habits that results from allowing the same syntax for both the desirable and undesirable kinds of markup.

A new block element is needed

Block elements usually constitute the majority of a source, conceptually and physically. Let us use the term *block modularity* to describe what allows L^AT_EX to handle block elements independently and abstractly. L^AT_EX already provides block modularity in several useful ways. An itemized list, for example, is a natural unit of text for which L^AT_EX provides the `itemize` environment whose appearance in the document is controlled by macro definitions that are in effect only inside the environment. This kind of local definition is made possible by T_EX's grouping mechanism. The items within the list are blocks in their own right, and these in turn might be divided further into paragraph blocks or other kinds of blocks, such as emphasized phrases, and so on. The `document` environment is another natural unit of text, a much more general one. Its appearance is controlled by the preamble, which includes `\documentclass` and `\usepackage` commands.

These examples show that block elements are specified in the source file in a broad variety of ways. Some of the causes and consequences of this variety are adduced in another section below.

I propose as a useful concept a block element of intermediate size called a *bit* defined as a *conceptually independent unit of text*, a block element that might reasonably appear in new contexts. If the block could reasonably appear with a null context, that is, could alone generate a reasonable document, it is not only a bit but also a *potential document*. The name is appropriate because any document could conceivably be placed in a new context, though long documents are quite unlikely to appear in any but the null context.

A collection of poems or recipes would consist almost entirely of a sequence of bits. An academic paper would probably contain only one bit, its entire self, but might contain a bit here or there, such as an embedded poem, illustration, or derivation. The standard L^AT_EX *letter* class provides a clear example of a bit structure. The `document` environment in a source of the *letter* class consists of one or more L^AT_EX `letter` environments, each of which is functionally modular and conceptually independent.

Bits are similar to sections as we might think of them when we read or write. *Sections*, however, may not be conceptually independent. The L^AT_EX sectioning commands such as `\chapter` have limited power and serve for the most part as informative markers in a continuous stream of text. Therefore they should be considered not as tagging the long block element that follows them, but rather as

an anchor specifying one fixed element (the section heading) and one float element (the entry in the table of contents).

The status of the L^AT_EX sectioning commands as anchors and not block delimiters is not required by their syntax; that is, not required because they occur singly and do not enclose text. The familiar sectioning commands *could* in fact be implemented to give a programmer control over the following text as a block element, but they are not implemented this way.

When we consider the modularity of sources and documents, bits are the most basic and important unit. They are atomic in the sense that every document has an integral number of them. When we think about the exchange of sources, we should think in terms of the bit.

A generally useful implementation of bits would have several key characteristics. First, there should be an anchor for bits, a command which says “put this bit here”. Bits should also be as independent as possible of disk files. By this I primarily mean 1) they should be referred to by names which do not depend on the disk file in which they occur, and 2) as L^AT_EX program code they should be portable — as independent as possible of their immediate programming context in the disk files that contain them. A bit should have a type and a name — a unique label. To process a source is to process a bit of type “document”. This task comprises relatively independent subtasks, one corresponding to each bit type, and one to handle the presentation of the entire sequence of bits and intervening text, the presentation of the document as a whole.

Many users are probably accustomed to implementing bits implicitly by taking advantage of the modularity of disk files. A bit can be put into its own disk file which can be incorporated into new contexts by the action of disk file inclusion. The bit name is the disk file name. Chief among the advantages of identifying bits explicitly in the markup is that disk file structure and disk file names can be freed from this frequent burden of bearing information about bits. This change facilitates several beneficial developments proposed below. Summarily, the introduction of a layer of abstraction between disk files and user syntax makes possible a broad and desirable flexibility in the distribution of source documents among disk files. Disk files might ideally be construed as objects which can export bits.

Many things said so far about source documents and disk files apply also to code libraries and disk files. In particular, there are similar advantages in recognizing functionally independent segments of

code through markup, rather than disk file structure, and to introducing a layer of abstraction between disk files and programmer syntax. These segments, which we can call *features* after GNU Emacs, are exact analogs of the bit in a context of programming code. The breaking up of the kernel and the standard document classes into functionally independent modules is the intended first stage of the $\mathcal{N}\mathcal{T}\mathcal{S}$ project (NTS, 1995).

Present limitations on \LaTeX disk file structure

The \TeX primitives `\input` and `\endinput` determine the possible structures of \LaTeX disk files. The `\input` command is an anchor for the contents of a disk file. When the contents of a disk file are being processed at the request of an `\input` command, the `\endinput` command signals the formatter to act as if the end of the file was encountered at the end of the current line.¹

Though many disk file configurations are possible, only one seems practical, namely, a *principal source* that contains a single `document` environment, and any number of *auxiliary sources* that contribute material to the principal source or to other auxiliary sources by disk file inclusion.

\LaTeX provides two more sophisticated interfaces to the `\input` primitive. The `\include` and `\includeonly` commands implement a convenient way to enable and disable the inclusion of files. The price is that `\include` commands cannot be nested like `\input` commands. The group of class and package commands new in $\LaTeX 2_{\epsilon}$ keep track of what files have been included, perform some checking to ensure that the right file was included, and implement a system of user options — a system of passing certain information to the included files.

Because both extensions are founded on the primitive `\input`, the basic modular unit is still the disk file, identified by its name. Several alternative ways to identify bits and features may now be suggested. One difficulty to keep in mind is that file names are often exactly how users choose to uniquely identify their information. Consider two versions of the same composition residing in two files in the same directory, distinguished only by a few changed words.

The possibilities seem to be:

1. Mandate a new \TeX primitive

```
\inbit{<bit-name>}
```

¹ Suppressing the continuation until the end of the line is a surprisingly complex task; here, to my mind, is a good candidate for a change to \TeX .

or primitives upon which such a command could be constructed. One can imagine a `kpathsea`-like system library which handles requests for both files and bits. When a search is ambiguous, the engine could issue a warning.

2. Bend the rules and allow the back end (system-dependent implementation) of `\input` to search for bits if its argument does not match a file name. This change would not break the `trip` test, and need not be considered an change to \TeX .

The search method for bits could be path searching like `kpathsea` or involve other schemes like file stamp attributes or a catalog of cross references like the SGML Open HTML catalog file (see Appendix D of Goossens and Saarela, 1995b). When a search is ambiguous, the engine could issue a warning.

This is the most promising solution to pursue.

3. Develop a sophisticated front-end to \LaTeX that will hide from the user a preprocessing stage that assembles a conventional source document for compilation.

A back-end solution is going to be more stable and portable than a front-end solution, but it must inevitably be less ambitious and slower to appear.

Experiments in \LaTeX

The author is in the process of implementing bits in a limited way within \LaTeX . The ongoing project, called *Frankenstein*, will provide several bit types as well as generic routines for creating new ones. It will be a poor-man's Object-Oriented Processor (see Baxter, 1994). The anchor for bits, available in the *newclude* package, has the following syntax:

```
\includebit{<bit name>}{<file name>}
```

And a bit is tagged as follows:

```
\begin{<bit type>}{<bit name>}{<init code>}
...
\end{<bit type>}
```

For example, the command

```
\includebit{Ode to a Lion}{safari}
```

would include the bit that is declared in the file `safari.tex` with

```
\begin{poem}{Ode to a Lion}
{\numberstanzas
\newcommand\growl{\large Gr"owl!}}
```

The new interface to `\input` provided by the *newclude* package implements two new basic commands, a command like `\include` but without the enclosing `\clearpages`, and a command to include

a delimited section of a disk file. The omission of the `\clearpages` is achieved by writing out a single `aux` file per document (eliminating the additional ones for each file included with `\include`). None of the features of the old `\include` command are lost. The task of including only a delimited section of an included file is accomplished by generalizing the *verbatim* package so that all irrelevant parts of the file can be ignored.

The success of the system so far is promising. It is now possible with L^AT_EX to create a document whose contents are L^AT_EX environments distributed among an arbitrary number of disk files. In this case these files are used as auxiliary sources, but nothing prevents them from being principal source files that can be processed independently by L^AT_EX. That is, they may have the usual `\documentclass` declaration and `document` environment or not, it is irrelevant.

One can imagine the following application: a teacher's 25 students turn in (or, better, make available over a network) 25 L^AT_EX sources containing three book reviews each. The teacher can prepare a document consisting of all the reports on a single book, or a portfolio of the best reviews of each student, or a snapshot of their work in progress. It remains possible for the students to format their sources in their own preferred manner, completely independent of the formatting the teacher chooses for the composite documents. Since the sources for each are the same, version control does not involve more than the usual task of keeping document images up to date with principal sources.

Further applications suggest themselves readily. A lecturer can extract derivations or figures or abstracts or quotations from scholarly papers and incorporate them into slide presentations. If any of the source material is revised in the future, it need only be revised in one place. Selected parts of a L^AT_EX source can be exported and published on the World Wide Web using `latex2html`. This is already possible using that program's conditional inclusion facilities, but the bit solution allows the same source to export different parts of itself to different web documents without need to alter it (see Goossens and Saarela, 1995b).

The limitations encountered in this system have prompted theoretical contemplation of more powerful improvements. The shortcomings include, notably, the continuing dependence of bit names on disk file names. The method of including regions of files accepts only a strict *verbatim* syntax and *verbatim* matches for the delimiters. T_EX inserts `\par` at

an `\input`, so that bit boundaries must correspond with paragraph boundaries.²

These are significant limitations, and the way forward in L^AT_EX is difficult. The *verbatim* and *doc* packages are evidence of how complex is the task of getting T_EX to perform the offices of even a simple inflexible text stream editor. Moreover, it seems foolhardy to attempt to emulate in L^AT_EX what can be done in UNIX with minimal effort, and probably with no more effort on other T_EX platforms. I believe these experiments in L^AT_EX are going to be useful, but the most satisfactory way forward must be toward one of the solutions suggested at the end of the previous section.

The implementation of block modularity

I observed above that block elements are specified in a variety of ways. Some (though not all) of the differences can be justified by an appeal to natural user syntax. It would certainly be inconvenient most of the time, for example, to have to type

```
\beginparagraph
...
\endparagraph.
```

The differences present a problem, however, to the L^AT_EX developer for whom, as a programmer, standards are always an advantage.

T_EX's internals confront us with basic differences between those units handled by an `every*` token variable (paragraphs, lines, etc.), those handled by T_EX's grouping mechanism (environments, `\items`), and those handled by macro arguments (e.g., `\emph`). A good programmer interface would hide these differences as much as possible.

The number and format of optional and mandatory arguments to environments is nonstandard. In the present L^AT_EX environment, one needs to rely on a convention, such as the syntax given above for bits, the syntax suggested by Baxter (1994) of a series of command sequences each taking a single mandatory argument, or a single argument parsed by the *keyval* package.

The `document` "environment" looks like an environment and should be parallel to one. The preamble is just a special kind of argument that is serving to instantiate a block element, a bit of type "document".

The `list` environment is an attempt to provide a standard programmer interface to defining a

² This behavior seems to be another good candidate for a change to T_EX.

block element made up of a single kind of subelement separated by dividing commands. This is the right idea and can profitably be made more general: the inheritance (nesting) of lists is ad hoc, and only one kind of subelement is allowed (that is, `\items`). In typesetting a play, for example, you would like at least two kinds of subelements: speeches and stage directions. This of course could be implemented using environments, but there are many situations in which using dividing commands rather than enclosing commands is preferable (e.g., ease of use, importing or converting source material).

Intra-package modularity

While developing the `Frankenstein` system, I ran into an interesting problem which led to the idea of *intra-package modularity*. I had a large package which accomplished a number of things that I preferred to use in constellation but others were going to use singly or in arbitrary combinations. The problem was to find a system that allowed me to share my code with the \LaTeX community in a way that both provided efficient code and allowed me to maintain the code easily. Using the vocabulary introduced in this paper, the problem was how to get a single disk file to efficiently provide more than one feature.

If I broke up my large package into a number of smaller independent packages, then each package would be efficient when used alone but I would have to maintain several different packages that shared common code and documentation. In some cases, the shared code was so brief that it seemed inefficient and confusing to separate it into its own disk file. There would also be an efficiency problem with \TeX 's resources such as command names and counters, since each package would have to reserve its own resources. A way was desired to let this group of packages share resources, while preventing conflicts with other packages.

With a tool like `noweb` (see Bzyl, 1995), the `doc` package and `docstrip` program, or an implementation of features as discussed above, a macro written only once in the source can end up in any number of extracted packages. None of the standard defining commands, however, are suitable for defining such a macro. If `\def` is used, all the extracted packages are vulnerable to name conflicts with other packages (not to mention self-conflict during development). But `\newcommand` is also wrong because then the extracted packages could not be used together—the second package to define the command would fail. If `\renewcommand` were used, the first

would fail. The command `\providecommand`, which defines a macro only if it is not already defined, was added to $\text{\LaTeX} 2\epsilon$. But using this, one assumes that if the macro was already defined it has an acceptable definition. This level of checking might be sufficient in some circumstances, but a command is desired that *guarantees* a macro will subsequently have a particular definition. To this end I define `\guaranteecommand` which calls `\newcommand` if its first argument is undefined and `\CheckCommand` if it is already defined.

In the `Frankenstein` source, any macro which will end up in more than one package, but which I do not want to install into a separate package required by the others, is defined using `\guaranteecommand`. The definition of `\guaranteecommand` occurs in the `safedefs` package, which all the other packages require (though it is not hard to bootstrap this one command, to get it to guarantee itself).

Disk files and copyrights

The \LaTeX developers have put a lot of effort into making it easy to create sources which reliably produce identical document descriptions (identical `dvi` files) at different sites. Such uniformity in the documents derived from a single source is very important in many situations, especially when a longer document is assembled from multiple sources.

One of the ways in which the developers have sought to establish and maintain this standard is by placing conditions on the distribution of the \LaTeX system files that require disk file names to serve as unique labels for segments of code. Because commands like `\documentclass` and `\usepackage` cause \LaTeX to load files with particular names, there is one and only one (legally-produced) document description that can be generated from a source which invokes standard \LaTeX classes and packages.

No one doubts the usefulness of a good standard for deriving documents from sources, but this standard has been achieved at the cost of abstraction. The unique labels which serve to establish the standard should not of necessity coincide with the labels by which a feature or document class is identified in the source. It should never be necessary to alter a source to derive different document descriptions from it. Altering sources is time-consuming. It causes timestamps to be updated and confuses version control systems. It can introduce errors, and it encourages a proliferation of not-quite-identical copies.

On the other hand, it should always remain obvious how to generate the standard document from

a source, so that standard documents are generally available and convenient.

In response to these two concerns, the author has developed the AL^AT_EX system. The *A* may be understood to stand for *alternate* or *abstract*, or to be the indefinite article, which emphasizes that fact that documents derived from sources with AL^AT_EX are just one of an indefinite number of possibilities.

AL^AT_EX is a simple system. It is a slightly-modified clone of the standard L^AT_EX format, which when used as distributed behaves exactly like L^AT_EX, except that it displays its own identification banner. (Only in the most perverse situations will the two formats not produce identical output.) Internally, the behavior is not quite the same. The `\documentclass` command in AL^AT_EX parses its arguments and passes them to a file of code called `metaclass.cfg`. As its name implies, the file is a meta-class which determines how the class specification in the source should be interpreted. And just as important, the file may be altered and distributed with no restrictions. The meta-class distributed with AL^AT_EX emulates L^AT_EX's behavior, but this can easily be overridden by changing the file, or putting a different file with the same name earlier in T_EX's search path. Code that enables either of two convenient mechanisms of overriding are provided in comments. Using one of these sample mechanisms restores a useful level of abstraction to L^AT_EX sources, because if full abstract control is available at the first line of the source, it is available for the whole source.

Because it is very difficult to have a working AL^AT_EX without also having a working L^AT_EX, no one is likely to find it inconvenient to create, from the same source, either a standard L^AT_EX dvi file (by invoking the L^AT_EX format) to facilitate seamless exchange of sources, or a not-standard-L^AT_EX dvi file (by invoking the AL^AT_EX format with an appropriate meta-class).

It should be emphasized that there is no reason at all to *compose* sources while previewing with AL^AT_EX. Doing so could compromise the portability of the source if a strange meta-class is used. AL^AT_EX is useful only to change the look of already-existing sources from the standard appearance to a nonstandard appearance. Even in this case, using AL^AT_EX is not always necessary, since it may be possible to legally modify the appropriate style files, or not too inconvenient to modify the source.

Conclusion

I have argued for the advantages of several kinds of modularity in the L^AT_EX user and developer environments that do not presently exist to a satisfactory degree. A primary difficulty not resolved is the choice of the domain in which to improve modularity. Are the *Frankenstein*, *newclude*, and AL^AT_EX solutions adequate? If not, should solutions be effected in L^AT_EX, in L^AT_EX3, at the back end of T_EX in platform-dependent T_EX distributions, at the front end of L^AT_EX in platform-dependent integrated L^AT_EX user and developer environments, in extensions to T_EX (ϵ -T_EX), or in enhancements to T_EX ($\mathcal{N}\mathcal{T}\mathcal{S}$)? In any case I hope I have won interest and enthusiasm for discussing and working on changes in a certain direction.

The *Frankenstein* system, the *safedefs* and *newclude* packages, and the AL^AT_EX format should be available on CTAN by the time of publication, if they are not in the meantime adopted in some form into the standard L^AT_EX distribution.

References

- NTS. "Frequently asked questions of NTS-L". 1995. 5th edition, available as CTAN:`info/nts-faq`, maintained by Jörg Knappen <`knappen@vkpmzd.kph.uni-mainz.de`>.
- W. E. Baxter. "An object-oriented programming system in T_EX". *TUGboat* **15**(3), 331–338, 1994.
- W. Bzyl. "Literate plain source is available!". *TUGboat* **16**(3), 297–299, 1995.
- M. Goossens and Saarela, Janne. "A practical introduction to SGML". *TUGboat* **16**(2), 103–145, 1995a.
- M. Goossens and Saarela, Janne. "T_EX to HTML and back". *TUGboat* **16**(2), 174–214, 1995b.
- A. Ogawa. "Object-oriented programming, descriptive markup, and T_EX". *TUGboat* **15**(3), 325–330, 1994.
- T. V. Raman. "An Audio View of (L^A)T_EX Documents — Part II". *TUGboat* **16**(3), 310–314, 1995.

A Multienumerate Package

Dennis Kletzing

Stetson University

DeLand FL 32720

USA

Email: kletzing@bliss.stetson.edu

Abstract

The `multienum.sty` package allows the user to produce an enumerated array of multiple columns, each vertically aligned on the counter. An optional argument provides for consecutive numbering of the array items, or an even-only or odd-only numbering scheme.

Introduction

Typesetting the solutions manual for a text usually involves creating an enumerated list involving many short answers. Typically these are set with several items per line, with no attempt made to vertically align the exercise numbers. This article describes a package, `multienum.sty`, which provides an environment, `multienumerate`, that produces an enumerated array in which columns are vertically aligned on the counter. If the user wishes, the enumeration counter can be changed to give a list of even-only numbers or odd-only numbers.

1. Not	2. Linear	3. Not	
4. Quadratic	5. Not	6. Linear	
7. No; if $x = 3$, then $y = -2$.	8. $x = 2$		
9. $(x_1, x_2) = (2 + \frac{1}{3}t, t)$ or $(s, 3s - 6)$			
10. $y = 7$	11. $x + y = 3$ and $z = 1$		
12. $(2, -1, 3)$	13. None	14. $(2, 1, 0, 1)$	
15. 2	16. 3	17. 4	18. 5
19. $(0, 0)$ and $(0, 1)$	20. If $x = 1$, $y = -2$.		
21. $(10, 11, 0, 0)$	22. $(0, -1, 0, -5)$		

Table 1: An enumerated array of solutions

What the package does

Table 1 shows a typical enumerated array. The second entry in the third row is left blank since we want the first item to expand into its space. To get the vertical alignment of the counter in column 3, we set row 3 as three entries, but left the second entry blank thus giving more space for the first entry. To produce this array, we typed the following:

```

\begin{multienumerate}
\mitemxxx{Not}{Linear}{Not}
\mitemxxx{Quadratic}{Not}{Linear}
\mitemxox{No; if $x=3$,
  then $y=-2$.}{$x=2$}
\mitemx{$(x_1,x_2)=(2+\frac{1}{3}t,t)$ or
$(s,3s-6)$}
\mitemxoxo{$y=7$}{$x+y=3$ and $z=1$}
\mitemxxx{$(2,-1,3)$}{None}{$(2,1,0,1)$}
\mitemxxxx{2}{3}{4}{5}
\mitemxx{$(0,0)$ and $(0,1)$}{If $x=1$,
  $y=-2$.}
\mitemxx{$(10,11,0,0)$}{$(0,-1,0,-5)$}
\end{multienumerate}

```

The environment `multienumerate` has an optional argument¹ for enumerating even-only or odd-only arrays.

- `\begin{multienumerate} ... \end{multienumerate}`
produces a consecutively enumerated array
- `\begin{multienumerate}[evenlist] ... \end{multienumerate}`
produces an enumerated array using only even numbers
- `\begin{multienumerate}[oddlist] ... \end{multienumerate}`
produces an enumerated array using only odd numbers

Using the package

Each row of the enumerated array is set using one of nine commands:

¹ The optional argument works only with L^AT_EX2e.

<code>\mitemx</code>	A single item in the row.
<code>\mitemxx</code>	Two items in the row.
<code>\mitemxxx</code>	Three items in the row.
<code>\mitemxox</code>	Three items in the row with the center item left blank so the first item can extend into its space.
<code>\mitemxxo</code>	Three items in the row, the last item left blank so the second item can extend into its space.
<code>\mitemxxxx</code>	Four items in the row.
<code>\mitemxoxx</code>	Four items in the row, the second space left blank so the first item can extend into its space.
<code>\mitemxxox</code>	Four items in the row, the third space left blank so the second item can extend into its space.
<code>\mitemxxxxo</code>	Four items in the row, the last space left blank so the third item can extend into its space.

For example, `\mitemxxx{a}{b}{c}` sets the entries *a*, *b*, and *c* equally spaced across the row, while `\mitemxox{a}{c}` sets the two items, *a* and *c*, across the row *as if* there were three items, leaving the second entry blank so that the first entry can extend into its space; and `\mitemxxo{a}{b}` sets the two items, *a* and *b*, *as if* there were three items, but leaves the space for the third item blank, allowing the second entry to extend into its space.

A convenient way to use the multienumerate package is with a two column layout using multicols. Figure 2 at the end of the article illustrates several possibilities.

A disadvantage of the package is that the user must choose how to typeset each line in the array rather than letting TeX decide how to do it. This creates a lot of overhead in the macro since separate commands are needed for each possibility. It is not difficult to write a macro that will let TeX decide how many items to set on each line. While this approach is more efficient, especially if one changes the entries, it does not always give the visual appearance the user may want.

How the package works

We describe how the package typesets a line containing two items. The other situations are similar.

Figure 1 shows two items on a line, each consisting of a label box of width $lw=\text{\labelwidth}$, a label separation of width $ls=\text{\labelsep}$, and a box containing the entry itself (set `\raggedright`) of width $.5rxx=.5\text{\remainxx}$. The length `\remainxx` is the total space remaining after two label widths

and two label separations have been removed; thus, it is the amount of space available for typesetting the two entries, each in a box whose width is one-half `\remainxx`. Since the total width of the line is `\hsize`, it follows that

$$\underbrace{2\text{\labelwidth} + 2\text{\labelsep} + \text{\remainxx}}_{\text{\usedxx}} = \text{\hsize}$$

and therefore

$$\text{\remainxx} = \text{\hsize} - \text{\usedxx}$$

In this way the the width of the box is calculated when two items are typeset on the line.

The macro

A somewhat trimmed copy of the package follows (`\newlength` and `\newcounter` declarations, higher levels of multienumerate list nesting, and four-across items are omitted). The full source of the package may be obtained by anonymous ftp from CTAN macros/latex/contrib/other/misc/multienum.sty, or by email from the author.

```
%Create multiple item styles
\newcommand{\labelname}{%
  \csname labelenum\romannumeral
  \themultienumdepth\endcsname}
\newcommand{\itemx}[1]{\parbox[t]{%
  {\labelwidth}{\hfill{\labelname}}%
  \hskip\labelsep%
  \parbox[t]{\remainx}{\raggedright #1}%
  \smallskip}
\newcommand{\itemxx}[2]{\parbox[t]{%
  {\labelwidth}{\hfill{\labelname}}%
  \hskip\labelsep\parbox[t]{%
  {.5\remainxx}{\raggedright #1}
  \hfill\parbox[t]{%
  {\labelwidth}{\hfill{\labelname}}%
  \hskip\labelsep\parbox[t]{%
  {0.5\remainxx}{\raggedright #2}
  \smallskip}
\newcommand{\itemxxx}[3]{\parbox[t]{%
  {\labelwidth}{\hfill{\labelname}}%
  \hskip\labelsep\parbox[t]{%
  {.3333\remainxxx}{\raggedright #1}
  \hfill\parbox[t]{%
```

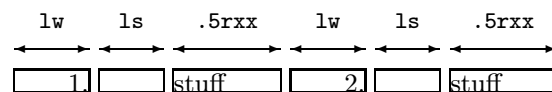


Figure 1: One line containing two items

```

    {\labelwidth}{\hfill {\labelname}}
\hskip\labelsep\parbox[t]{%
  {0.3333\remainxxx}{\raggedright #2}
\hfill\parbox[t]{%
  {\labelwidth}{\hfill{\labelname}}
\hskip\labelsep\parbox[t]{%
  {0.3333\remainxxx}{\raggedright #3}
\smallskip}
\newcommand{\itemxox}[2]{\parbox[t]{%
  {\labelwidth}{\hfill{\labelname}}
\hskip\labelsep\parbox[t]{%
  {\remainxox}{\raggedright #1}
\hfill\parbox[t]{%
  {\labelwidth}{\hfill{\labelname}}
\hskip\labelsep\parbox[t]{%
  {0.3333\remainxxx}{\raggedright #2}
\smallskip}
\newcommand{\itemxox}[2]{\parbox[t]{%
  {\labelwidth}{\hfill{\labelname}}
\hskip\labelsep\parbox[t]{%
  {0.3333\remainxxx}{\raggedright #1}
\hfill\parbox[t]{%
  {\labelwidth}{\hfill{\labelname}}
\hskip\labelsep\parbox[t]{%
  {\remainxox}{\raggedright #2}
\smallskip}
%Define counter options
\newcommand{\oddlisti}{%
  \setcounter{multienumi}{-1}%
  \renewcommand{\labelenumi}{%
    {\ifodd\value{multienumi}%
      \addtocounter{multienumi}{2}%
      \themultienumi.\else
      \addtocounter{multienumi}{1}%
      \themultienumi.%
      \addtocounter{multienumi}{-2}\fi}}
\newcommand{\evenlisti}{%
  \setcounter{multienumi}{0}
  \renewcommand{\labelenumi}{%
    {\ifodd\value{multienumi}%
      \addtocounter{multienumi}{1}%
      \themultienumi.%
      \addtocounter{multienumi}{-2}\else
      \addtocounter{multienumi}{2}%
      \themultienumi.\fi}}
\newcommand{\regularlisti}{%
  \setcounter{multienumi}{0}%
  \renewcommand{\labelenumi}{%
    {\addtocounter{multienumi}{1}%
      \arabic{multienumi}.}}
\newcommand{\listtype}[1]{#1}
\newcommand{\mitemx}[1]{%
  \item[] \itemx{#1}}
\newcommand{\mitemxx}[2]{%
  \item[] \itemxx{#1}{#2}}
\newcommand{\mitemxxx}[3]{%
  \item[] \itemxxx{#1}{#2}{#3}}
\newcommand{\mitemxox}[2]{%
  \item[] \itemxox{#1}{#2}}
\newcommand{\mitemxox}[2]{%
  \item[] \itemxox{#1}{#2}}
%Define the multienumerate environment
\newenvironment{multienumerate}%
  [1][regularlist]{%
  \ifnum \themultienumdepth >3
  \@toodeep\else
  \addtocounter{multienumdepth}{1}
  \edef\@multienumctr{%
    multienum\romannumeral%
    \themultienumdepth}
  {\csname label\@multienumctr%
    \endcsname}{%
    \usecounter{\@multienumctr}}%
  \listtype{\csname#1\romannumeral
    \themultienumdepth\endcsname}\fi
\begin{list}}{-%
  \ifnum\themultienumdepth=2
  \setlength{\leftmargin}{23pt} \else
  \setlength{\leftmargin}{0pt} \fi%
  \setlength{\labelwidth}{18pt}
  \setlength{\labelsep}{5pt}%
  \setlength{\usedx}{\labelwidth}%
  \addtolength{\usedx}{\labelsep}%
  \addtolength{\usedx}{\leftmargin}%
  \setlength{\remainx}{\hspace}%
  \addtolength{\remainx}{-\usedx}%
  \setlength{\usedxx}{2\labelwidth}%
  \addtolength{\usedxx}{2\labelsep}%
  \addtolength{\usedxx}{\leftmargin}%
  \setlength{\remainxx}{\hspace}%
  \addtolength{\remainxx}{-\usedxx}%
  \setlength{\usedxxx}{3\labelwidth}%
  \addtolength{\usedxxx}{3\labelsep}%
  \addtolength{\usedxxx}{\leftmargin}%
  \setlength{\remainxxx}{\hspace}%
  \setlength{\remainxox}{.666\remainxxx}%
  \addtolength{\remainxox}{\labelwidth}%
  \addtolength{\remainxox}{\labelsep}%
  \setlength{\itemindent}{0pt}}{-%
  \addtocounter{multienumdepth}{-1}%
\end{list}}

```

Answers to Even Exercises**Chapter 1****Section 1**

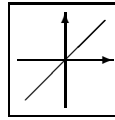
- | | | |
|------------------------|--------------|-------------|
| 2. 2 | 4. 5 | 6. -4 |
| 8. $x = 1$ | 10. $y = -7$ | 12. $z = 3$ |
| 14. $x^2 - 3x + 7 = 0$ | | 16. Yes |
| 18. 2 | 20. 5 | 22. -4 |
| 24. $x = 1$ | 26. $y = -7$ | 28. $z = 3$ |
| 30. $x^2 - 3x + 7 = 0$ | | 32. Yes |
| 34. 2 | 36. 5 | 38. -4 |
| 40. $x = 1$ | 42. $y = -7$ | 44. $z = 3$ |
| 46. $x^2 - 3x + 7 = 0$ | | 48. Yes |
| 50. 2 | 52. 5 | 54. -4 |
| 56. $x = 1$ | 58. $y = -7$ | 60. $z = 3$ |
| 62. 2 | 64. 5 | 66. -4 |
| 68. $x = 1$ | 70. $y = -7$ | 72. $z = 3$ |
| 74. 2 | 76. 5 | 78. -4 |

Section 2

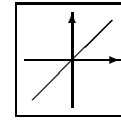
- | | |
|--|--------------------------|
| 2. Yes | 4. $3x^2 + x = -2$ |
| 6. If $x = 1$, the only solution is $y = 3$. | |
| 8. $x = 1$ | 10. $y = -7$ 12. $z = 3$ |
| 14. Yes | 16. No 18. No |
| 20. 2 | 22. 5 24. -4 |
| 26. $x^2 - 3x + 7 = 0$ | 28. Yes |

- | | | |
|------------------------|--------------|-------------|
| 30. 2 | 32. 5 | 34. -4 |
| 36. $x = 1$ | 38. $y = -7$ | 40. $z = 3$ |
| 42. $x^2 - 3x + 7 = 0$ | | 44. Yes |
| 46. 2 | 48. 5 | 50. -4 |

52.



54.



56. Billy should sell 3 red marbles, buy 2 white marbles, and keep the rest.
58. Sarah should buy 2 pounds of squash, 3 pounds of potatoes, and 4 pounds of fish.

Answers to Odd Exercises**Chapter 2****Section 1**

- | | | | |
|-------------------|--------------------|-----------------------|-------------|
| 1. Yes | 3. No | 5. No | 7. 3 |
| 9. 5 | 11. 2 | 13. -8 | 15. 7 |
| 17. $2x - 3y = 6$ | | 19. 1 | 21. -2 |
| 23. 2 | 25. 5 | 27. $3x^2 - 2y^2 = 1$ | |
| 29. $y = 3$ | 31. -5 | 33. $x = 9$ | 35. $y = 1$ |
| 37. 7 | 39. $2x + 3y = -5$ | 41. 6 | |
| 43. 7 | 45. 3 | 47. 1 | 49. 6 |

Quiz #1. Circle the correct answer.

- Which of the following numbers is a solution of the equation $2x + 5 = 9$:
a. 0 b. 1 c. 2 d. 3
- Which of the following numbers is a solution of the equation $x^2 + 5 = 9$:
a. 0 b. -1 c. -2 d. -3
- Which of the following expressions is equal to $x^2 - y^2$:
a. $(x - y)^2$ b. $(x + y)^2$
c. $(x - y)(x + y)$ d. 0
- The graph of the equation $3x^2 - 2y = 0$ is a:
a. circle b. parabola
c. ellipse d. line
- If $x = 2$, then the value of $x^3 - x + 3$ is:
a. 2 b. 6 c. None of these.
- Which of the following statements correctly expresses the meaning of the algebraic expression $2(x + y) = 6$:
a. twice the value of x added to twice the value of y is equal to 6 b. twice the sum of x and y is equal to 6
- Evaluate the expression $2 - [3 + (5 - 9)] + -3$.
a. 6 b. -1 c. 4 d. 0
- Evaluate the expression $5 + [3 - (1 + -2)]$.
a. 5 b. -1 c. 9 d. 1

Figure 2: Samples typeset using `multienum.sty`

Hyphenation in T_EX — Quo Vadis?*

Petr Sojka

Faculty of Informatics, Masaryk University
Burešova 20, 60200 Brno
Czech Republic
Email: `sojka@muni.cz`

Pavel Ševeček

Faculty of Informatics, Masaryk University
Burešova 20, 60200 Brno
Czech Republic
Email: `pavel@muni.cz`

Abstract

Significant progress has been made in the hyphenation ability of T_EX since its first version in 1978. However, in practice, we still face problems in many languages such as Czech, German, Swedish etc. when trying to adopt local typesetting industry standards.

In this paper we discuss problems of hyphenation in multilingual documents in general, we show how we've made Czech and Slovak hyphenation patterns, and we describe our results achieved using the program PATGEN for hyphenation pattern generation. We show that hyphenation of compound words may be partially solved even within the scope of T_EX82. We discuss possible enhancements of the process of hyphenation pattern generation and describe features that might be reasonable to think about to be incorporated in Ω or another successor to T_EX82.

Motivation

*“Go forth and make masterpieces
of hyphenation patterns . . .”
(Haralambous, 1994)*

Editors' and publishers' typographical requirements for camera-ready prepared documents are growing. To meet some of their requirements in T_EX, especially when typesetting in narrow columns, one needs perfect hyphenation patterns in order to find almost all permissible hyphenation points.

When making Czech hyphenation patterns and typesetting multilingual documents we encountered some problems with achieving quality hyphenation and decent-looking documents with T_EX. This work has led to our ideas about possible remedies and future extensions in a successor to T_EX.

Our paper consists of three parts. In the first part we try to summarize the developments that have been made on the issue since T_EX's birth.

In the second, we describe our attempts to create Czech and Slovak hyphenation patterns and summarize hints and suggestions for PATGEN users.

In the third part we discuss possible improvements that might take place in a T_EX successor (Ω , ε -T_EX or New Typesetting System ($\mathcal{N}\mathcal{T}\mathcal{S}$)).

The hyphenation story

Let's review the developments in hyphenation in T_EX that have been made so far.

English In T_EX78 a rule-driven algorithm for English was built-in by Liang and Knuth. Their algorithm found 40 % of the allowable hyphens, with about 1 % error (Liang, 1981). Although authors claimed that these results are “quite good”, Liang continued working on the generalization of the idea of rules expressed by hyphenating and inhibiting patterns. In his dissertation (Liang, 1983) he describes a method, which is used in T_EX82, based on the generalization of the prefix, suffix and the vowel-consonant-consonant-vowel rules. He wrote (in WEB) the program PATGEN (Liang and Breitenlohner, 1991) to automate the process of pattern generation from a set of already hyphenated words.

* Reprinted with corrections from EuroT_EX' 94 proceedings, Gdańsk, pp. 59–68 with permission.

He started with the 1966 edition of Webster’s Pocket Dictionary that included hyphenated words and inflections (about 50 000 entries in total). In the early stages, testing the algorithm on a 115 000 word dictionary from the publisher, 10 000 errors in words not occurring in the pocket dictionary were found. “Most of these were specialized technical terms that we decided not to worry about, but a few hundred were embarrassing enough that we decided to add them to the word list.” (Liang, 1983, p. 30). He reports the following figures: 89,3 % permissible hyphens found in the *input* word-list with 4447 patterns with 14 exceptions.

Liang’s method is described by Knuth (1986b, Appendix H) and was later adopted in many programs such as `troff` (Emerson and Paulsell, 1987) and `Lout`, and in localizations of today’s WYSIWYG DTP systems such as QuarkXPress, Ventura, etc. Although specialized dictionaries such as Allen’s (1990) by Oxford University Press separate possible word-division points into at least two categories (preferred and less recommended), we have not seen any program that incorporates the possibility of taking into account these classes of hyphenation points so far.

Those other languages

“... *patterns are supposed to be prepared by experts who are paid well for their expertise.*”
(Knuth, 1986b, p. 453, 8th printing)

The first version of T_EX82 allowed only one set of patterns to be loaded at a time. Thus it was not possible to typeset multilingual documents with correct hyphenation in all languages and this limitation was quite unsatisfactory. Already in 1985, two attempts to solve the problem were made:

Multilingual T_EX: Extensions, most of which afterwards Knuth adopted in T_EX 3.x were suggested and implemented by Ferguson (1985). A new primitive `\language`¹ was introduced for switching between several sets of `\patterns` and hyphenation exceptions. A new `\charsubdef` primitive is still used in today’s 8-bit implementations of T_EX. Full details are give by Ferguson (1988).

l^ST_EX: Barth and Nirschl (1985) presented an approach on achieving decent hyphenation in German texts under the name S^IT_EX, or in its interactive version under the name l^ST_EX. Their method, (available as a change file for UNIX T_EX from `eiunix.tuwien.ac.at`) has been used

in Germany for years and is being improved (Barth and Steiner, 1992; Barth, Steiner, and Herbeck, 1993). This approach has been proposed for inclusion in $\mathcal{N}\mathcal{T}\mathcal{S}$ (NTS-L, 1992–).

S^IT_EX (l^ST_EX for the interactive version) introduces a new primitive `\nebenpenalty` which allows differentiation between main (compound word boundaries) and secondary (word stem) hyphenation points.

A new notation for hyphenation patterns is introduced and a hyphenation algorithm for German is hardwired into the program. The tables for the algorithm, file `sihyphen.tex` (60K) are written manually and can be simply edited and enriched. However, no provision for the generation of these patterns from a word-list (such as the PATGEN program) is offered.

During the last 15 years almost every year there appeared a paper in *TUGboat* reporting new patterns for some language (see table 1). Another couple of hyphenation patterns, fonts and preprocessors are available in ScholarT_EX² (Haralambous, 1991).

Although Don Knuth introduced the new primitives `\language` and `\setlanguage` for switching between several sets of hyphenation patterns in T_EX 3.0, there are indications that not all of the related problems have been solved and further investigations are necessary (Fanton, 1991).

Proposals on how to customize T_EX for a new language were suggested by Partl (1990). New tools to simplify the generation of 8-bit (virtual) fonts were designed — `fontinst` (Jeffrey, 1993) and `accents` (Zlatuška, 1991). A macro package for simple language switching, `babel` (Braams, 1991a; Braams, 1991b; Braams, 1993), was produced to simplify typesetting of multilingual documents. An international version of the `Makeindex` program was written (Schrod, 1991). The DC fonts (Ferguson, 1990; Haralambous, 1992a; Haralambous, 1993a), designed to permit hyphenation in many languages, are now being widely distributed, forced by the new L^AT_EX wave. Compliance with the suggestions of the working group TWGMLC³ (Haralambous, 1992a) could help too (naming conventions for hyphenation files, etc.). Multilingual document aspects of typesetting are being collected in the scope of L^AT_EX3 project in (Gaulle, 1994), where a nice collection of language-related T_EX primitives can be found, together with definitions of the terminology used.

² ScholarT_EX is a registered trademark of Yannis Haralambous

³ T_EXnical Working Group on Multiple Language Coordination

¹ A rather misleading name, as it deals with only one particular feature of a language — hyphenation — which feature is of only limited interest to linguists.

Table 1: Hyphenation patterns for \TeX with PATGEN statistics for various languages

language	trie	ops	done by	#patt	size	author (& reference)
BG (Bulgarian)	688	56	hand	263	1672	Ognyan Tonev/90
CA (Catalan)	661	11	hand	826	6136	Goncal Badenes, Francina Turon/91
CY (Welsh)	8552	143	PATGEN	6728	43162	Yannis Haralambous (Haralambous, 1993b)
CZ ₁ (Czech)	3676	90	hand	4479	25710	Ladislav Lhotka (1991)
CZ ₂	5302	67	PATGEN	4196	23474	Pavel Ševeček (Sojka and Ševeček, 1994)
DE _{min} (German)	6099	170	PATGEN	4066	25660	Norbert Schwarz/88
DE _{max}	9980	255	PATGEN	7007	45720	Norbert Schwarz/88
DE (v3.1)	8375	207	PATGEN	5719	39251	Norbert Schwarz, Bernd Raichle/94 (Schulze, 1984; Partl, 1988; Breitenlohner, 1988; Obermiller, 1991; Kopka, 1991)
DK (Danish)	1815	60	PATGEN	1145	6411	Frank Jensen/92
EL (Mod. Greek)	1278	23	hand	1616	8786	Yannis Haralambous/92
EO (Esperanto)	4895	143	PATGEN	4118	23224	Derk Ederveen/93
ES (Spanish)	1106	29	hand	578	4609	Francesc Carmona/93
ET (Estonian)	2054	45	PATGEN	1267	7976	Enn Saar/92
FI (Finnish)	583	27	hand	232	1342	Kauko Saarinen/92, (Saarinen, 1988)
FR (French)	1634	86	comb.	917	30022	Jacques Désarménien (1984), Daniel Flipo, Bernard Gaulle et al./84–94
Ancient Greek			hand			Yannis Haralambous (Haralambous, 1992b)
HR (Croatian)	1471	46	hand	916	7250	Cvetana Krstev/93
HY (Armenian)						Yannis Haralambous (in Scholar \TeX)
IS (Icelandic)	5477	145	PATGEN	4187	29919	Jorgen Pind/87
IT (Italian)	1327	15	hand	729	4255	Salvatore Filippone/92 (Canzii, Genolini, and Lucarella, 1984)
IT (Italian)	529	37	hand	210	2571	Claudio Beccari/93 (Beccari, 1992)
Latin			hand			Yannis Haralambous (1992b)
Modern Latin			hand			Claudio Beccari (1992)
LT (Lithuanian)	2169	77	PATGEN	1546	9639	Vitautas Statulevicius & Yannis Haralambous/92
NL ₁ (Dutch)	7824	124	PATGEN	6105	37997	CELEX/89
NL ₂	10338	187	PATGEN	7928	50969	CELEX/89
NL ₃	520	24	hand	326	1732	Peter Vanroose
NO (Norwegian)	3669	220	PATGEN	2371	15589	Ivar Aavatsmark/92
PL (Polish)	4954	194	hand	4053	28907	Hanna Kołodziejska (1987, 1988)
PT (Portuguese)	374	10	hand	126	534	Pedro J. de Rezende (1987)
RU (Russian)	4599	92	hand	4121	29272	Dimitri Vulis (Vulis, 1989; Malyshev, Samarin, and Vulis, 1991a; Malyshev, Samarin, and Vulis, 1991b; Samarin and Urvantsev, 1991)
SK (Slovak)	3600	248	hand	2569	22628	Jana Chlebíkova/92
SK	7606	78	PATGEN	6137	35623	Pavel Ševeček (Sojka and Ševeček, 1994)
SR (Serbian)	1475	40	hand	896	6890	Cvetana Krstev/89 (Krstev, 1991)
SV (Swedish)	5269	125	PATGEN	3733	23821	Jan Michael Rynning/91
TR (Turkish)	678	16	hand	1834	9580	Pierre A. MacKay (1988)
UK (UK English)	10995	224	PATGEN	8527	54769	Dominik Wujastyk/93
US (US English)	6075	181	PATGEN	4447	27302	Frank Liang/82 (Liang, 1983)
US	6661	229	PATGEN	4810	30141	G.D.C. Kuiken (1990)

Exception logs

“If any computer center decides to preload different exceptions from those in plain T_EX (i.e., in the file *HYPHEN.TEX*), the changed exceptions should not under any circumstances be put into *HYPHEN.TEX* or *PLAIN.TEX*. All local changes should go into a separate file, so that T_EX will still produce identical results on all machines. In fact, I recommend not preloading those changes, but rather assuming that individual users will have their own favorite collection of updates to the standard format files.”
(Knuth, 1983)

The exception log and corrections for US English hyphenation have been reported several times – (e.g. Thulin, 1987; Beeton, 1989; Kuiken, 1990; Beeton, 1992), as shown in table 2. These listings are published in accordance with DEK’s wish (Knuth, 1983). Only words with *wrongly* placed hyphenation points are listed, not those where T_EX finds only a subset of possible breakpoints.

Table 2: Growing number of exceptions for *hyphen.tex*

# of exceptions	where reported
14	(Liang, 1983)
24	(Beeton, 1984, <i>TUGboat</i> 5, no. 1)
88	(Beeton, 1985, <i>TUGboat</i> 6, no. 3)
127	(Beeton, 1986, <i>TUGboat</i> 7, no. 3)
129	(Thulin, 1987, <i>TUGboat</i> 8, no. 1)
501	(Beeton, 1989, <i>TUGboat</i> 10, no. 3)
543	(Beeton, 1992, <i>TUGboat</i> 13, no. 4)

This shows that significant care and effort is still needed and *is being gradually spent* on the checking of hyphenation points during proof-reading and that the standard US patterns are not sufficient to satisfy current needs. Additional sets of patterns (2 versions – *ushyphen.add* and *ushyphen.max*) have been generated by Kuiken (1990) to cover the exceptions by additional patterns and these add-on files are available on CTAN and other hosts, e.g., <ftp.cs.umb.edu>. *But*, after having added one of these files at the end of the `\patterns` command in *hyphen.tex*, in order to overcome huge exception lists that should be loaded with every document, one loses the compatibility between different installations and acts against Knuth’s wishes.

The need to re-generate US English patterns

! TeX capacity exceeded, sorry
[exception dictionary=307.]
DEK

So, to follow Knuth’s rules, every document should start with loading the exception file – for this, one has to increase T_EX82’s exception size (in words) from 307 to at least 607 (as is now usual in UNIX T_EX, emT_EX and other installations). However, this is barely sufficient for the current English exception file (remember one has to add words in all possible inflexions) but for flexive languages (such as Czech, where from one stem there are about 20 different suffixes) it is unusable.

Maybe it is time to re-generate the patterns from a bigger (say, 200 000 entry) word-list once again from scratch?⁴ Imagine the day when you will know that T_EX will find 99.99% of hyphens contained in your copy of Webster, so you will not have to go through a list of exceptions and a couple of dictionaries to check hyphenation points in your document! For backward compatibility one has to save every document together with the patterns and exceptions used anyway.⁵

Making Czech and Slovak hyphenation patterns with PATGEN

“A program should do one thing, and do it well.”
Ken Thompson

The first Czech patterns were made in 1988 by Novák using PATGEN from a list of 170 000 word forms. Because of errors in his word-list, and only partially optimized PATGEN parameter settings, the patterns were good but not perfect.

The patterns weren’t publicly available, so a second attempt was done by hand by Lhotka (1991) just as MacKay (1988) did for Turkish. Because of lots of exceptions to the ‘rules’, their usage was not quite comfortable either.

As Novák’s list of words had been lately made public, we started compiling a bigger word-list from various sources using the old patterns for bootstrapping. We’ve learned a lot from the experience described by Rynning (1991) and Haralambous (1993b) and in a tutorial (Haralambous, 1994).

⁴ Otherwise in 2050 there will have to be an extra issue of *TUGboat* devoted to the publication of exceptions to *hyphen.tex*.

⁵ A search on CTAN via `quote site index` command shows 5 files of *different* lengths with the name *hyphen.tex*. (And Knuth and Liang’s *hyphen.tex* can be found there under four different names – *hyphen.tex*, *ushyph1.tex*, *ushyphen.std*, *ushyphen.tex* – which leads to the total confusion!)

Czech hyphenation rules Czech hyphenation rules are described in (Zdeněk Hlavsa et al, 1993, pp. 56–57) and in a special book (Haller, 1956) where a list of exceptions was published. Briefly, we have syllable hyphenation with ‘etymological’ exceptions. Hyphenation is preferred between a prefix and the stem, and on the boundary of compound words. Things become complicated when:

1. The word evolved in such a way that although historically it was built from a prefix plus the stem of another word, today it is perceived as a new word stem. As an example may serve the word *ro-zu-měť* – “to understand” (syllable division) against *roz-u-měť* (*roz* is the prefix and *uměť* means “to know”).
2. There is no agreement on word hyphenation – e.g., the *current* rules for word *sestra* – “sister” allow one to hyphenate *se-stra*, *ses-tra* and *sest-ra*.
3. Word stem hyphenation points change when a suffix is added – e.g., *hrad* – “castle” can’t be hyphenated, but with a suffix could – *hra-du*.
4. Compound words e.g. *tři-a-třiceti-letý* – “33 years old” are taken into account. Czech has a lot of compound words, but not to the extent that German has.
5. The hyphenation of a word depends on the semantics: *nar-val* and *na-rval*.

These rules make it hard to create patterns that describe all these exceptions and exceptions to exceptions. As we had handy a word-list with lists of allowable prefixes and suffixes, together with preliminary patterns to hyphenate word stems for bootstrapping, we decided to generate a hyphenated list of Czech words for PATGEN.

Stratified sampling

“A large body of information can be comprehended reasonably well by studying more or less random portions of the data. The technical term for this approach is stratified sampling.”
(Knuth, 1991, p. 3)

Czech is a very flexive language; on average 20–30 inflexions can be derived from one word stem by changing the suffix added and one can multiply it almost twice, as negation can be created from many words (adjectives, verbs) by prefixing *ne*. Thus from a 170 000 stem word-list about 5 000 000 inflexions may be generated. Generating patterns from such a list would be very impractical. Because the suffixes are often the same or similar, we generated a word-list by means of the following rules:

1. We add only every 7th (actually 17th worked as well) derived word form from the full list to the PATGEN input list, with exceptions that:
2. every stem must be accompanied by at least one derived form, and
3. every derived form with overlapping prefixes has to be present in the PATGEN input list as well, and
4. only one word with prefixes *ne* (by which one can create negation to almost every word) and *nej* (by which one creates superlatives) is included, and
5. the hand-made list of exceptions from Haller (1956) (about 10 000 words) and other sources are always included.

With this procedure we have 372 562 Czech words to work with PATGEN. We used the same approach for Slovak. The results are in table 3.

Table 3: PATGEN statistics for the Czech and Slovak languages

# of words	# of hyphenation points		
	Correct	Wrong	Missed
Czech			
372562	1019686 (98.26 %)	39 (0.01 %)	18086 (1.74 %)
Slovak			
333139	1025450 (98.53 %)	34 (0.01 %)	15273 (1.47 %)

Samples of PATGEN statistics are presented in tables 4, 5 and 6. These tables show that by twiddling with PATGEN parameters one may generate various versions of patterns. Usually the size of patterns and % of bad hyphenations are the minimization criteria, but maximization of % of good (found) hyphenations and other strategies might be chosen.

Compound words

“Hints for hyphenation are most often needed at the word boundaries of compound words.”
(Saarinen, 1988, p. 191)

As an experiment we took our (rather huge) word-list of Czech words in which there was marked hyphenation only on prefix and compound word boundaries.

Table 4: Standard Czech hyphenation with Liang’s parameters for English

level	length	param	% correct	% wrong	# patterns	size
1	2–3	1 2 20	96.95	14.97	+ 855	
2	3–4	2 1 8	94.33	0.47	+1706	
3	4–5	1 4 7	98.28	0.56	+1033	
4	5–6	3 2 1	98.22	0.01	+2028	32 kB

Table 5: Standard Czech hyphenation with improved (size optimized) strategy (cf. table 3)

level	length	param	% correct	% wrong	# patterns	size
1	1–3	1 2 20	97.41	23.23	+ 605	
2	2–4	2 1 8	85.98	0.31	+ 904	
3	3–5	1 4 7	98.40	0.78	+1267	
4	4–6	3 2 1	98.26	0.01	+1665	23 kB

Table 6: Standard Czech hyphenation with improved (% of correct optimized) strategy

level	length	param	% correct	% wrong	# patterns	size
1	1–3	1 5 1	95.43	6.84	+2261	
2	1–3	1 5 1	95.84	1.17	+1051	
3	2–5	1 3 1	99.69	1.24	+3255	
4	2–5	1 3 1	99.63	0.09	+1672	40 kB

Table 7: Czech hyphenation of composed words with Liang’s parameters but allowing 1-length patterns in level 1

level	length	param	% correct	% wrong	# patterns	size
1	1–3	1 2 20	72.97	14.32	+ 300	
2	2–4	2 1 8	69.32	3.09	+ 450	
3	3–5	1 4 7	84.09	4.02	+ 870	
4	4–6	3 2 1	82.61	0.33	+2625	25 kB

Table 8: Czech hyphenation of composed words with slightly modified parameters(% of correct slightly optimized)

level	length	param	% correct	% wrong	# patterns	size
1	1–3	1 2 20	72.97	14.32	+ 300	
2	2–4	2 1 8	69.32	3.09	+ 450	
3	3–5	1 4 3	90.82	4.24	+3014	
4	4–6	3 2 1	89.07	0.36	+2770	40 kB

Table 9: Czech hyphenation of composed words with other parameters (% of correct optimized, but % of wrong and size increased)

level	length	param	% correct	% wrong	# patterns	size
1	1–3	1 5 1	64.35	5.34	+1415	
2	2–4	1 5 1	67.10	1.88	+1261	
3	3–5	1 3 1	97.94	5.39	+8239	
4	4–6	1 3 1	97.91	1.14	+2882	84 kB

The PATGEN program was able to produce hyphenation patterns for this list successfully. The number of patterns was rather large, but feasible (25–84 kB, depending on parameters). From a 380 698 item word-list the patterns found 307 470 of the hyphenation points⁶ correctly, 5 040 points were hyphenated wrongly (exceptions), and 4 680 hyphenation points were missing.

To test the possibility of creating patterns for compound words in detail, we generated a word-list of more than 100 000 words with 101 687 hyphenation points marked. The list included both compound words and simple ones too.

The results of some of the runs are shown in tables 7, 8 and 9.

Some other numbers Just for fun we’ve tried patterns for different languages on our Czech PATGEN input word-list—see table 10. There are interesting speculations about these numbers—e.g., trying Slovak patterns on the Czech word-list, one finds more than 90 % of hyphenation points. On the contrary, probably because of non-syllabic principles and different rules for pronunciation, UK English rules are totally different — only 19 % of Czech words are hyphenated correctly by UK patterns. Surprisingly, Swedish, Finnish and Dutch (NE₃) patterns make fewer wrong hyphenations than the Czech old hyphenation patterns. The difference between Dutch patterns made by hand (NE₃) based on the syllabic principle) and those made by PATGEN (NE₁, NE₂) may be caused by the fact that general syllable hyphenation is relatively good for languages in which the hyphenation is based on syllabic principles. Having hyphenated word lists of different languages, it might be interesting to measure the ‘syllabic principles of hyphenation’ of different languages on an universal syllable hyphenation.

As hyphenation in most languages is based on syllabic principles, it is worth trying to create universal syllabic hyphenation and only learn the difference (exceptions) from this universal hyphenation. Let’s try to summarize what we think should be done in the future.

⁶ Some of these points might be wrong, as the database we used is only preliminary. Due to our experience with the standard hyphenation list, after correction of errors (wrongly marked hyphenation points, typos) PATGEN can generalize *substantially* better and the size of the list of patterns is reduced significantly.

Table 10: Patgen-like statistics for using various language patterns on Czech hyphenated word-list

Language	Correct	Wrong	Missed
CZ (Sev)	98.26 %	0.01 %	1.74 %
NE ₃	57.38 %	4.11 %	42.62 %
SV	57.10 %	5.32 %	42.90 %
FI	52.67 %	5.40 %	47.32 %
CZ (Lho)	93.39 %	5.89 %	6.61 %
SK	90.77 %	7.28 %	9.23 %
US	31.84 %	9.58 %	68.16 %
IT	49.27 %	9.88 %	50.73 %
NO	51.61 %	11.32 %	48.39 %
FR	59.07 %	11.54 %	40.93 %
NE ₁	59.14 %	11.59 %	41.86 %
NE ₂	58.80 %	11.99 %	41.20 %
UK	18.84 %	12.19 %	81.16 %
DE _{min}	58.62 %	12.50 %	41.38 %
DE _{max}	58.56 %	12.70 %	41.44 %
PL*	69.00 %	12.96 %	31.00 %
PL	68.06 %	13.12 %	31.94 %
DE (v.3.1)	58.84 %	13.86 %	41.16 %

* with transformed patterns — accented letters substituted by non-accented ones

Future

*“I hope T_EX82 will remain stable
at least until I finish Volume 7
of The Art of Computer Programming.”
(Knuth, 1989, p. 625)*

Possible extensions in a successor to T_EX

*“Good typography therefore is a silent art;
not its presence but rather
its absence is noticeable”
(Mittelbach and Rowley, 1992b)*

It seems feasible to incorporate either S^IT_EX (Barth, Steiner, and Herbeck, 1993) changes or separate compound word hyphenation patterns in ϵ -T_EX.

These experiments, discussed above (in the section “Compound words”) show that, even with the current T_EX, only doubling the patterns for a language with compounds might allow, e.g., switching between standard hyphenation in narrow columns and compound-word-only hyphenation in wide columns.

With a simple change in the program, one may achieve additional flexibility in hyphenation:

New registers `\leftcompoundhyphenmin` and `\rightcompoundhyphenmin` may be helpful for filtering unneeded hyphenation near compound word borders and `\compoundwordhyphenpenalty` might set a penalty (usually much lower than

`\hyphenpenalty`) for breaks on compound word boundaries. In this case `\compoundwordchar` character (i.e., the compound word mark in the DC fonts) could be *automatically* inserted there to prevent ligatures going over a compound word boundary.

Another minor addition might be added too, e.g., ε -T_EX: in the old version of MLT_EX there was implemented a flag `\dischyph` indicating whether or not to hyphenate words with discretionaries (i.e. embedded hyphens) or not. As an example may serve the citation (for Author-Prepared Books, 1993) in this paper, where we had to insert discretionaries by hand in the compound word “Author-Prepared” to achieve the limits on underfull boxes set by the editor. With setting `\dischyph=1` this wouldn’t be necessary.

Pattern generalization Apart from PATGEN extensions according to character clustering, which are orthogonal, we are thinking of the following generalization. Currently, there are only 2 classes of inter-letter state: an odd or even number that carries information whether to hyphenate or not. The natural generalization would be to have n classes. Inter-letter numbers in patterns would code these classes in such a way that number m between letters will mean that this position belongs to the class number $m \pmod n$ (when numbering classes from 0). The case $n = 2$ is the current situation, so `\pattern[2]` might mean the classical Liang patterns. Another class might be prefix boundary, compound word boundary or whatever else might possibly be useful for the hyphenation algorithm to be aware of in the word (discretionary being another possibility).

An application for English is straightforward too. Our approach will allow one to distinguish “preferred” and “less recommended” classes of hyphenation points as published in Allen (1990).

In German, one may make other classes (and patterns), e.g. classes for different discretionary breaks.

Possible extensions in a successor to T_EX

*“Please correct if you have a hyphenated word at the bottom of a right-hand page.”
(for Author-Prepared Books, 1993)*

A possible direction was shown by Plaice (1993) and in Haralambous and Plaice (1994) and Plaice (1994). With suggested clustering of letters and enriched PATGEN (Liang and Breitenlohner, 1991) one could achieve context-dependent discretionaries and thus solve the c-k → k-k-like problems in German.

Taylor (1992, p. 249) mentions a possible definition of `\brokenpenalty = \ifrecto 500\else 200\fi`. If the output routine could communicate with the parameter-breaking algorithm, word breaks crossing page boundaries could be eliminated.

Conclusions

*“Therefore it still is not the right moment to manufacture T_EX on a chip.”
(Knuth, 1989, p. 641)*

In our survey we presented an overview on the topic of hyphenation in T_EX and our results based on experience with Czech and Slovak. We conclude that the current possibilities of T_EX are far from perfect and might be improved either in the scope of T_EX82 (creation of better hyphenation patterns for various languages by PATGEN), ε -T_EX (e.g. duplication of hyphenation mechanism for compound words), or Ω or $\mathcal{N}\mathcal{T}\mathcal{S}$ (special capabilities for context-dependent discretionaries).

Acknowledgement

The presentation of this work has been made possible due to the support of Czech Grant Agency (grant Nr. 201/93/1269). We would like to thank our referee, Yannis Haralambous, Libor Škarvada and Jiří Zlatuška for comments and useful suggestions on how to improve this paper.

References

- Allen, R.Ě. *The Oxford Spelling Dictionary*, volume II of *The Oxford Library of English Usage*. Oxford University Press, 1990.
- AMS–Instructions for Author-Prepared Books. “AMS–Instructions for Author-Prepared Books”. 1993.
- Barth, W. and H. Nirschl. “Implementierung eines Verfahrens für die Silbentrennung”. Technical Report Bericht Nr. 26, Institut für Praktische Informatik, 1985.
- Barth, W. and H. Steiner. “Deutsche Silbentrennung für T_EX 3.1 (German hyphenation for T_EX 3.1)”. *Die T_EXnische Komödie* (Heft 1), 1992. Journal of DANTE (Deutschsprachige Anwendervereinigung T_EX e.V.); Group of German-speaking T_EX Users.
- Barth, W., H. Steiner, and H. Herbeck. “ \uparrow S^IT_EX Interaktive Silbentrennung für die deutsche Sprache unter T_EX 3.14 und 3.141 unter UNIX (Interactive hyphenation for German and T_EX 3.14 and 3.141 under UNIX)”. electronic documentation of \uparrow S^IT_EX distributed from `eiunix.tuwien.ac.at`, 1993.

- Beccari, Claudio. “Computer Aided Hyphenation for Italian and Modern Latin”. *TUGboat* **13**(1), 23–33, 1992.
- Beeton, Barbara. “Hyphenation exception log”. *TUGboat* **10**(3), 336–341, 1989.
- Beeton, B. N. “Hyphenation exception log”. *TUGboat* **5**(1), 15, 1984.
- Beeton, B. N. “Hyphenation exception log”. *TUGboat* **6**(3), 121, 1985.
- Beeton, B. N. “Hyphenation exception log”. *TUGboat* **7**(3), 145–146, 1986.
- Beeton, B. N. “Hyphenation exception log”. *TUGboat* **13**(4), 1992.
- Braams, J. “Babel, a multilingual style-option system”. *Cahiers GUTenberg* **10-11**, 71–72, 1991a.
- Braams, Johannes. “Babel, a multilingual style-option system for use with L^AT_EX’s standard document styles”. *TUGboat* **12**(2), 291–301, 1991b.
- Braams, Johannes. “An update on the babel system”. *TUGboat* **14**(1), 60–62, 1993.
- Breitenlohner, Peter. “German T_EX, a next step”. *TUGboat* **9**(2), 183–185, 1988.
- Canzii, G., F. Genolini, and D. Lucarella. “Hyphenation of Italian words”. *TUGboat* **5**(1), 14, 1984.
- de Rezende, Pedro. “Portuguese hyphenation table for T_EX”. *TUGboat* **8**(2), 102–102, 1987.
- Désarménien, Jacques. “How to run T_EX in a French environment: Hyphenation, fonts, typography”. *TUGboat* **5**(2), 91, 1984.
- DUDEN. *Duden Band 1 — Rechtschreibung der deutschen Sprache*. Dudenverlag, 20., neu bearbeitete und erweiterte Auflage edition, 1991.
- Emerson, Sandra L. and K. Paulsell. *troff Typesetting for UNIXTM Systems*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.
- Fanton, M. “T_EX : les limites du multilinguisme”. *Cahiers GUTenberg* **10-11**, 73–80, 1991.
- Ferguson, Michael J. “A multilingual T_EX”. *TUGboat* **6**(2), 57–58, 1985.
- Ferguson, Michael J. “T_EX is Multilingual”. In Thiele (1988), pages 179–189.
- Ferguson, M. J. “Fontes latines européennes et T_EX 3.0”. *Cahiers GUTenberg* **7**, 29–32, 1990.
- Gaulle, Bernard. “Requirements in multilingual environments”. in electronic form (version 1.02) on CTAN as file `vt15d02.tex`, 1994.
- Haller, Jiří *Jak se dělí slova (How the words get hyphenated)*. SPN Praha, 1956.
- Haralambous, Y. “ScholarT_EX”. *Cahiers GUTenberg* **10-11**, 69–70, 1991.
- Haralambous, Yannis. “T_EX Conventions Concerning Languages”. *T_EX and TUG News* **1**(4), 3–10, 1992a.
- Haralambous, Yannis. “Hyphenation patterns for ancient Greek and Latin”. *TUGboat* **13**(4), 457–469, 1992b.
- Haralambous, Yannis. “DC fonts — questions and answers”. *T_EX and TUG News* **2**(1), 10–12, 1993a.
- Haralambous, Yannis. “Using PATGEN to Create Welsh Patterns”. Submitted to *TUGboat*, 1993b.
- Haralambous, Yannis. “A small tutorial on the multilingual features of Patgen2”. in electronic form, available from CTAN as `info/patgen2.tutorial`, 1994.
- Haralambous, Yannis and J. Plaice. “First applications of Ω: Adobe Poetica, Arabic, Greek, Khmer”. *TUGboat* **15**(3), 344–352, 1994.
- Jeffrey, Alan. “A PostScript font installation package written in T_EX”. *TUGboat* **14**(3), 285–292, 1993.
- Knuth, Donald. “A note on hyphenation”. *TUGboat* **4**(2), 64, 1983.
- Knuth, Donald E. *The T_EXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986b.
- Knuth, Donald E. *T_EX: The Program*, volume B of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986a.
- Knuth, Donald E. “The Errors of T_EX”. Technical Report STAN-CS-88-1223, Stanford University, Department of Computer Science, 1988.
- Knuth, D. E. “The Errors of T_EX”. *Software — Practice and Experience* **19**(7), 607–681, 1989. This is an updated version of Knuth (1988).
- Knuth, Donald Ervin. *3:16 Bible texts illuminated*. A-R Editions, Inc., 1991.
- Kołodziejska, Hanna. “Dzielenie wyrazów polskich w systemie T_EX”. Technical Report 165, Sprawozdania Instytutu Informatyki Uniwersytetu Warszawskiego, 1987.
- Kołodziejska, Hanna. “Le traitement des textes polonais avec le logiciel T_EX”. *Cahiers GUTenberg* (0), 3–10, 1988.
- Kopka, Helmut. *L^AT_EX — Erweiterungsmöglichkeiten mit einer Einführung in METAFONT*. Addison-Wesley Verlag, Bonn, Germany, second edition, 1991.
- Krstev, Cvetana. “Serbo-Croatian hyphenation: a T_EX point of view”. *TUGboat* **12**(2), 215–223, 1991.
- Kuiken, Gerard. “Additional Hyphenation Patterns”. *TUGboat* **11**(1), 24–25, 1990.
- Lhotka, Ladislav. “České dělení pro T_EX (Czech hyphenation for T_EX)”. *C_STUG bulletin* (4), 8–9, 1991.

- Liang, Frank and P. Breitenlohner. “PATtern GENeration program for the T_EX82 hyphenator”. Electronic documentation of PATGEN program version 2.0 from UNIX T_EX distribution at `ftp.cs.umb.edu`, 1991.
- Liang, Frank M. “T_EX and hyphenation”. *TUGboat* **2**(2), 19–20, 1981.
- Liang, Franklin Mark. “Word Hy-phen-a-tion by Com-pu-ter”. Technical Report STAN-CS-83-977, Stanford University, 1983.
- MacKay, Pierre A. “Turkish hyphenations for T_EX”. *TUGboat* **9**(1), 12–14, 1988.
- Malyshev, B., A. Samarin, and D. Vulis. “Russian T_EX”. *Cahiers GUTenberg* **10-11**, 1–6, 1991a.
- Malyshev, Basil, A. Samarin, and D. Vulis. “Russian T_EX”. *TUGboat* **12**(2), 212–214, 1991b.
- Mittelbach, Frank and C. Rowley. “The future of high quality typesetting: structure and design”. In Zlatuška (1992), page 255.
- Mittelbach, Frank and C. Rowley. “The pursuit of quality—How can automated typesetting achieve the highest standards of craft typography?”. In *Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography, Lausanne, Switzerland, 1992*, edited by C. Vanoirbeek and G. Coray, pages 261–273, New York. Cambridge University Press, 1992b.
- NTS-L. “New Typesetting System discussion list”. 1992–. This is an electronic list devoted to discussions about T_EX’s successor. To subscribe, send a request with the text `subscribe nts-1` to `listserv@vm.urz.uni-heidelberg.de`.
- Obermiller, Walter. “T_EX in Germany”. *TUGboat* **12**(2), 211–212, 1991.
- Partl, Hubert. “German T_EX”. *TUGboat* **9**(1), 70–72, 1988.
- Partl, Hubert. “How to make T_EX and L^AT_EX international”. In *Man-Machine Interface in the Scientific Environment. Proceedings of the 8th European Summer School on Computing Techniques in Physics. Skalský Dvur, Czechoslovakia, 19–28 September 1989*, edited by J. Nadrchal, volume 61 of *Computer Physics Communications*, pages 190–200, Amsterdam, The Netherlands. European Summer Schools on Computing Techniques in Physics, North-Holland Publishing Company; Elsevier Science Publishers B. V., 1990. Invited paper.
- Plaice, John. “Language-dependent ligatures”. *TUGboat* **14**(3), 271–274, 1993.
- Plaice, John. “Progress in the Omega project”. *TUGboat* **15**(3), 320–324, 1994.
- Rynning, Jan Michael. “Swedish Hyphenation for T_EX”. Received in electronic form from author via email `jmr@nada.kth.se`, 1991.
- Saarinen, Kauko. “Experiences with T_EX in Finland”. In Thiele (1988), pages 189–194.
- Samarin, A. and A. Urvantsev. “CyrTUG, le monde T_EX en cyrillique”. *Cahiers GUTenberg* **12**, 71–74, 1991.
- Schrod, Joachim. “An International Version of *MakeIndex*”. *Cahiers GUTenberg* **10-11**, 81–90, 1991.
- Schulze, Bernd. “German hyphenation and *Umlauts* in T_EX”. *TUGboat* **5**(2), 103, 1984.
- Sojka, Petr and P. Ševeček. “Hyphenation in T_EX—Quo Vadis?”. In *Proceedings of the 9th European T_EX Conference, Gdańsk, 1994*, edited by W. Bzyl and T. Przechlewski, pages 59–68. 1994.
- Taylor, Philip. “The Future of T_EX”. In Zlatuška (1992), pages 235–254.
- Thiele, Christina, editor. *Proceedings of the T_EX Users Group 9th Annual Meeting, Montréal, 1988*, Providence, U.S.A. T_EX Users Group, 1988.
- Thulin, Anders. “More hyphenation exceptions”. *TUGboat* **8**(1), 76–76, 1987.
- Vulis, Dimitri. “Notes on Russian T_EX”. *TUGboat* **10**(3), 332–336, 1989.
- Zdeněk Hlavsa et al. *Pravidla českého pravopisu (The rules of the Czech spelling)*. Academia Praha, 1993.
- Zlatuška, J. “Automatic generation of virtual fonts with accented letters for T_EX”. *Cahiers GUTenberg* **10-11**, 57–68, 1991.
- Zlatuška, Jiří editor. *Proceedings of the 7th European T_EX Conference, Prague, 1992*. Masarykova Universita Brno, 1992.

Notes on Compound Word Hyphenation in T_EX

Petr Sojka
Faculty of Informatics
Masaryk University Brno
Burešova 20, 60200 Brno
Czech Republic
Email: `sojka@muni.cz`

Abstract

The problems of automatic compound word and discretionary hyphenation in T_EX are discussed. At present, such hyphenation points have to be marked manually in the T_EX source file. Several methods for tackling with these problems are presented. The results obtained from experiments with a German word-list are discussed.

Motivation

... problems [with hyphenation] have more or less disappeared, and I've learnt that this is only because, nowadays, every hyphenation in the newspaper is manually checked by human proof-readers. (Jarnefors, 1995)

In (Sojka and Ševeček, 1994) (reprinted in these Proceedings) we presented a case study of problems related to achieving quality hyphenation in T_EX — especially pattern generation for flexive languages like Czech. It was shown that most issues can be handled within the frame of good old T_EX, but some of them definitely not, because T_EX was primarily designed not as a universal tool for the typesetting of all kinds of publications in all languages, but as one for typesetting of *The Art of Computer Programming* (Knuth, 1968), which is written in American English.

In this paper we continue elaborating these issues, with the emphasis on the hyphenation problems in the presence of long compound words in Germanic (and Slavic) languages.

Problems

Compounds. The main problem with automatic hyphenation was nicely expressed on the ISO-10646 electronic discussion list by Jarnefors:

“The leading Swedish daily newspaper *Dagens Nyheter* had severe problems with occasional incorrect hyphenations a couple of years ago. It (and its computerised typesetting) was for a time the object of much amusement, ridicule and irritation from its readers. These problems have more or less disappeared, and I've learnt that this is only because, nowa-

days, *every* hyphenation in the newspaper is manually checked by human proof-readers. Because of the higher frequency of long words in Swedish compared to e.g. English or French, around a third of all lines in a typical newspaper article (with approximately 30 characters per line) end with a hyphenated word.

The hyphenation problems in Swedish have to do with the high frequency of compound words (the Swedish vocabulary can't be enumerated: new compounds are easily created by anyone) and the rule that a compound word shall always be hyphenated between the constituent word parts, to ease the flow of reading.”

For instance, in German and Czech there are no hyphens in compound words, you take the first word, rarely a fill-character and the second word. In some languages, compounds are built with hyphens. With this construction, it is easy to break at the end of line and to spell-check. However, in most of the languages compound word boundaries cannot be deduced from syntax only.

Dependency of hyphenation points on semantics. In some cases, even the context of the sentence is needed in order to be able to decide on the hyphenation point. Collection of examples for several languages follows:

Czech *nar/val* ‘narwhal’ and *na/rval* ‘gathered by tearing, plucked’; *pod/robit* ‘subjagate, to bring under one's domination’ and *po/drobit* ‘to crumble’; *o/blít* ‘to vomit up’ and *ob/lít* ‘to pour around’

Danish *træ/kvinden* ‘the wood lady’ and *træk/vinden* ‘the draught’; *ku/plet* ‘verse’ and *kup/let* ‘domed’

Dutch *kwart/slagen* ‘quarter turns’ and *kwarts/lagen* ‘quartz layers’; *go/spel* ‘the game of Go’ and *gos/pel* ‘certain type of music’; *rots/tempel* ‘rock temple’ and *rot/stempel* ‘damned stamp’; *dij/kramp* ‘cramp in the thighs’ and *dijk/ramp* ‘dike catastrophe’; *ver/ste* ‘farthest’ and *vers/te* ‘most fresh’.

English *rec/ord* and *re/cord*

German *Staub/ecken* ‘dusty eck’ and *Stau/ Becken* ‘traffic jam in the valley’; *Wach/stube* ‘guard room’ and *Wachs/tube* ‘wax tube’

Exceptions. Some hyphenation points are forbidden because of unwanted connotations the new parts of the word may have:

Czech *kni/hovna*, *sere/náda*, *tlu/močení*, *se/kunda*

English *the/rapists*, *anal/ysis*

German *Spargel/der*, *beste/hende*, *Gehörner/ven*, *bein/halten*, *Stiefel/tern*

Discretionary hyphenation points.

1. `\discretionary{xx}{x}{xx}` (in German, *x* is a consonant *f*, *l*, *m*, *n*, *p*, *r* or *t*)

Now there will be the situation that the first word ends with a double consonant and the second word starts with the same consonant. If the second letter of the second word is a consonant, nothing changes — *Sauerstoff* + *Flasche* composes to *Sauerstoffflasche*. If the second letter of the second word is a vowel, the three consonants will be reduced to two — *Schiff* + *Fahrt* composes to *Schiffahrt*. One can find meaning-dependent discretionaries: *Bett/tuch* ‘sheet’ vs. *Bet/tuch* ‘prayer shawl’.

2. `\discretionary{k}{k}{ck}` (German)

This discretionary (as most of the others) has the rationale in the fact that pronunciation of *c* depends on the following letter (as in other languages). If hyphen occurs just after the letter *c*, the reading is slowed down because the reader doesn’t know how to pronounce it and the eye has a long way to the beginning of the next line.

Even here the hyphenation can depend on the word meaning: word *Druckerzeugnis* is hyphenated *Druck/erzeugnis* in case of ‘printed matter’ or *Druk/kerzeugnis* when speaking about a ‘certificate for a printer’.¹

¹ The German speaking countries are in the process of introducing new rules for hyphenation, in which *ck* is not any more allowed to be hyphenated. With the new rules, an old way which was introduced in 1902 — e.g. hyphenation of *Zuk/ker* ‘sugar’ might change to *Zu/cker* in the future norm.

3. `\discretionary{a}{}{aa}` (Dutch)

There is another type of discretionary in which a character is deleted in case hyphenation occurs — word *omaatje* becomes *oma/tje* when hyphenated.

4. `\discretionary{é}{}{ee}` (Dutch)

Apart from character deletion another change may occur: *cafeetje* becomes *café-tje* when hyphenated.

5. `\discretionary{l}{l}{ll}` (Catalan)

In Catalan the word parallel is broken as paral|lel, intelligencia as intel|ligencia. *ll* is considered as one character (trigraph). With this hyphenation it changes to another two characters.

Stability of a language. Another complication is the fact that language is not fixed, non-evolving entity, but it changes, sometimes quite rapidly. New words, especially compounds, are being adopted every day. An example of an adaptation of a language to the technology — the typewriter and telegraphy in this case — may serve different spelling allowed for unlauded characters *ä*, *ö*, *ü* and *ß* in German (*ae*, *oe*, *ue*, *ss*). Some compounds are becoming perceived as base words. Thus the idea of fixing hyphenation algorithm/patterns once and forever is not a clever one.² A solution may consist in relatively easy generation of algorithm or patterns from the updated dictionary or description of changes.

Solutions

Compounds. It is obvious that we need to take the burden of the manual markup of compound word borders from the writer and leave it to the machine (typesetting system). The proper solution of this problem is a language module for every language, with the ability of creating new words by composition from others. This module, based on the morphology of a language, is needed, e.g., in spellchecker for that language anyway. Most probably, such language modules will become a part of the language support of operating systems in near future. Such dynamic libraries will be shared among software applications. Building such a module, however, is not a trivial task, because only some of the compounds are meaningful words.

² When storing document for later retypesetting with T_EX we also have to save the hyphenation patterns.

Table 1: Example of discretionary hyphenation table for German

pre break text 1	post break text 2	no break text 3	left context 4	right context 5	discretionary character 6	example 7
k	k	ck	c	k	c_1	Drucker
ek	k	äck	äc	k	c_2	Bäcker
ff	f	f	f	f	c_3	Schiffahrt
ll	l	l	l	l	c_4	Rolladen
mm	m	m	m	m	c_5	Programmeister
nn	n	n	n	n	c_6	Brennessel
pp	p	p	p	p	c_7	Stoppunkt
rr	r	r	r	r	c_8	Herraum
tt	t	t	t	t	c_9	Balettheater

Looking for a temporary \TeX patch that will help the current \TeX users, especially those writing in Germanic and Slavic languages, the following algorithm may be used (compare with Sojka and Ševeček, 1994):

1. For a particular language a special word-list is created, which contains all word forms, but only compound word borders are marked there.
2. Hyphenation patterns from this word-list are created by PATGEN (Liang and Breitenlohner, 1991).
3. A special pass in \TeX 's paragraph breaking algorithm (for detailed description consult Knuth and Plass, 1981; Knuth, 1986a; Knuth, 1986b) is added after the first (no hyphenation trial) pass. Words are hyphenated using the compound word patterns. Then, an extra penalty `\compoundwordhyphenpenalty` is associated with these hyphenation points.
4. If `\tolerance` hasn't been met by now, further hyphenation points are added using the 'standard' patterns. These new hyphenation points have associated `\hyphenpenalty`, allowing differentiation between the two types of hyphenation points.
5. Hyphenation points 'near' the word borders (specified by `\leftdiscretionaryhyphenmin` and `\rightdiscretionaryhyphenmin` are suppressed (removed).
6. The algorithm now continues with the 'old' second and eventually the third (`\emergencystretch`) passes.
7. `\compoundwordchar` (as e.g. in Cork-coded fonts `\char'027`) is included at compound word breakpoint in order to prevent ligatures spanning over the word borders *šéflékař* 'chief

doctor' versus *šéflékař* which is wrong due to the appearance of the fl ligature).

Discretionary hyphenation points. Manual insertion of discretionary points is tedious and it is usually forgotten³, leading to typographic errors.

One solution is as follows. For every language a table of possible discretionary points is created (for a German example see Table 1).

In the word-list, the words with these discretionary points are added with the "discretionary character" inserted between "left context" and "right context". From such extended word-list the patterns are generated.

The hyphenation algorithm of \TeX (for details see Knuth, 1986a, parts 38–43, sections 813–965) has to be extended. Roughly speaking

1. As a first step, "normal" hyphenation points in the word in question are found.
2. The discretionary exception table is looked up (similar to the `\hyphenation` list of exceptions). If the word is found there, a discretionary point is inserted and algorithm ends, otherwise continue to step 3.
3. The discretionary table is looked up and at the hyphenation points that match "left and right context" strings (columns 4 and 5 in Table 1), the "discretionary character" (column 6) is inserted. Such a word is hyphenated once again to check whether this discretionary point really applies at this position. If so, the corresponding discretionary point (columns 1–3 of Table 1) is automatically inserted.

³ How many of you, English-speaking \TeX users, remember to type `\eighdiscretionary{t}{t}{t}` instead of just `\eighteen`?

4. “Normal” hyphenation points, which appear ‘near’ to “discretionary” hyphenation points (within the ‘window’ specified by the values of counters `\leftdiscretionaryhyphenmin` and `\rightdiscretionaryhyphenmin`), are removed.

This approach takes the advantage of the data structure used for storing the information about the hyphenation points. The patterns are stored using the *trie* data structure (Knuth, 1973, pp. 481–505). This data structure allows effective prefix and postfix compression. Because of that, the increase in the size of the patterns is negligible, as the patterns doublets share both prefix and postfix parts in the trie.

Also, the look up time in the trie is linear with respect to the word length of hyphenated words. The time needed for looking up in the trie for the second time is thus acceptable—it is only performed sometimes—when the context of a hyphenation point is matched in the discretionary table.

The algorithm is backward compatible in the sense that if discretionary table is not present for the current language, nothing changes with respect to the standard T_EX behaviour.

Exceptions. The exceptions can be reasonably handled by the patterns. Although the generation of patterns for languages with lots of exceptions may lead to the complex patterns, it is much better to regenerate the patterns with the exceptions than maintaining huge lists of exceptions and to slow down the processing considerably.

Because regenerating of patterns is not always possible, to allow enrichment of the knowledge of discretionary hyphenation points compiled into the patterns, it is wise to introduce new `\discretionaryhyphenation` for this purpose.

Experiments

For experiments we had several databases of words available. For flexive languages (Czech, German), they were based on morphology, for English it was just a list of word forms. We did our PATGEN experiments with German word-list generated from the full word-list by our stratified sampling technique very similar to that we described on page 63 in (Sojka and Ševeček, 1994) for Czech. We took German because the problems there are the most serious. Simple statistics show how the languages differ:

Non-uniformity of languages. In the Table 2 on page 295 there are histograms of word lengths in our databases. Although it is clear that shorter words are more frequent than the long ones, we see that in German the average word is much longer

than in English and also in Czech. It is interesting to compare the total number of words. As Czech is very flexive language, from about 170 000 word stems we got more than 3 300 000 word forms. One can compare that with the best English dictionaries and spellers, which have not more than 200 000 word forms. Flexive number (ratio of total number of word form and number of word stems) for German is about 3 (we have about 120 000 word stems), but for Czech it is almost 20.

The average word length depends on the word-list chosen, but in general our results are commensurable with the result published for Welsh (Haralambous, 1993)—9.71 characters per word, but the words like *Llanfairpwllgwyngyllgogerychwryndrobwillllantysiliogogoch* were not taken into account there.

Compounds (German). In the word-list, only the compound word borders and prefixes were marked. This led to about 150 000 positions in our German word-list. The words without any breaks of this kind were not removed. The results of PATGEN runs applied to this word-list are summarised in tables 3 and 4. The efficiency achieved (about 90% breaks covered) is pretty sufficient, as ‘normal’ hyphenation pass follows and the error when hyphenation point is classified as ‘normal’ instead of ‘compound’ reflects only different penalty associated with this break. At the expense of pattern size we can do even better (see Table 5).

Discretionary hyphenation points. In our German word-list we had 1626 words with the *c-k* discretionary and 42 words with the discretionary hyphenation of type *x-x*, where *x* is a consonant—(see Table 1, (Raichle, 1995) or (DUDEN, 1991) for a list of possible discretionaries in German).

Then we created doublets of these words with these discretionaries by inserting the discretionary character (column 6) at the hyphenation position and added them to our word-list. Then we applied PATGEN at this new word-list. The results can be compared in tables 6 and 7. The difference in pattern size is small as expected—the size of pattern file increased by less than 0.4 kB, which makes a difference in the trie structure of about 100 bytes only.

Conclusions

We are claiming that the integration of language modules with built-in knowledge about a particular language is a must in today’s top-rated systems for publishing. We have suggested extensions of hyphenation algorithms of T_EX that may help

with hyphenation especially in Germanic languages with high frequency of compound words and discretionary hyphenation. Suggested extensions are possible with limited changes to TEX —The Program (Knuth, 1986a). Their implementation in any conservative successor to TEX will be rather straightforward and when the community is agreed on their usefulness they will be implemented as an independent change file. We remain undecided on the extended syntax and primitives our approach needs.

Acknowledgement

The presentation of this work has been made possible due to the support of Czech Grant Agency (grant Nr. 201/93/1269). The author would like to thank Pavel Ševeček (LOGOS, Inc.) for providing language word-lists to make the experiments and for valuable discussions on these topics. I also thank everyone who helped to improve the wording of this paper.

References

- DUDEN. *Duden Band 1—Rechtschreibung der deutschen Sprache*. Dudenverlag, 20., neu bearbeitete und erweiterte Auflage edition, 1991.
- Haralambous, Yannis. “Using PATGEN to Create Welsh Patterns”. Submitted to *TUGboat*, 1993.
- Jarnefors, O. “ISO-10646 email discussion list”. 1995.
- Knuth, D. E. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, 1973.
- Knuth, D. E. *The Art of Computer Programming*. Four volumes. Addison-Wesley, 1968. Seven volumes planned.
- Knuth, Donald E. *The TEX book*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986b.
- Knuth, Donald E. *TEX : The Program*, volume B of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986a.
- Knuth, Donald E. and M. F. Plass. “Breaking Paragraphs into Lines”. *Software—Practice and Experience* 11(11), 1119–1184, 1981.
- Liang, Frank and P. Breitenlohner. “PATtern GENeration program for the TEX 82 hyphenator”. Electronic documentation of PATGEN program version 2.0 from UNIX TEX distribution at `ftp.cs.umb.edu`, 1991.
- Raichle, B. “Kurzbeschreibung – `german.sty` (Version 2.5)”. 1995. Available from CTAN.
- Rynning, Jan Michael. “Swedish Hyphenation for TEX ”. Received in electronic form from author via email `jmr@nada.kth.se`, 1991.
- Sojka, Petr and P. Ševeček. “Hyphenation in TEX —Quo Vadis?”. In *Proceedings of the 9th European TEX Conference, Gdańsk, 1994*, edited by W. Bzyl and T. Przechlewski, pages 59–68. 1994.

Table 2: Available word-lists' statistics

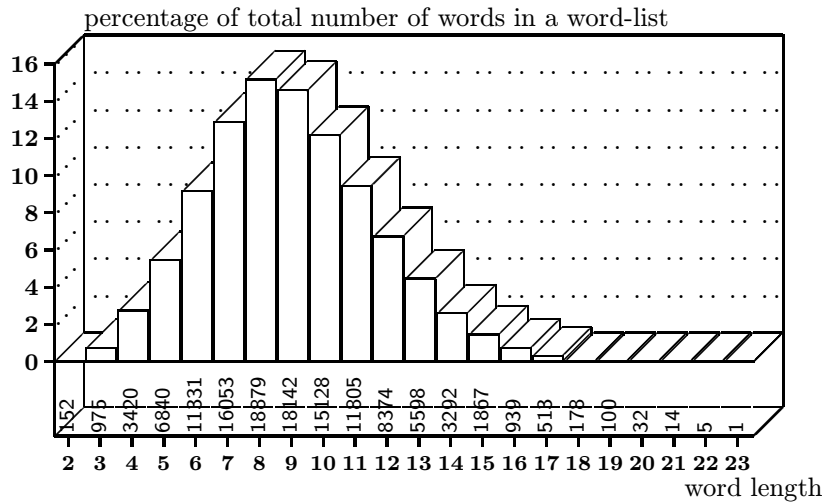
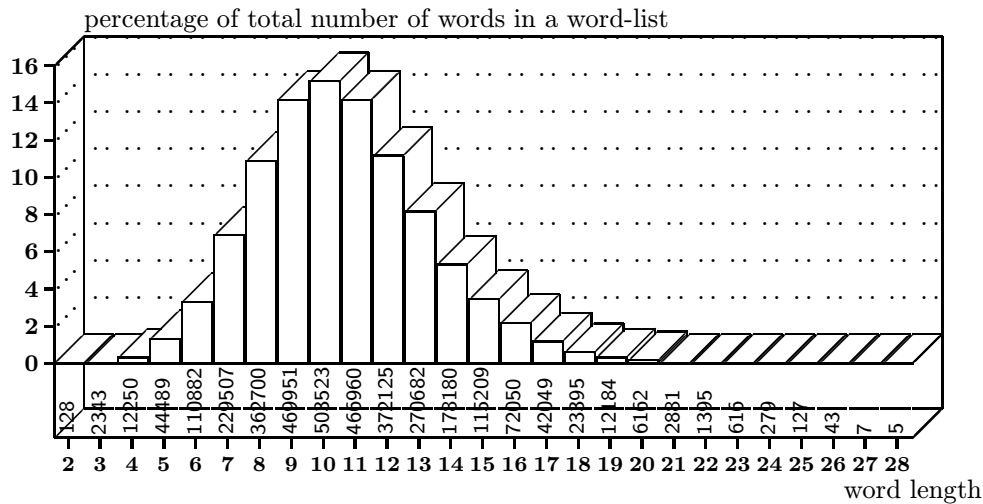
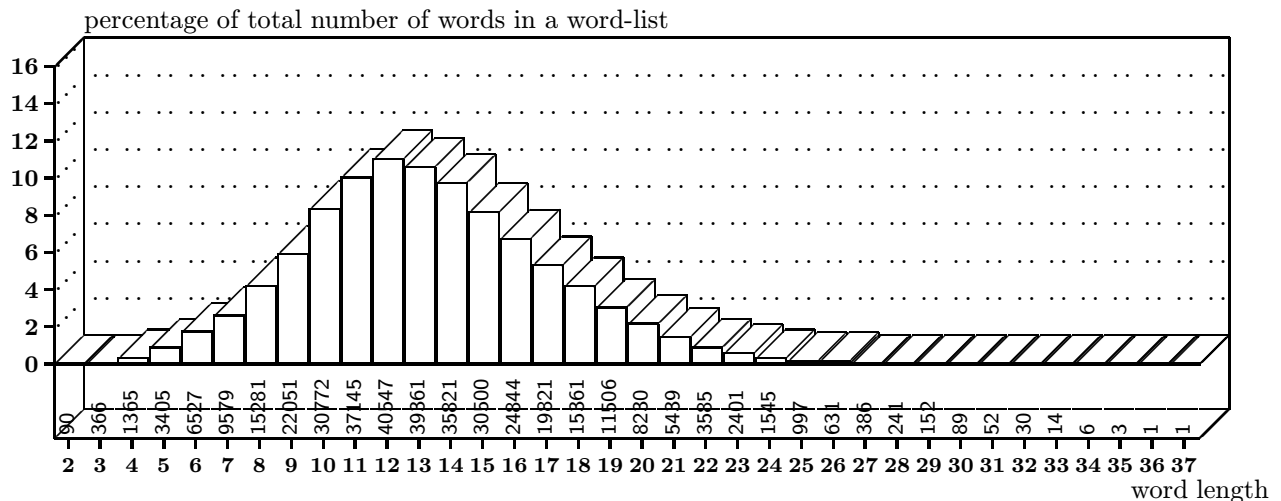
US English word-list (123 664 words), average word length 8.93 characters**Czech word-list (3 300 122 words), average word length 10.55 characters****German word-list (368 152 words), average word length 13.24 characters**

Table 3: German compound word hyphenation with pattern size optimized strategy

level	length	param	% correct	% wrong	# patterns	statistics
1	1-3	1 2 20	62.41	13.38	+ 472	good=134279
2	2-4	2 1 8	52.89	2.53	+ 712	bad=676
3	3-5	1 4 7	87.11	4.05	+2951	missed=22636
4	4-6	3 2 1	85.57	0.43	+1506	patterns size=33.6 kB

Table 4: German compound word hyphenation with different (% of correct optimised) strategy

level	length	param	% correct	% wrong	# patterns	statistics
1	1-3	1 2 20	62.41	13.38	+ 472	good=143478
2	2-4	2 1 8	52.89	2.53	+ 712	bad=698
3	3-5	1 4 3	93.06	4.23	+6612	missed=13437
4	4-6	3 2 1	91.44	0.44	+1586	patterns size=56.5 kB

Table 5: German compound word hyphenation covering even more break points

level	length	param	% correct	% wrong	# patterns	statistics
1	1-3	1 3 1	60.43	9.87	+4819	good=149502
2	1-4	1 3 2	60.24	4.21	+1714	bad=888
3	3-6	1 2 1	98.76	10.82	+1939	missed=7413
4	3-7	1 1 1	95.28	0.57	+ 353	patterns size=70.2 kB

Table 6: Standard German hyphenation patterns generation (slightly improved (size) Liang's parameters)

level	length	param	% correct	% wrong	# patterns	statistics
1	1-3	1 2 20	94.25	23.72	+ 449	good=485590
2	2-4	2 1 8	82.66	0.56	+1183	bad=48
3	3-5	1 4 7	98.59	1.08	+1737	missed=8047
4	4-6	3 2 1	98.37	0.01	+1333	patterns size=25.2 kB

Table 7: German hyphenation patterns generation with word-list with discretionary points added (the same parameters as above)

level	length	param	% correct	% wrong	# patterns	statistics
1	1-3	1 2 20	93.90	23.40	+ 456	good=492366
2	2-4	2 1 8	82.48	0.55	+1182	bad=60
3	3-5	1 4 7	98.60	1.13	+1760	missed=8155
4	4-6	3 2 1	98.37	0.01	+1388	patterns size=25.6 kB

Literate plain Source is Available!

Włodek Bzyl

Instytut Matematyki

Uniwersytet Gdański

Wita Stwosza 57

80-952 Gdańsk

Poland

Email: matwb@univ.gda.pl

Abstract

Based on Norman Ramsey's NOWEB system a new literate tool for the \TeX language has been built. The new system was used to create a 'literate plain' source. Although the resulting file is principally `plain.tex` code interleaved with documentation, borrowed mainly from *The \TeX book*, it presents the whole code from a different perspective. The documentation is organized around the macros as they appear in the `plain.tex` file rather than around the topics as in *The \TeX book*. This means that the typeset `plain.dvi` is *not* a user manual, even though many notions are explained there.

Introduction

When it was introduced, literate programming was synonymous with WEB, a system for writing literate Pascal programs. Since then many different WEBS, each aiming at a particular programming language (or small group of related languages), have been created. Each WEB is constructed of two separate parts, one called TANGLE, the other WEAVE. Typically each part consists of just one program performing many tasks — it expands macros, prettyprints code, generates and sorts an index, etc. This makes adaptation of the existing WEB to another language extremely difficult.

Another approach to literate programming was presented by Norman Ramsey, in NOWEB. He designed and realized the TANGLE/WEAVE pair as UNIX pipes. By extending and/or replacing parts of pipes with programs, written in AWK, ICON, Flex, Perl, C, \TeX or METAFONT, a new tool could be created with relatively small effort. As a result, with NOWEB, it was possible to create a simple \TeX -WEB system by writing an AWK script and a new \TeX format.

WEB for everyone?

WEB is a powerful tool. The strength of literate programs lies in their ability to produce high-quality typeset documentation. The strength of literate programming lies in allowing you to write code where you are telling humans what the computer should do, instead of telling computer what should be done. Obviously we are more efficient and precise when

communicating with humans than computers. Thus literate programs are more easily written and maintained than ordinary ones.

WEB is a complex tool. A literate program consists of pieces of documentation and named chunks containing code and references to other chunks. The pieces are arranged in an order which helps to explain (and understand) the program as a whole. The WEB system consists of two processors: TANGLE and WEAVE.

TANGLE is used to extract a program by replacing one named chunk by its definition. The process of replacement is recursive; it continues until no named chunks remain. From one WEB source many programs could be extracted (by presenting TANGLE with different chunks).

WEAVE is used to convert WEB markup into \TeX markup as described and coded in a separate format file. It handles numerous typographical details of typeset documentation and provides support for typical tasks such as cross-referencing, preparation of indexes, bibliography. Formats for long and short documents will be different. To typeset a converted file you will need \TeX running on your system. Errors can creep into \TeX code. Getting \TeX code working with other formats could end with a short trip into the \TeX language (this will be needed if you plan your literate program to form part of an article, a report, or a book).

We learn by reading: why not read 'literate books'? There are a few such books already and more will appear. We learn by writing too: why

not try one of the existing tools? The C/C++/Fortran programmer could try CWEB or FWEB. Programmers writing in other languages could check the CTAN directory `/tex-archive/web` for other possible tools. If your language is not on the list, or you are not able to express yourself within the style offered, then you are welcome to join the province of those who build their own tools. This territory is growing fast due to the efforts of Norman Ramsey, who established a base for creating simple and extensible literate tools.

Presenting a new tool: \TeX -WEB

Norman Ramsey was the first to attempt to create a generic literate tool, not aimed at a particular language. Such a tool would (of itself) be useless because of its generality—the key to the usefulness of NOWEB lies in its extensibility. The tasks for TANGLE and WEAVE were divided among stand-alone programs. To simplify tangling and weaving a front end was introduced. It performs a kind of lexical analysis of the source, a task previously performed by both processors separately. The front end provided with NOWEB is called `markup` because it marks each line of source as line of text, as beginning/end of code/documentation, as definition/use of named chunks, etc.¹

WEAVE

```
markup foo.tw |
    awk -f web2tex.awk > foo.tex
```

With `markup` as its front end, WEAVE was built as a pipeline where AWK, obeying commands from the script `web2tex.awk`, reads a marked source line by line and performs actions depending on the line type. Most of the time it inserts a bunch of \TeX macros, for example inserting index macros.

The format `tweb.sty` provides support for cross references, indexes, and multicolumn output. There you find macros `\chapter`, `\[sub[sub]]section`, `\paragraph`², `\printcontents`, `\title`.

TANGLE

```
markup foo.tw | nt > foo.sty
markup foo.tw | nt -R'Chunk B' > foo.sty
markup foo.tw | mnt 'Chunk B' 'Chunk A'
```

Here we have several possibilities. We can extract code beginning from the chunk named `<<*>>`, or from `'Chunk B'` (see template file below). Finally,

¹ There is `unmarkup` which works in the opposite way. I also borrowed two more programs: `nt` (tangle) and `mnt` (multiple tangle) from NOWEB.

² These macros should not be overused. Usually the chunk name alone is a better choice.

`'Chunk A'` and `'Chunk B'` could be simultaneously extracted to the files with the same names.

\TeX

```
tex foo.tex
makeindex -s dnd.ist -o foo.dnd foo.ddx
makeindex -s und.ist -o foo.und foo.udx
makeindex -s chn.ist -o foo.chn foo.chk
tex foo.tex
```

Indexes are sorted by `makeindex`. Three very short index style files provide formatting of the different indexes. (MSDOS `makeindx` breaks on large indexes.)

Sample Makefile. To ease work with tools a simple Makefile is provided. Type `make` on the command line, press the **Enter** key, and the following lines will appear on a terminal:

```
Tangling:  make foo.sty
Texing:    make foo.dvi
Weaving:   make foo.tex
Making archive: make archive
Cleaning:  make clean or veryclean
```

Since there are many different conventions for where to store files in a file system, three variables are defined in the Makefile:

- `SCRIPTDIR`—where `web2tex` and other scripts are stored (defaults to `BIN`),
- `INDEXDIR`—where index styles are stored (defaults to `IDXSTY`),
- `NOWEBDIR`—where the programs `markup`, `nt`, `mnt` are stored (defaults to `/usr/local/lib/nweb`).

Also:

- `MAKEINDEX`—the name of the `makeindex` program (defaults to `makeindex`),

deals with the fact that the command has a different name on MSDOS systems.

Template of \TeX -WEB source. The structure of a \TeX -WEB file is shown in the example below.

```
File name: foo.tw
-----
\title{foo.tw -- template file}
\printcontents % if you want TOC
@
The skeleton of the file foo.tw
<<*>>=
<<Chunk A>>
<<Chunk B>>
@
Documentation for Chunk A.
<<Chunk A>>=
( $\TeX$  code / references to other chunks)
```

```
@
Documentation for Chunk B.
<<Chunk B>>=
(TeX code / references to other chunks)
```

Documentation chunks begin with the line that starts with @ followed by space or newline. Code chunks begin with <<Chunk name>>= on a line by itself. Chunks are terminated by the beginning of another chunk or end of file.

Making changes/updates. The change file mechanism is not needed in the case of the TeX language. Change files are used to incorporate system dependent code into a source file, but TeX code is already system independent: TeX code could only be ‘format dependent’. Another feature of the format file is that it evolves with time, but the intermediate versions are used for preparation of books, articles etc. All these versions and configurations must be kept well organized, otherwise you are bound to be lost. The Revision Control System (RCS) is an appropriate tool to assist with these tasks. With RCS it is possible, with small overhead, to preserve *all the revisions* which evolved from a given text document, to merge changes made by others, to compare different versions, and to keep a log of changes.

RCS

```
ci foo.tw           (check-in latest version)
co foo.tw           (check-out latest version)
co -r<rev> foo.tw
rlog foo.tw
rcsdiff -r<rev> foo.tw
rcsmerge -r<later_rev> -r<earlier_rev> foo.tw
```

When the first command is executed `foo.tw` is stored in a *group file* (with default name `foo.tw,v` on UNIX machines, or `foo.tw%` on MSDOS) as a new revision. For each revision you deposit, `ci` prompts for a log message. The file `foo.tw` is deleted unless you ask otherwise (`ci -l foo.tw`). The message “`ci error: no lock set by (login)`” tells you that RCS was configured with the ‘strict locking feature’ enabled. Locking prevents clashes between different users’ modifications if several are working on the same file. This feature is disabled by the command `rcs -U foo.tw`; it is unnecessary if only the owner of the file is expected to deposit revisions into it.

The next two commands are used to extract the latest, or the specified, revision from the group file. `rlog` is used to print log messages. Different revisions of a document may be compared using `rcsdiff`. The command `rcsdiff foo.tw` compares the latest revision with the contents of the working file. The differences themselves are found by the

program `diff`; if you do not like `diff`’s default output, change it by passing appropriate switches to `rcsdiff`. The last command undoes the changes between revisions; the file `foo.tw` will be overwritten. `rcsmerge` incorporates changes between two revisions into the working file. A similar effect could be achieved with a stand-alone program called `merge`. If files being compared are `mine`, `older`, `yours` then given the command

```
merge mine older yours
```

`merge` tries to add to `mine` the result of subtracting `older` from `yours`; if overlap occurs, i.e., both files `mine` and `yours` have changes to the same segment of lines in `older`, then `merge` delimits the alternatives with

```
<<<<<< mine
(lines in) mine
=====
(lines in) yours
>>>>>> yours
```

and writes above to `mine`. Now it is up to you which set of changes you adopt. `merge -p ...` sends the result of merging to the standard output.

To keep the working directory uncluttered, all RCS files are usually stored in the subdirectory with the name `RCS`. RCS commands look first into this directory when searching for files.

Concluding remarks

It seems that the TeX language constitutes a good starting point for exploring the idea of literate programming. The system is simple, because many features present in other WEBs are not needed. The system is extensible, which means that it is possible to try different styles and features. And finally, programs written in TeX are not too long — `plain.tex` is about 1000 lines of code — which means that you can print the documentation of real programs yourself and share it with others.

For those convinced by the analysis above, the literate source of `plain.tex` has been submitted to the CTAN archives, in directory `web/tweb`; please read it and enjoy.

Teaching CS/1 Courses in a Literate Manner

Bart Childs

Department of Computer Science
Texas A&M University
College Station, TX
USA
Email: `bart@cs.tamu.edu`

Deborah Dunn

Department of Computer Science
Texas A&M University
College Station, TX
USA
Email: `debbie@cs.tamu.edu`

William Lively

Department of Computer Science
Texas A&M University
College Station, TX
USA
Email: `lively@cs.tamu.edu`

Abstract

The first course in Computer Science is often called ‘CS/1’ based upon the designation in curriculum recommendations. The content of CS/1 courses often shows that it should include a significant amount of documentation, problem solving, problem formulation, . . . Experience has shown that instructors often slide into almost total emphasis on language syntax. Ask the student who has taken such a class as to its content and the answer usually comes back like “It was a C (or Pascal or . . .) course.”

We will report on an experiment of teaching the honors section of our first course at Texas A&M University using Knuth’s `WEB`. The primary advantage we saw in the use of the system is that the `WEB` system would enable the progression through the problem solving methodology by editing and extending the same document.

Our analysis of data obtained by tracking the students in later semesters shows significant benefit from the use of literate programming. We found little or no problem using `emacs`, `TeX`, `WEB`, and requiring **documentation** after the initial scares in the course. We will describe how we taught the course, present performance statistics, and outline our recommendations for pursuit of similar goals. Finally, we will outline our longer range goals with the use of similar systems.

Introduction

We embarked on a project to teach the first computer science course (CS/1) (Denning, Comer, Gries, Mulder, Tucker, Turner, and Young, 1989; Tucker, 1990) using literate programming (Knuth, 1984) and still covering all the topics covered in the usual sec-

tions (Dunn, 1995). The parallel sections used Turbo Pascal and its supporting environment.

CPSC 110 is entitled “Programming I”. The catalog description does not specify the languages to be used, but we normally use English and Pascal. A few years ago we tried C instead of Pascal

but have never tried any substitutes for English except \TeX ian (although some students may argue the point). The reasons that the C experiment were a failure will not be addressed in this paper.

An inherent part of these CS/1 courses is to develop the student's skills in *problem solving*. Indeed, in many course outlines, that is part of the title and the main emphasis in the description of the course contents. A problem solving methodology is often stated in CS/1 courses which generally has steps like:

1. State the problem completely!
2. Develop all necessary assumptions.
3. Develop an algorithm and test data set(s).
4. Code the problem.
5. Analyze the results (and iterate?).

Literate programming is a style in which the design of the code reflects that the human reader is as important as the machine reader. The human reader is often associated with the expensive process of maintenance and the machine reader is the compiler/interpreter. Literate programming is a process which should lead to more carefully constructed programs with better, relevant 'systems' documentation. We think that the first sentence in this paragraph should be particularly relevant to students because the human reader (the one who assigns grades) is obviously the most important reader.

The features of literate programming that gave us the confidence to expect positive results are:

1. top-down and bottom-up programming since it is a structured pseudo-code,
2. programming in small sections where most sections of code and documentation (section in this use is similar to a paragraph in prose) are approximately a screen or less of source,
3. typeset documentation (after all, Knuth was rewriting \TeX),
4. pretty-printed code where the keywords are in bold, user supplied names in italics, . . . , and
5. extensive reading aids are automatically generated including a table of contents and index.

We offer these comments about the above list. We repeat the item numbers for clarity:

1. these topics are usual in CS/1 books but they generally lack the integration to make them really effective for the student,
2. divide and conquer is also espoused but the larger examples that are furnished in many books forsake the principle,

3. it may be argued that this is 'feeding pearls to the swine' but we like the cognitive emphasis that comes from logical substitution of words for key-words . . . ,
4. the fact that **weave** breaks lines based on its parsing is another cognitive reinforcement,
5. encouraging/requiring students to review their programs as documents makes them **think** about readability.

Problems with 'Problem Solving'

Researchers have found that many of the difficulties experienced by novice programmers are not a result of misunderstanding the language constructs, but a result of problems with "putting the pieces together" (Spohrer and Soloway, 1986). Thus, the process by which programs (and documentation) are developed should be examined.

Linn and Clancy (1992) state that a good programmer needs both a knowledge of the programming language *and* good problem solving skills. Introductory courses tend to emphasize programming; that is, the *product* of good design and development. Although this is obviously an important aspect of programming, the real problems exist in the design of problem solutions (Linn, Sloane, and Clancy, 1987). Few textbooks used in the introductory courses actually emphasize teaching the student how to develop good design solutions (Linn and Clancy, 1992), regardless of the university catalog description.

Linn, Sloane, and Clancy (1987) found, in teaching program design, that teachers who discuss how they solve problems, including their interpretation of the problem statement, are more effective than those who present just the subject matter. Studies have shown that explicit teaching of problem solving strategies greatly influences learning (Linn and Dalbey, 1985; Linn, Sloane, and Clancy, 1987).

Soloway (1986) states that *goals* and *plans* are the two key components in the task of representing problems and solutions to a problem. Problem solving, and hence learning to program, requires that students learn to construct *mechanisms* and *explanations* for those mechanisms. Students are led to believe that programs are the output from the programming process. Rather, they must be made to understand that programming is a design discipline. Instead of the programming process being viewed as a program, it should be viewed as "an artifact that performs some desired function" (Soloway, 1986).

Soloway and colleagues (Soloway, Ehrlich, Bonar, and Greenspan, 1982; Spohrer and Soloway,

1986) have studied *bugs*—errors in programs—and *misconceptions*—misunderstanding in the minds of novice programmers—in an attempt to identify the needs of novice programmers by understanding the kinds of mistakes they are likely to make. Because there are many ways to solve a given problem, bugs are identified using a *goal/plan analysis*. Goals are what is to be accomplished and plans are those stereotypical sections of code that are used to achieve the goal. Thus, bugs are the differences between the correct plans and the incorrect implementations used by novices (Spohrer and Soloway, 1986).

Soloway believes a program has two audiences (Soloway, 1986), as shown in Figure 1. Soloway uses this to conclude: “learning to program amounts to learning how to construct mechanisms and how to construct explanations” (Soloway, 1986).

The Audiences for a Program
◇ <i>The computer</i> , which, based on instructions is a <i>mechanism</i> for <i>how</i> a problem is solved.
◇ <i>The human reader</i> , who needs an <i>explanation</i> for <i>why</i> the program solves the problem.

Figure 1: Program Audience

“Are You Crazy!?”

The title of this section was frequently shouted at us because, *everybody in the world knows*:

- emacs is impossible to learn and use,
- T_EX is impossible to learn,
- WEB’s steps make it too many steps to learn, and
- there is a reason for all those Aggie jokes.

and **therefore** our project was doomed!

Well, we are Aggies and we decided to try teaching CS/1 using all those horrible things. We exercised a little judgement and did it on the smallest sections of the course, namely the honors sections.

The next four subsections are the things that we did that are different from our usual CS/1 course. They are presented in the order that the students saw them.

Testing The first meeting of the laboratory included some quizzes that did not affect the grade but were done to determine the students’ backgrounds.

During the semester there were some different questions on tests that addressed problem solving more than usual.

Introductory computer science students have difficulty viewing programming as a means by which

we solve problems. Computer science instruction, at the introductory level, tends to emphasize programming, which is the *product* of problem solution design (Linn and Clancy, 1992). Most textbooks give examples of programs, rather than demonstrate the method by which the given solution was derived (Linn and Clancy, 1992).

It has been said the use of literate programming allows us to associate a given design step with its consequences, that is, the resulting code (van Ammers, 1993). Students should be taught that problem solution *design* leads directly to the result, which is the program. The use of literate programming encourages the inclusion of the design step in the source of the resulting program.

The results of the research were used to determine whether improvements in problem solving and programming skills can be attributed to the use of literate programming. An evaluation of the teaching methodology was made based on several factors:

1. Completion of a pre-test which was developed to indicate the students’ problem solving ability and computing background as they entered the course.
2. Periodic tests which were designed to indicate the change in problem solving ability and programming skills.
3. An evaluation of the programs and documentation produced and the consistency between code and its corresponding documentation.
4. Completion of a post-test which indicates the students’ ability to solve problems and write programs at the end of the test period.
5. An evaluation of the students’ performance in the subsequent Programming II course.
6. An evaluation of the students’ performance in the subsequent Data Structures course.

The results were expected to indicate an increase in problem solving ability over time. Programmers who use the literate programming paradigm were expected to be more *problem-oriented* rather than *program-oriented*.

Emacs and web-mode We decided to use Mark Motl’s **web-mode** environment (Cameron and Rosenblatt, 1991; Motl, 1990). This keeps the neophyte (and expert) user from making a number of simple mistakes that are easily committed. It relieves the user of knowing how and where to insert that necessary T_EX mumbo-jumbo that WEBs start with; it ensures matched @< and @> pairs (complete with the “=” when needed); and allows selection of existing section names (rather than having to type it identically).

At a later time, it will help them in other ways by allowing navigation in terms of WEB terminology.

Emacs was introduced with a one page hand-out and a modified version of the ‘Emacs Reference Card’. The reference card is printed on 8.5" × 11" paper, two sided, with three panels on each side. The modifications were to remove some of the more advanced features of Emacs and replace them with **web-mode** features. Emacs was covered in one hour of laboratory time with some questions and answers at the start of subsequent periods.

Knuth’s WEB The subsequent courses in our curriculum are based upon having some use of the Pascal language in CS/1. Thus, we selected a current implementation of Knuth’s original WEB (Knuth, 1983), which is Pascal based.

The current implementation means that we did not convert the output of **tangle** to all upper case, shorten variable names, nor remove underscores. The resulting output of **tangle** is still relatively “unfit for human consumption”.

The rules of WEB were covered by the use of a five page “memo” from the WEB distribution, Knuth’s “WEB User’s Manual”.

How The Course Was Taught The focus of the semester was on problem solving. The students were taught Pascal syntax, but the emphasis was on problem solving using the WEB style of programming. A portion of the class was spent on learning (and evaluating) problem solving skills for the design and development of programs. One method by which problem solving was taught was by example. The students were given several examples of how to design solutions to a problem. This technique of problem solving with examples was used throughout the semester as the difficulty of the problems increased.

An important part of learning problem solving was to practice iteration in the design of a solution. An iteration of the students’ problem solution was evaluated by the teaching assistant. The students received feedback regarding their iterative process, such as whether they were approaching the details of the problem at an acceptable level and whether they were considering all aspects of the problem.

The final measurement in the design and development phase was made upon completion of the program assignment. Each program was examined and an evaluation made as to the correctness of the solution, the consistency of documentation and code, and the quality of the documentation. The intent was to determine if the documentation portion of a section was, in fact, an explanation of the code.

Do All Labs Twice We were particularly fortunate that when the curriculum was revised and a formal laboratory was added, the professor in charge decided that the laboratory meeting would not be one extended period, but two one hour periods with a day in between.

We used this to great advantage to require that each lab to be turned in for grading twice:

- Do the first three parts of the problem solving procedure outlined above **without any code!** We wanted the student to document that the problem was understood! WEB can be considered to be a structured system of pseudo-code and is therefore ideal for this purpose.
- The 47 hour lapse between laboratory meetings enabled the grading of those important first steps of problem solving to assist the student in understanding what is to be converted to code.

Initial Reactions and Thoughts

Students in first year courses are often rather intimidated but generally ready for any challenges that might arise. Of course we have that same experience. It is interesting to note those that were not ‘the usual’.

CS/1 courses will often have a number of students who have had one to five years of computer experience, much of it unstructured. We certainly had our share.

There were a number of students who had at least one year of the use of Turbo Pascal in secondary school and who had obviously used it significantly outside that educational environment. Testing showed these students to have two general characteristics:

- a lack of understanding of how to state a problem;
- a great desire to do nothing other than use Turbo Pascal.

It was also common for these students to react in rather vigorous ways. We think that it is a characteristic of those in the programming professions to resist change unless it is change that they tried to start.

Nearly half the class indicated no programming background from their secondary training. (They may have had word processing, spread sheets, and computer math; but they indicated no programming. Further, they were frequently not CS majors.)

Thirty-eight students enrolled in the honors class during the Fall 1993 semester. The administration of a pre-test provided information regarding

the general background and experience of the participants. The purpose of the pre-test was to establish that these were, in fact, novice programmers. The results of the problem solving portion of the test provided a basis for measuring the initial problem solving skills of the participants.

The students entered the course with a variety of backgrounds in computer science. Only one student had never taken a computer science course and one student had taken only a computer literacy/computer history course. Few of the students had any background in computer science at the college level. Table 1 is a summary of the college level experience of the participants.

Table 1: Unusual or Exceptional Computer Experience of Subjects

Count	Exceptional Experience
1	C course at a Junior College
4	University level Fortran course

The majority of the students had some type of computer science class in high school. Table 2 is a summary of the high school experience of the participants. Although there were thirty-eight students enrolled in the class, many of the students had experience in more than one of the areas listed.

Table 2: High School Computer Experience of Subjects

Count	Computer Experience
8	Microcomputer applications, typically including DOS, WordPerfect, Lotus 1-2-3, and/or dBase
8	Computer Math, which may or may not include some experience in BASIC and/or Pascal
12	BASIC course
21	One or more semesters of Pascal

Despite the appearance of having a significant background in computers, these students must still be considered novice programmers. Although a significant number had some background in Pascal programming, fifteen felt they could program without the use of a reference manual. Even so, their knowledge of advanced Pascal constructs cannot be considered to be comprehensive. None of the students had experience as a professional programmer.

One student had limited experience with the emacs editor. The remaining thirty-seven had no

experience with emacs. None of the students had heard of WEB programming; therefore, none of the test study participants had previous experience with literate programming.

The pre-test included a question designed to provide some measurement of the students' initial problem solving ability. The students were asked to state the steps necessary to solve a given problem. They were instructed to give detailed answers in complete English sentences and paragraphs. The problem was stated as follows:

You are the manager of Aggie Lawn Service. Alvin is your new employee. You must explain to Alvin the process of calculating an estimate for a potential customer. (Of course, in the future this may use a hand-held computer.) The quote will include a cost statement and estimated time to complete the job.

This estimate is based upon the area of the lawn and a standard (confidential) charge per square foot. Grass can be cut at the rate of 2 square feet per second. You may assume that a rectangular house is situated in a rectangular yard. Give the details of the process and itemize all assumptions you have made.

It is difficult to measure a person's problem solving ability. For example, if it is easily seen that the problem is a basic input-process-output problem, then each subject should receive points if the necessary inputs and the required outputs were described. In terms of the processing, many students felt it was sufficient to merely give the formula for the area of a rectangle. They then subtracted the area of the house from the area of the lawn (sometimes shown, again, as a formula).

In general, most of the students were able to give an answer which solved the problem. However, several exceptions were noted as follows:

- some participants simply gave the necessary formula(s), omitting any description of the inputs and/or outputs;
- some participants failed to describe their solution using complete English sentences and paragraphs;
- some participants described the necessary inputs and the required processing, but failed to produce a result; and
- some participants made and described additional assumptions or expressed a need for additional information regarding items such as driveways, sidewalks, trees, flower beds, etc.

The students' solutions were scored based on their ability to solve the problem. Table 3 is a summary of the minimal set of problem solving issues that should have been addressed or noted, with their associated point value.

Table 3: Problem Solving Issues

Points	Problem Solving Issue
2	Obtain dimensions of yard
2	Obtain dimensions of house
2	Calculate area for house and yard
2	Calculate area for lawn to be cut
3	Calculate total cost to cut the lawn
3	Calculate the time for completion
2	Convert the time to minutes or hours
2	Produce the final cost for cutting lawn
2	Produce the time for completion

A final score of twenty indicates that the student adequately described the required inputs, calculations, and necessary outputs. A student lost points for omitting information or not describing the process in sentence form. A student could earn extra points by addressing issues that were not explicitly mentioned, but might be a factor in solving the problem.

Table 4: Initial Problem Solving Ability

Percent of Students	Problem Solving Ability
31.6	Excellent (18+ points)
15.8	Above average (16-17 points)
21.1	Average (14-15 points)
13.2	Below average (12-13 points)
18.4	Poor (below 12 points)

Table 4 is a summary of the results of measuring the students' initial problem solving ability. There are 47.4% above and only 31.6% below average. The grade of "C" is described as average, yet it is rare that a class will have as many D's and F's as A's and B's. The distribution of the data in Table 4 is consistent with grade distributions for the CS/1 course over the last few years.

Results

We feel the background of the students is not atypical of many CS/1 type courses. The majority of the class are majoring in computer science, but a significant number are using the course as a minor

elective, a basis for deciding if they want CS as a major, or other reasons.

Some results will be presented with this diversity as an identifying factor. Results will also reflect the tracking of the students in subsequent courses and differences between other semesters of the same course.

Performance during the semester The actual scores received by the test group on the problem solving portion of each test are included in Appendix D. The mean of the scores for the problem solving portion of each test are shown in Table 5.

Table 5: Mean Problem Solving Scores – Tests (Percent)

Test	Overall	Majors	Non-Majors
Pre-Test	72.6	74.0	68.3
Test 1	78.8	79.7	76.1
Test 2	66.6	65.7	71.6
Test 3	80.9	80.3	82.7
Post-Test	76.6	76.2	77.8

It is difficult to determine whether or not the problem solving skills for the test group increased over the course of the semester. The class, as a whole, experienced a decrease in scores on the second test, although there was a greater decrease for computer science majors. This decrease in scores for the second test may be attributed to the fact that the problem for that test was significantly different and more difficult than any of the problems encountered previously during the lab or on a test. The scores also decreased on the post-test, or final exam, as compared to the third test; however, they still improved as compared to the scores on the pre-test.

The problem solving scores, as a whole, were higher on the labs than they were for the exams. This was to be expected since the problem solving portion of the lab was not developed under stressful situations, as in the test-taking scenario. Another reason for having higher scores in the lab is that measuring problem solving skills is not something we are used to doing on a test. It is much easier to evaluate someone's problem solving skills developed through iteration during lab than it is to evaluate one-time problem solving skills on a test.

Table 6 is a summary of the overall grade distribution for students completing the CS/1 course for the subject and comparison classes (in percent form).

The percentage of students that passed the CS/1 course was similar for each of the classes. A

Table 6: Overall Grade Distribution (Percent)

Semester	A	B	C	D	F
Fall 90-H	20.6	50.0	14.7	5.9	8.8
Fall 92-H	51.3	20.5	20.5	2.6	5.1
Fall 93-H	24.3	40.5	21.6	5.4	8.1

grade of “A”, “B”, or “C” is considered passing. The Fall 1990 and the Fall 1992 comparison groups had 85.3% and 92.3% of the students, respectively, pass the course. The test group had 86.4% of the students pass the course.

Performance in CS/2 Approximately 65-70% of the honors CS/1 students enrolled in the CS/2 course (73.5% of the Fall 1990 class, 66.7% of the Fall 1992 class, and 67.6% of the Fall 1993 class).

Table 7 is a summary of the overall grade distribution for the subsequent CS/2 course for those students in the subject and comparison classes in percent form.

Table 7: Overall CS/2 Grade Distribution (Percent)

Semester	A	B	C	D	F
Fall 90-H	68.0	28.0	4.0	0.0	0.0
Fall 92-H	73.1	19.2	7.7	0.0	0.0
Fall 93-H	52.0	40.0	4.0	0.0	4.0

At first glance it appears that the students in the Fall 1990 honors and the Fall 1992 comparison classes performed much better than the students in the test study group in the CS/2 class. Both of the comparison groups had a higher percentage of students make “A”s in the subsequent course. However, all of the classes had over 90% of the students make an “A” or a “B” in the course.

Table 8 is a comparison of the average grades in the CS/1 class and the subsequent CS/2 class for those students in the subject and comparison classes. The grade point shown is out of a total possible grade of 4.0. The Mann-Whitney U-test was used to conclude that there is not a significant difference in average grade point ratio for any of the groups.

This may still not be a good representation of how the students in the subject and comparison classes performed in the subsequent course. These grades can be evaluated in terms of the particular section and semester the class was taken and the instructor that taught the class.

Table 9 is a summary of the average difference in grades between the subject class, the comparison

Table 8: Average Grade for CS/1 and CS/2 Courses

Semester	CS/1	CS/2
Fall 90-H	2.676	3.640
Fall 92-H	3.103	3.654
Fall 93-H	2.676	3.360

classes, and the other CS/2 classes. This summary is itemized by section, instructor, and semester.

Table 9: Average Difference in Grade for CS/2 Classes

Semester	Difference in Section	Difference in Instr.	Difference in Semester
Fall 90-H	+0.02	+0.01	+0.01
Fall 92-H	+0.03	+0.01	+0.01
Fall 93-H	+0.06	+0.05	+0.09

With these figures, it is shown that the students in the CS/1 comparison classes scored somewhat higher than their peers in the same section of the CS/2 course. However, those students in the CS/1 test group scored even higher than their peers in the same sections of the CS/2 course. This data was also analyzed including the CS/2 instructors and semester. The same results held.

When the performance of the students in the test study group was compared with the performance of their peers, it was determined that the students in the test study group actually scored higher than the students in the comparison groups (and the other students) in the CS/2 course.

Performance in Data Structures Approximately 45-55% of the honors CS/1 students enrolled in the Data Structures course (55.9% of the Fall 1990 class, 56.4% of the Fall 1992 class, and 45.9% of the Fall 1993 class).

Table 10 is a summary of the overall grade distribution for the Data Structures course for those students in the subject and comparison classes in percent form.

Table 10: Overall Data Structures Grade Distribution (Percent)

Semester	A	B	C	D	F
Fall 90-H	21.1	63.2	15.8	0.0	0.0
Fall 92-H	50.0	13.6	22.7	9.1	4.5
Fall 93-H	52.9	35.3	11.8	0.0	0.0

Not only did the test study group have a larger percentage of students make an “A” in the course, but a larger percentage of students made an “A” or a “B” in the course.

Table 11 is a comparison of the average grades in the CS/1 class, the CS/2 class, and the Data Structures class for those students in the subject and comparison classes. Again, the grade point shown is out of a total possible grade of 4.0. Using an unpaired t-test, with $\alpha = 0.10$, it was concluded that there is a significant difference in average grade for the Data Structures course between the Fall 1993 test group and both the Fall 1990 and the Fall 1992 comparison groups.

Table 11: Average Grade for CS/1, CS/2, and Data Structures Courses

Semester	CS/1	CS/2	DS
Fall 90-H	2.676	3.640	3.053
Fall 92-H	3.103	3.654	2.955
Fall 93-H	2.676	3.360	3.412

A chi-square test of independence was conducted to determine if the grade and CS/1 semester variables are related (or dependent). The critical value of X^2 for $\alpha = 0.10$ and degrees of freedom = 8 is 13.36. The computed value, 15.368, exceeds 13.36, so we conclude that the two variables are dependent. That is, the proportion of students receiving a particular grade varies depending on the semester in which they took CS/1.

When the performance of the students in the test study group was compared with the performance of their peers in a course which requires extensive problem solving skills, it was determined there is a significant difference in the performance of the students in the test study group compared with the performance of the students in the comparison groups.

Too bad that few (if any) were still using lp.

Student Evaluation of CPSC 110 Teaching Methodology Upon nearing completion of the CS/1 course, the students were asked to submit a paper reflecting their feelings and attitudes towards the WEB programming methodology. It was stressed that statements made would in no way affect their grade in the course. This was to be written as a typical one-page technical note.

Three people evaluated the reaction of the test subjects. None of the people had prior training in rating. A rating scale (Meister, 1985) was developed and the reports were evaluated in order to appraise the students’ reactions to the WEB programming pro-

cess. The scale consisted of five categories, rated 1-5 and a 0 value that was taken to mean no response.

Below is a summary of the results of the rating process. The mean of the scores (columns R-1, . . . , R-3) for each of the three volunteers who rated are shown in Table 12. Kendall’s coefficient of concordance (Meister, 1985) was used to test agreement between the ratings. The result was a value of 0.673, which indicates there was a modest level of agreement between the evaluations of the questionnaire results.

Table 12: Evaluation of Fall 1993 CPSC 110H Students’ Reactions

Question	R-1	R-2	R-3	Overall
1	3.21	2.81	2.52	2.85
2	3.20	1.60	3.25	2.67
3	2.69	2.64	1.87	2.43
4	3.44	3.14	1.67	2.88
5	3.41	3.28	2.90	3.20
6	3.54	2.87	3.57	3.31

The questions and some comments of interpretation were:

1. What was your original reaction to being told you were going to learn something called WEB?

Although a few of the students were enthusiastic about the idea, many were unhappy with the fact that they were going to be using a different methodology. Much of the unhappiness was due to the fact that many of the students entered the course with prior expectations about what is taught in the class.

2. What was your expectation of the course?

Most students entered the course under the impression that CPSC 110H was a course in Turbo Pascal, despite the course description.

3. What was your reaction to emacs?

Many of the students objected to the use of the emacs editor. This may be due to the fact that the user interface is not extremely user-friendly, especially to the novice user. The students were required to use predefined keystrokes, rather than pull-down menus. (This was apparently because the question was asked. It did not show up on the course evaluation.)

4. What was your reaction to T_EX?

Although a minimal amount of T_EX knowledge is required, the students seemed to find the language difficult. Although several examples were provided, with a variety of T_EX commands, they students did not seem to adapt

well to the use of \TeX . Despite the lack of \TeX knowledge, the students seemed to adapt to the \WEB environment. (Same comment?)

5. What was your reaction to \WEB programming?

The evaluators of the students' reaction seem to believe this response was a bit above average. The lack of enthusiastic response may have been due to their overall difficulty in understanding the \WEB process and concepts.

6. What was your reaction to the overall \WEB process/concepts?

Generally, the students' understanding was average to good. Many of the students continued to have difficulty separating the concepts of editor, \WEB files, \TeX commands, etc. Some seemed overwhelmed in the beginning with having to learn more than just 'a language'.

There were certainly a number of students who can code but did not catch the 'big picture'. Comments like "why document when you have the code to read" were not uncommon.

For the purposes of table 12, answers to the questions (except question 2) were 'not discussed'; poor; fair; average; good; and excellent, receiving numeric values of 0, . . . , 5 respectively. The answers to question 2 varied from 'unknown \equiv 0' to 'Turbo Pascal \equiv 3' to 'Problem Solving and Programming \equiv 5'.

Conclusions

We taught an honors section of a CS/1 course in a different manner than usual, namely using literate programming. The students used an editor, a formatting system, and a coding style that was new to all. The students' performance in subsequent courses was not hurt and may have been helped with the different methodology. The results of using the program development methodology in the CS/1 course indicate that the methodology is successful in teaching novice programmers good problem solving skills.

These are the results of the experiment:

- The students showed an increase in their problem solving skills.
- Those students unfamiliar with the Pascal programming language, or any other programming language, were more successful than those familiar with Pascal at using the literate programming paradigm to capture and document their problem solving process.
- The students were able to learn the \WEB rules, the \web-mode environment, GNU Emacs, and

\TeX rules, as well as the Pascal syntax and constructs.

- Those students exposed to the program development methodology utilizing the literate programming paradigm were as successful in the subsequent CS/2 course as those not exposed to the methodology.
- Those students exposed to literate programming were significantly more successful in the Data Structures course than those not exposed to the methodology.
- The subject program development methodology may lead to an improved software development process; however, more tests should be conducted.

Norman Ramsey (author of Spider and NoWEB) has recently presented a position paper at an ICSE '95 workshop entitled "Literate Programming should be a model for Software Engineering and Programming Languages", dated March 1995. While we are in agreement with the obvious philosophy, we are concerned that it is too late because we have observed first year students are already like the professionals: "No, I do not want to learn anything new if I already have some knowledge in the area". We think it should appear early in the curriculum and repeatedly.

Recommendations and Future Use

In teaching a CS/1 course, you can do darned near anything and succeed. You just have to keep your eye on the goal, don't apologize, and push! We have known of real problems because the professor (assigned to the class at the last minute) often is less familiar with the specifics of 'Turbo Pascal' than many of the students. The lesson from that is "give them a new challenge". We feel that there is a real benefit to the use of literate programming and requiring that students practice writing, using of pseudo-code, and documenting their programs. It is a new challenge to them.

The following is a list of things we wish we had or recommend to similar projects:

1. It sure would be nice if we have some video training on how to do some of these things, particularly simple \TeX , emacs, and DOS.
2. Now that we have MetaPost and other drawing packages, we think that more diagrams should be included in the first lab attempts.
3. Make students write! Repeat "**Make students write!**"
4. This is a natural for "cooperative learning"; have the students do **extensive** peer review.

5. It would be easier if we had some tools that would extract a component of the grade based upon “user supplied” index entries, long variable names, variable names made of words from the dictionary, . . . Then the person should add significant markup from reading the English.
6. Assign some labs that are extensions of previous work to let them see what maintenance is. If you have time, let them work on programs done with and without literate programming. Otherwise, they will never learn! This can be difficult in a first course, but should always be part of later courses.

References

- D. Cameron and Rosenblatt, B. *Learning GNU Emacs*. O'Reilly & Associates, Inc., Sebastopol, CA, 1991.
- P. J. Denning, Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R. “Computing as a discipline”. *Communications of the ACM* **32**(1), 9–23, 1989.
- D. L. B. Dunn. *Literate Programming as a Mechanism for Improving Problem Solving Skills*. Ph.D. thesis, Texas A&M University, College Station, TX, 1995.
- D. E. Knuth. “The WEB system of structured documentation”. Stanford Computer Science Report CS980, Stanford University, Stanford, CA, 1983.
- D. E. Knuth. “Literate programming”. *Computer Journal* pages 97–111, 1984.
- M. C. Linn and Clancy, M. J. “The case for case studies of programming problems”. *Communications of the ACM* **35**(3), 121–132, 1992.
- M. C. Linn and Dalbey, J. “Cognitive consequences of programming instruction: Instruction, access, and ability”. *Educational Psychologist* **20**(4), 191–206, 1985.
- M. C. Linn, Sloane, K. D., and Clancy, M. J. “Ideal and actual outcomes from precollege Pascal instruction”. *Journal of Research in Science Teaching* **24**(5), 467–490, 1987.
- D. Meister. *Behavioral Analysis & Measurement Methods*. John Wiley & Sons, Inc., New York, 1985.
- M. B. Motl. *A Literate Programming Environment Based on an Extensible Editor*. Ph.D. thesis, Texas A&M University, College Station, TX, 1990.
- E. Soloway. “Learning to program = learning to construct mechanisms and explanations”. *Communications of the ACM* **29**(9), 850–858, 1986.
- E. Soloway, Ehrlich, K., Bonar, J., and Greenspan, J. “What do novices know about programming?”. In *Directions in Human-Computer Interaction*, edited by B. Shneiderman and A. Badre, pages 27–54. Ablex Publishing Corp., Norwood, NJ, 1982.
- J. C. Spohrer and Soloway, E. “Novice mistakes: Are the folk wisdoms correct?”. *Communications of the ACM* **29**(7), 624–632, 1986.
- A. B. Tucker. “Computing Curricula 1991—Report of the ACM/IEEE-CS Joint Curriculum Task Force”. Technical report, Association for Computing Machinery, New York, NY, 1990.
- E. W. van Ammers. “Communication on July 16, 1993 at 7:05 CDT”. Literate Programming Mailing List, 1993. Email: ammers@rcl.wau.nl.

An Audio View of (L^A)T_EX Documents — Part II

T. V. Raman*

Digital Equipment Corporation

Cambridge Research Lab

One Kendall Square, Building 650

Cambridge, MA 02139, USA

Email: raman@crl.dec.com

URL: <http://www.cs.cornell.edu/Info/People/raman/raman.html>

Abstract

A_ST_ER — Audio System For Technical Readings — is a computing system that produces audio renderings from the *same* (L^A)T_EX source used to produce the printed document. Raman (1992) described our preliminary work on this project. At the time, correct handling of user-defined (L^A)T_EX macros was described as one of the key issues in building a fully extensible audio rendering system. A_ST_ER (Raman, 1994) has now been fully implemented. This paper reports on the approach used in A_ST_ER to handle user-defined macros.

The approach used not only makes A_ST_ER fully extensible; it points out a unique advantage of (L^A)T_EX — the ability of the author to encode semantic meaning into the markup by extending the document model in ways appropriate to the specific document instance that is being encoded.

Introduction



A_ST_ER — Audio System For Technical Readings — is a computing system that aurally renders electronic documents marked up in the (L^A)T_EX family of markup languages (Raman, 1994). A_ST_ER uses the structural markup present in the electronic source to advantage in producing high-quality, interactive audio renderings. This paper focuses on a specific aspect of the problem; namely that of flexibly rendering the extended document logical structure encapsulated in a (L^A)T_EX document.

One primary advantage of (L^A)T_EX is the flexibility it provides the author in defining logical structures that are specific to a particular document instance. In this sense, the class of logical structures that can be encapsulated in a (L^A)T_EX document is extensible. (L^A)T_EX macros allow an author to abstract away the layout details. At the same time, they provide a powerful mechanism for defining new constructs that are not already present in the document style (DTD in SGML parlance) in use. As a consequence, when introducing a new piece of math-

ematical notation, an author can first define a new (L^A)T_EX macro that produces a desired layout, and then use this newly defined construct throughout the document.

The flexibility of the (L^A)T_EX macro facility initially proved a major stumbling block in building a fully extensible audio rendering system. A system that attempts to produce aural renderings by *mapping* the builtin (L^A)T_EX commands to an equivalent aural representation faces the severe shortcoming of not being able to render documents that contain user-defined macros. At the same time, it is impossible to translate such user-defined (L^A)T_EX macros into a suitable aural representation. This is because T_EX in its full glory is a Turing-complete programming language, and saying “we can translate a general T_EX macro to audio” is equivalent to saying that “Given a T_EX program, we can predict the result”. Being able to achieve the above without actually running T_EX on the program (document fragment) would amount to being able to solve the Halting problem!

In the rest of this paper, we describe the solution used in A_ST_ER to circumvent this difficulty. The solution we used in fact turns the presence of user-definable (L^A)T_EX macros into an advantage. Such user-defined constructs allow A_ST_ER to glean even more information about the document logical structure than would be possible if the document were

* *Now at:* Adobe Systems, Advanced Technology Group, 1585 Charleston Road, Mountain View, CA 94039-7900; Email: raman@adobe.com

encoded using only the built-in (L^A)T_EX operators; as a consequence, the audio renderings produced are also significantly better.

Document Models in A_ST_ER

A_ST_ER produces audio renderings by first extracting the document logical structure. In this model, all forms of rendering, i.e., visual, aural, etc. are regarded as a projection of the structure present in the information being conveyed onto the medium being used to communicate the information. Thus, typesetting a document requires visual formatting — projecting the information structure onto a two-dimensional visual tablet; aural rendering requires presenting the structure using various features of the auditory display.

The recognizer used in A_ST_ER extracts logical structure present in documents encoded in the (L^A)T_EX family of languages. An important feature of this recognizer is that it works on the entire gamut of encodings, ranging from plain ASCII documents, i.e., no explicit markup, up to documents containing completely unambiguous encodings of the logical structure.

The basic document model used in A_ST_ER is the attributed tree. Each hierarchical level of the document is modeled as a node in this tree. Each node can have content, children and attributes. Using object-oriented terminology, each different kind of node of the tree is called an *object* and represents a document element. Thus, “chapter”, “section”, “paragraph”, and “sentence” are all objects. If a document contained five sections, its representation in A_ST_ER would have five instances of object “section”. This object-oriented terminology is used because A_ST_ER actually uses CLOS objects in this fashion. The use of an object-oriented language was instrumental in allowing us to develop and implement the ideas in A_ST_ER incrementally and effectively.

This attributed tree structure is augmented to represent mathematical content; we call this augmented representation the *quasi-prefix form*, (see figure 1 above). Expressions that are completely unambiguous, e.g., $x + y$, are captured in their prefix form. In addition to linearizing the underlying tree structure, mathematical notation uses *visual attributes* such as superscripts and subscripts, whose interpretation is context-dependent. We extend the prefix form to capture such visual attributes — hence the name *quasi-prefix*.

The next section describes how this model is extended to encapsulate the use of user-defined constructs in (L^A)T_EX.

Extended Logical Structure

The (L^A)T_EX facility can be used to extend the document logical structure by defining new constructs. Thus, an author preparing a manuscript on inference logic might define

```
\newcommand{\inference}[2]{\frac{#1}{#2}}
```

and write

```
\inference{x}{y}
```

and use this construct throughout the document.

Notice that defining the `\inference` as shown above and using it to encode inference statements is distinct from and more powerful than just using the T_EX builtin operator `\over` throughout the document. A commonly mentioned advantage in this context is that using the newly defined construct `\inference` will permit the author to easily change the notation used to denote *inference*. Notice, that this is in fact the same as saying that

If distinct elements in a document instance are marked up using distinct constructs, then it is possible to recognize and process these elements in a multiplicity of ways.

In A_ST_ER, the (L^A)T_EX facility of defining a second `\inference` macro that produces a different layout for *inference* can be generalized to the notion of different *audio renderings* for *inference*.

As explained above (“Document models”), A_ST_ER achieves its aural renderings by building a rich internal representation of the document content. In this representation, each document element¹ E is represented by an instance of object O_E . A_ST_ER provides a predefined type O_E for each of the builtin constructs in (L^A)T_EX. Thus, we could represent the use of `\inference` defined above in terms of object O_{over} . However, notice that this would mean losing valuable information. When building up the internal representation, the additional semantic information provided by the author’s use of the `\inference` construct is very useful. In addition, expanding all (L^A)T_EX macros results in a pure layout representation, which is not appropriate for producing aural renderings (Raman, 1992). If we were to represent instances of `\inference` in terms of O_{over} , A_ST_ER would be forced to render `\inference` the same as the `\over` construct. Though the author in this particular example may have chosen to use the same visual rendering for inferences that is normally used for fractions, the same may not carry over well to the aural domain.

¹ We use the term *element* loosely to mean a logical unit of the document.

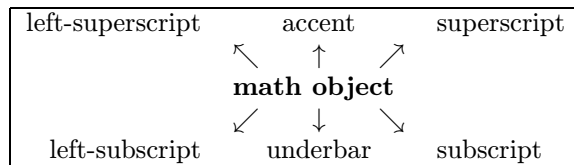


Figure 1: A math object with attributes. Each of the attributes themselves contain math objects.

Representing Extended Logical Structure

AsTeX solves the problem of representing and rendering the extended logical structure arising from user-definable macros by considering each macro definition as introducing a new object type. Instances of a macro M , are represented by instances of object O_M . Thus, in the example shown above, the definition of the construct `\inference` introduces a new object type $O_{\text{inference}}$. The $(\text{I}\mathbb{A})\text{TeX}$ macro consists of two parts; a declaration, and a series of TeX commands that the macro expands into. The macro expansion is nothing but a visual rendering rule that specifies how TeX should display instances of the object represented by the macro.

AsTeX provides an equivalent mechanism for extending the class of logical structures that are recognized. Once AsTeX has been told about a user-defined macro, audio rendering rules for the new object type introduced by this macro can be defined in AFL (Audio Formatting Language). Notice that such audio rendering rules have to be defined by the user, just as the $(\text{I}\mathbb{A})\text{TeX}$ macro is defined by hand. It is not possible in general to translate the TeX macro into a set of audio rendering rules. This is because the TeX macro is capable of performing any arbitrary computation permitted by the operators present in the TeX language (Knuth, 1984) — a Turing-complete programming language.

Rendering Information

AsTeX renders information by applying *rendering rules* to the internal representation described above (“Document models”). The system of rendering rules used in AsTeX and the language in which they are written (AFL — Audio Formatting Language) are described in detail in (Raman, 1994). In a sense, AFL is to audio formatting as Postscript is to visual formatting, although AFL is a much smaller language.

Here, we show a small example of such a rendering rule for a user-defined macro. In the following, we use CLOS generic function `read-aloud`. For the present, let us assume that function `read-aloud` executes the necessary actions to render its argument.

After extending AsTeX to process the $(\text{I}\mathbb{A})\text{TeX}$ macro `\inference` shown above (“Logical structure”), we can define

```
(defmethod read-aloud((inference inference))
  "Sample rendering for object inference."
  (read-aloud (argument 1 inference))
  (read-aloud "implies")
  (read-aloud (argument 2 inference)))
```

Given $\frac{A}{B}$, this produces “A implies B”.

If we wished to produce a rendering that inverts the order in which the arguments to macro `\inference` are rendered, we would define:

```
(defmethod read-aloud((inference inference))
  "Renders inference with arguments reversed."
  (read-aloud "We know that ")
  (read-aloud (argument 2 inference))
  (read-aloud "because")
  (read-aloud (argument 1 inference)))
```

which produces “We know B because A”.

Switching between these two rendering rules has the effect of inverting a proof-tree! Notice that writing a new rendering rule for an object O_E has the same effect as redefining the $(\text{I}\mathbb{A})\text{TeX}$ macro that corresponds to E .

AsTeX makes it easy to write several rendering rules for the same object and also allows rendering rules to be partitioned into rendering *styles*. Such *styles* can be thought of as being analogous to $\text{L}\mathbb{A}\text{TeX}$ styles, but with one important difference. Due to the non-interactive nature of traditional paper documents, a paper is typically typeset in a given style. It is not possible for the reader to change the style in which the document is typeset. Typically, we do not feel the shortcoming of not being able to change the way a mathematical expression is rendered when reading a printed paper because the eye is capable of reading the various parts of an expression in any order that is convenient. However, when listening to an aural presentation, the listener does not have this flexibility. In other words, an active reader peruses a printed paper, a passive display, whereas in the case of audio, these roles are reversed—the aural display scrolls *actively* past a passive listener.

A_ST_ER overcomes these difficulties by being a fully interactive system. It is possible for the listener to interrupt the rendering, change the rendering style in use, and listen to the document. In an interactive session with A_ST_ER, switching between rendering styles (a collection of rendering rules for different objects) and invoking individual rendering rules can be done with a few keystrokes, making it easy for a listener to obtain many different views of a document. This facility enables *active* listening.

A_ST_ER derives its power from representing document content as objects and by allowing multiple user-defined rendering rules for individual object types. These rules can cause any number of audio events (ranging from speaking a simple phrase, to playing a digitized sound). The pitch of the voice, the physical head-size of the virtual speaker, the volume, and many other parameters can be changed by rendering rules, making it easy to create sound cues to help display structure. In fact, the design of A_ST_ER does not restrict the system to producing purely aural renderings; there is nothing to preclude us from defining renderings that produce truly multimodal output; i.e., renderings where the traditional visual rendering is augmented with aural feedback. We conjecture that such multimodal renderings may prove very useful for persons with learning impairments.

To give an example of a multimodal rendering, the logo for A_ST_ER is



and is produced by (L^A)T_EX macro `\asterlogo`. After appropriately extending A_ST_ER to recognize this macro, we can define an audio rendering rule for object *asterlogo* that produces a bark when rendering instances of this macro. Thus, the same piece of markup `\asterlogo` produces the picture of Aster² when rendered visually, and an appropriate sound³ when rendered aurally.

This feature was exploited to advantage when producing the audio formatted version of the author's thesis. The dedication page of the thesis contains a large picture of Aster, and the audio formatted version⁴ contains a verbal description of the picture, accompanied by the sound of Aster panting

² Aster is my guide-dog.

³ The bark is that of a generic dog, Aster is too well trained to bark, and could not therefore be recorded.

⁴ An audio formatted version of the thesis produced by A_ST_ER (about 6 hours) is being distributed by RFB — Recordings For The Blind—as the first fully computer-generated talking book.

in the background. You can listen to this example on the WWW—visit the A_ST_ER home page by following the link to the A_ST_ER demonstration from my home page⁵ and clicking on the picture of Aster.

Several ideas come together to make all this possible. First, logical structure is of paramount importance—not its display on any one particular medium. The more a document makes structure explicit, the better the document can be displayed on (projected onto) several different mediums.

Next, the use of (L^A)T_EX macros to encode structure makes it possible to have a system like A_ST_ER, in which the internal structure can be extended to fit a document. This allows the encoding of the structure in a flexible, uniform, and consistent representation such as an attributed tree, with the addition of the quasi-prefix form for dealing with mathematics.

Finally, providing different rendering rules and styles and a flexible way to switch among them makes it possible to obtain multiple views of a document in an interactive fashion.

Conclusion

The approach used in A_ST_ER to exploit the additional semantic information present in the electronic encoding in the form of user-defined constructs points to an important feature of markup systems like (L^A)T_EX that is currently missing to a certain extent in systems like SGML. When A_ST_ER as at its inception, I firmly believed that one should use a semantic-oriented DTD to encode a document in order to be able to produce high-quality audio renderings. I still believe this; however the work on A_ST_ER does point out one shortcoming with the fixed document DTD model. Given that mathematical and technical notation is being invented all the time, a fixed DTD forces the author to encode new constructs using *only* primitives that are provided by the DTD. As a consequence, authors end up using a presentation-oriented encoding even though the DTD in use is one that is semantically oriented.

To make this concrete, consider the case of the *inference* construct described above (“Logical structure”). If the document were being encoded using a fixed non-extensible DTD that only provides a *fraction* element, the author would be forced to encode *inference* using this element.

Since in general it is not possible to define an all-encompassing DTD that covers every possible kind of math notation (those currently known and those

⁵ <http://www.research.digital.com/CRL/personal/raman/raman.html>

yet to be discovered) extensibility of the DTD as provided by (\LaTeX) is of vital importance.

Another good example of this facility in (\LaTeX) being put to good use is the Hyper \TeX system — an extension to \TeX that allows the user to view his legacy (\LaTeX) documents as online hypertext. Conceptually, we can think of `\ref` and `\label` as being object types; traditionally, these cause specific marks to appear on paper when rendered visually by \TeX ; to a system like Hyper \TeX these turn into *active* links that a user can follow interactively.

The ability to produce multiple renderings of the same object provided by ASTER was introduced in the context of aural presentations. However, such multiple presentations become equally relevant when interactively perusing online documents visually. For instance, when reading a document that presents a complex proof, a user may wish to have the same proof displayed as an outline in one window, and as a proof-tree in another (Lamport, 1993). In the case of paper documents, the user has to use her imagination to achieve such multiple views — though she is aided in this by the visual notation. In the interactive scenario presented by electronic documents, the previewer can provide some additional functionality to aid in this process.

References

- D. E. Knuth. *The \TeX book*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, Massachusetts, 1984.
- L. Lamport. “How to write a proof”. Technical Report 94, DEC Systems Research Center, Palo Alto, CA, 1993. To appear in *American Mathematical Monthly*.
- T. V. Raman. “An audio view of (\LaTeX) documents”. *TUGboat* **13**(3), 372–379, 1992.
- T. V. Raman. *Audio System for Technical Readings*. Ph.D. thesis, Cornell University, 1994.

Another Look at L^AT_EX to SGML Conversion

Sebastian Rahtz

Elsevier Science Ltd

The Boulevard

Langford Lane

Kidlington

Oxford OX5 1GB

UK

Email: s.rahtz@elsevier.co.uk

Abstract

Publishers are starting to use SGML as their permanent form of storage for documents. How can L^AT_EX files be converted to an SGML instance? This paper discusses possible strategies, and describes an implementation by Elsevier Science of a system based on conversion in T_EX itself, and extraction of SGML code from the dvi file.

The problem

The work outlined in this paper concerns the translation of L^AT_EX files into SGML. “Why,” the T_EX user asks, “do you want to do this? L^AT_EX produces nice typeset pages, it’s a worldwide convention for document interchange, and T_EX’s math markup is a standard.” But “Why,” the SGML people ask, “do you want to use L^AT_EX at all? SGML is a real, ISO, standard, with much stronger validation and portability than T_EX, and with a growing set of excellent tools.” The problem arises for us at Elsevier Science because we are caught between two camps. On the one hand, many of our authors use plain T_EX or L^AT_EX to prepare articles, and we are asked to use their files. On the other hand, the majority of our papers come to us in a wide variety of markup and word-processor formats. We must therefore have a common markup format to convert them all into. We *could* simply use each file in its native format, be it T_EX, Word or Quark Xpress, but then there are serious practical difficulties:

1. We recognise, like any serious publisher, the long-term potential of an electronic archive of material, and we want our ultimate archive to conform to the most rigorous standards. We want all our files in the same markup system, so that we can develop tools for alternative publishing media, and tools to quality-check the markup.
2. We do not, by and large, do our own typesetting, but send electronic files to a range of external firms, most of whom prove to be very reluctant to accept T_EX files. We can however achieve efficient handling of our electronic files

by the typesetters if we supply a single type of fully-tagged file which contains all the information the typesetter needs.

3. We publish over 1000 journals, and a single L^AT_EX article may arrive for any of several hundred of these; it is not practical or logical to train all our production editors to deal with L^AT_EX in a sophisticated way, and it can frustrate the process if we have to push occasional articles onto L^AT_EX-experienced staff.
4. On a purely day-to-day basis, we cannot maintain all the possible hardware and software configurations at all production sites, or even keep all our production editors trained to use the many different systems and platforms.

The only possible solutions are therefore either to adopt an *ad hoc*, temporary, workaround like doing all our work in Word or Quark, or to invest in a total conversion to SGML, flexible enough to meet all our known and perceived future needs. Elsevier Science made their decision to adopt SGML some years ago, has developed its own DTDs, and is now engaged in the mammoth task of converting its journal production environment to a system which produces totally electronic files for typesetting or electronic publication. The standards and practices adopted are described in [3].

Consequently, to achieve a complete rigorous quality-controlled archive, L^AT_EX files have to be converted to SGML. But SGML is merely a language for describing markup, and ‘conversion’ could mean little more than syntactic conversion, whereby all L^AT_EX macros were changed to a corresponding SGML tag; thus `\section` is changed to `<section>`.

This is, in effect, the approach adopted by some of the apparent $\LaTeX \Leftrightarrow$ SGML systems — constraining the DTD to mimic \LaTeX . What it does not do is solve the problem of parsing the general \TeX syntax.

Another approach for the particular case of mathematics is to declare a single SGML element `<texmath>`, and leave the math markup exactly as is. This is attractive, but has two drawbacks:

1. Many authors have dependencies in their markup on their own definitions, or external style files; it is not trivial to totally ‘flatten’ all these references, but if they are left in, it is irritating to have to work around it when the document is used later (possibly *much* later, when the style files may have changed...). An even worse case is when the author *changes* definitions half-way through the paper, so that `\MyX` expands to `X_x~y` for the first few equations, but to `X_a~b` for the rest — quite legal, however inelegant.
2. If our archive is to still be useful in 20 years, it needs to be totally internally consistent and complete. The `<texmath>` notation assumes the existence in 20 years of a program able to parse it in the same way as \TeX does now — likely, but not under our control.

We are left with having to genuinely translate arbitrarily-complicated \TeX code into ‘real’ SGML; i.e., our DTD may bear no relation to the logical structure of \LaTeX , and we have to deal properly with mathematics and tables. The remainder of this paper discusses possible solutions, and describes the one we have adopted.

Possible solutions

There are four technologies we can adopt:

1. Throw away the electronic file, and retype the entire text from the printed copy direct into SGML.
2. Strip out all \TeX coding, and treat it like an unknown word-processor — leave all the words intact, but re-key all the markup, maths and tables.

This approach and the first are not as ridiculous as they sound to \TeX experts; they may well be the cheapest, quickest, most reliable solution for a small quantity of material. The startup cost is zero, and the unit cost may be high — but this may be preferable to a large development cost for a system which is not heavily used and needs maintenance. The reasons *not* to adopt it include the probable gain in processing time of an automatic conversion,

author satisfaction at decreased proof-reading, and the potential high cost of this approach for complex mathematics.

3. Write a new program to parse the \LaTeX , and write SGML. The advantages are:

- Writing parsers is a well understood process, although the ‘traditional’ *lex* and *yacc* tools are necessarily not suited to macro languages like \TeX ;
- We get an efficient, portable program which (if written well) can be maintained.
- Nothing gets into the output that we do not understand.

but there are the following disadvantages:

- Parsing \TeX is notoriously hard, because its interface is a macro programming language, and thus the syntax is extensible.
- There are no fully-successful implementations known, although there are many *partial* relevant solutions (such as Julian Smart’s *tex2rtf*).
- We would have to assume ‘normal’ \LaTeX , and hand-fix any strange files.

This approach could involve a two-stage program, the first attempting (a more or less impossible aim) to regularise the \TeX by expanding macros, and the second doing the parsing and translating. Aspects of this are encapsulated in the Perl script for `latex2html` (discussed in useful detail in [1]), but the philosophy of that system is to convert anything it cannot understand into a picture (via \TeX and PostScript to GIF), which means that it does not really try to be a robust \TeX parser.

4. Work with \TeX itself, to take advantage of the entire macro-processing language and input parser, but fix the output to produce SGML instead. Joachim Schrod [4] has exposed the myth that \TeX cannot be recoded in a more open system, and his implementation would be a good starting point for a \TeX -like program. We can also, however, simply write a special purpose \TeX format which defines all the interesting \TeX and \LaTeX constructs as SGML commands, and extracts the information from the `dvi` file after processing. This is the basis of the approach we are considering using. It has some considerable advantages:

- The task of parsing the \TeX language (expanding the macros) comes ‘for free’;
- The conversion from \LaTeX construct to SGML construct is done in the same language as the original definition; it can also

be customised by existing L^AT_EX-trained staff;

- The system is portable, easily modularised, and can cope with any form of T_EX (with the appropriate macro programming);

while on the downside

- The output is not regularised, since only some items (those foreseen) are trapped; everything else passes through T_EX's processes, which can be both good and bad;
- Writing sophisticated programs in T_EX is not easy, or well-understood, constraining future contract programmers to be T_EX-qualified;
- Some things may not be possible at the macro redefinition stage, notably in maths;
- Extracting material from the dvi file is far from easy.

The third approach has been tried and tested by Bart Wage (Elsevier Amsterdam) for a very large number of relatively simple documents (the front matter of articles), but the program was becoming unwieldy when trying to extend it to the full range of papers we publish. This then was the impetus for the system described in this paper. The approach of rewriting T_EX innards has been largely completed by ICPC (Dublin)¹ with a reworking of the basic WEB code, and this is certainly the most reliable plan of attack; however, it still requires extensive macro programming to get the most from the system, and so the majority of our work is unaffected.

A practical solution

Given the advantages and disadvantages outlined above, we decided to experiment with a T_EX-based approach; we knew that we had the existing traditional parsing program to fall back on, and we knew that ICPC were working on a radical solution altering T_EX. The main reason for wanting T_EX to do the parsing was that our author L^AT_EX markup is extremely variable, and essentially beyond our control; maximum flexibility and ease of *ad hoc* quick fixes was therefore important. It must be stressed that this is a solution tailored to a particular setup; it is not necessarily the best universal solution!

The general outline of our system is shown in Fig. 1. The steps, which are discussed in detail in the next section, are:

1. Roughly clean up the L^AT_EX, and impose some standard markup;
2. attach the `lat2cap` package, and run L^AT_EX (twice, for cross-referencing);
3. extract the SGML text from the DVI file;
4. clean up the SGML and parse it against an intermediate DTD;
5. transform to the final Elsevier article DTD, and parse again.

The justification for the extra stage of intermediate DTD is very dependent on the relationship between the L^AT_EX markup in use, and the DTD. In our case, we found that although we could match up most structures directly, some of the L^AT_EX was not amenable to a one-pass program.

Details of the system

Cleaning the L^AT_EX Since our approach is to re-define T_EX macros to produce SGML tags, we could in theory cope with any dialect of T_EX; in practice, however, our articles are structured in a clear, but complicated way, and it is preferable to get the information right before we look at converting it. We have maintained a set of L^AT_EX style files for some years for internal typesetting of many journals, which are used in conjunction with a single public preprint style, which implements all the constructs in a typographically simple way. The complexity is largely in the front matter, as shown in Figs. 2 and 3, where we present the input markup, and one style of typeset output. Almost all author files need some work in this area, so it makes sense to concentrate the translation *assuming* the editing has been done.

A more important consideration is the *richness* of the L^AT_EX. In theory, a valid L^AT_EX file:

```
\documentclass{article}
\begin{document}
  ABOUT CATS

  by

  Seroster

  once upon a time...
\end{document}
```

could be correctly translated to

```
<article>ABOUT CATS<p><p>by<p><p>
Seroster<p>once upon a time...
</article>
```

but this is hardly useful. For an article to have some value in an electronic warehouse, we must minimally identify in an unambiguous way the following elements:

¹ Contact Seamus McCague (seamus@icpc.ie) for details.

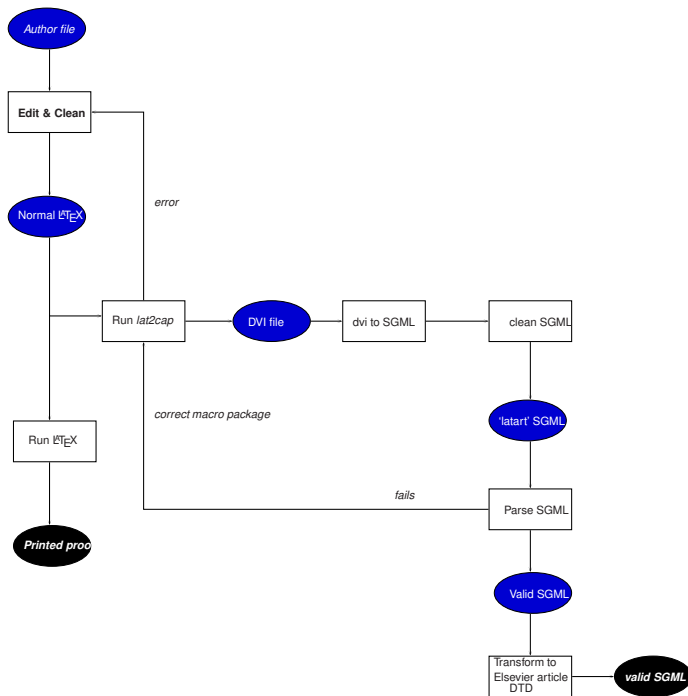


Figure 1: Processing scheme for LaTeX to SGML

1. the authors and their addresses;
2. the title, abstract and keywords;
3. section headings;
4. bibliographical references and citations;
5. figure references.

We may find it useful, in fact, to distinguish 4 ‘types’ of LaTeX file:

Bad It passes through LaTeX, but very low-level visual markup is used; this file says `Figure {\cmbxten 66}`

Clean Composed according to the commands in the LaTeX manual, but without utilizing the abstract features like cross-referencing, floating figures etc.; this file says `Figure~\textbf{66}`;

Rich Uses *all* the commands of the LaTeX manual, at the highest level possible for the context; this file says `\figname~\ref{fig66}`

Over-ripe This file has no spelling or grammatical mistakes and has not used any visual markup; it says `\FIG{66}`, with a single definition in a separate macro package allowing the caption to be placed above or below the figure.

It is clear that only ‘rich’ LaTeX or better can be translated to useful SGML; the alternative is to do all the cleanup at the SGML stage, but this has the disadvantage that we have no macro language; a 100 page LaTeX article with generic, but wrong, markup

may require a single change, the SGML output needs hundreds. This is discussed further below (“Where in the work-flow?”).

The macro package The bulk of our work consists in redefining every possible LaTeX command, at as high a level as possible. We present below selections from the macro package, showing those which are less than obvious.

Basic tools In order to simplify the TeX output, our first aim is to constrain and nullify the vast majority of low-level visual markup code. We set everything in LaTeX 2_ε’s T1 encoding, in a single font which is completely monospaced and allows no hyphenation. The use of T1 means that LaTeX will automatically translate all accented and non-ASCII characters, which may be entered with TeX macros, into 8-bit characters in the output. Our dvi processor will translate these into the appropriate SGML entities or accent tags.

```

1  \def\baselinestretch{1}
2  \DeclareFontShape{T1}{sgml}{m}{n}{<-> sgml}{
3  \DeclareFontShape{OT1}{sgml}{m}{n}{<-> sgml}{
4  \fontencoding{T1}\let\fontencoding=@gobble
5  \fontfamily{sgml}\let\fontfamily=@gobble
6  \fontseries{m}\let\fontseries=@gobble
7  \fontshape{n}\let\fontshape=@gobble
8  \fontsize{10}{10pt}\let\fontsize=@gobbletwo
9  \global\let\mathversion@gobble
10 \global\let\getanddefine@fonts@gobbletwo

```


```

\begin{document}
\volume{33} \issue{4} \ssdi{93}{E0045F}
\accepted{4 Kal 44} \received{2 Kal 44}
\firstpage{101} \runauthor{Cicero, Caesar and Vergil}
\runtitle{In Catilinam}
\begin{frontmatter}
\title{In Catilinam IV: A murder in 5 acts\thanksref{X}}
\author[Paestum]{Marcus Tullius Cicero\thanksref{Senate}}
\author[Rome]{Julius Caesar}
\author[Baiae]{Publius Maro Vergilius}
\thanks[X]{This is the history of the paper, etc etc}
\address[Paestum]{Buckingham Palace, Paestum}
\address[Baiae]{The White House, Baiae}
\address[Rome]{Senate House, Rome}
\thanks[Senate]{Partially supported by the Roman Senate}
\begin{abstract}
Cum M. Cicero consul Nonis Decembribus senatum in aede
Iovis Statoris consuleret, quid de iis coniurationis Catilinae
sociis fieri placeret, qui in custodiam traditi essent, factum
est, ut duae potissimum sententiae proponerentur, una D. Silani consulis
designati, qui morte multandos illos censebat,
altera C. Caesaris, qui illos publicatis bonis per municipia
Italiae distribueudos ac vinculis sempiternis tenendos existimabat. Cum
autem plures senatores ad C. Caesaris quam ad
D. Silani sententiam inclinare viderentur, M. Cicero ea, quae
infra legitur, oratione Silani sententiam commendare studuit.
\end{abstract}

\begin{keyword}
Cicero; Catiline; Orations
\end{keyword}
\end{frontmatter}

```

Figure 2: Example Elsevier front-matter markup



ELSEVIER

0 0 0 0 - 0 0 0 0 (0 0) X X X X X - X X

A Journal Of Research 33 (1995) 101–105
 © Elsevier Science Limited
 Printed in Great Britain. All rights reserved
 000-0000/95/\$9.50

In Catilinam IV: A murder in 5 acts¹

Marcus Tullius Cicero^{*,2} Julius Caesar[‡] Catullus[‡] Publius Maro Vergilius[†]

^{*} *Buckingham Palace, Paestum*
[†] *The White House, Baiae*
[‡] *Senate House, Rome*

(Received 2 Kal 44; accepted 4 Kal 44)

Cum M. Cicero consul Nonis Decembribus senatum in aede Iovis Statoris consuleret, quid de iis coniurationis Catilinae sociis fieri placeret, qui in custodiam traditi essent, factum est, ut duae potissimum sententiae proponerentur, una D. Silani consulis designati, qui morte multandos illos censebat

Altera C. Caesaris, qui illos publicatis bonis per municipia Italiae distribueudos ac vinculis sempiternis tenendos existimabat. Cum autem plures senatores ad C. Caesaris quam ad D. Silani sententiam inclinare viderentur, M. Cicero ea, quae infra legitur, oratione Silani sententiam commendare studuit.

Key words: Cicero; Catiline; Orations

Figure 3: Elsevier front-matter typeset

```
11 \selectfont
12 \def\selectfont{}
```

Within our SGML DTD there is no requirement for typesize markup, so for authors who use size-changing commands we simply nullify them:

```
1 \let\normalsize\relax
2 \let\tiny\relax
3 \let\scriptsize\relax
4 \let\footnotesize\relax
5 \let\small\relax
6 \let\large\relax
7 \let\Large\relax
8 \let\LARGE\relax
9 \let\huge\relax
10 \let\Huge\relax
```

Writing SGML tags `\SGML` is a special command to emit SGML tags; it is important to do this, rather than use the actual `<`, `>` and `&` characters, as those will be translated to entities by the later processing. Note that our dvi processor will emit the parameter of the `\special` command as literal text.

```
1 \def\ENT#1{\leavevmode\special{&#1};}
2 \long\def\SGML#1{\leavevmode
3 \special{<#1\special{>}}
```

Headings Headings are treated differently to \LaTeX in the Elsevier DTD; there is only one heading tag, but nesting is used to indicate hierarchy. We need to trace the section depth and close open levels at the right time. The actual *title* of the section is not an attribute or context of the tag, but comes in a separate section title tag. We therefore define a single macro which takes each sectioning command, with a level, and keeps track of what level we are at, ending and starting heading elements as needed.

```
1 \def\@dblst#1#2{\@Section*{#1}{#2}}%
2 \def\@Section{#1}{#2}}
3 ...
4 \def\section{\expandafter\secdef
5 \@dblst{2}{section}}
6 ...
7 \def\@Section#1#2#3{%
8 \ifnextchar[%
9 {\@Section{#1}{#2}{#3}}%
10 {\@Section{#1}{#2}{#3} []}%
11 }
12 \newcount\current@sectionlevel
13 \current@sectionlevel=99
14 \def\@Section#1#2#3[#4]#5{%
15 \ifnum\current@sectionlevel=99\else
16 \loop
17 \ifnum\current@sectionlevel>#2
18 \typeout{Section level #2 inside
19 \the\current@sectionlevel}%
20 \advance\current@sectionlevel by -1
21 \SGML{/sec}%
22 \repeat
```

```
23 \ifnum\current@sectionlevel=#2
24 \SGML{/sec}
25 \fi
26 \fi
27 \global\current@sectionlevel#2
```

As we will see later, the cross-referencing mechanism is altered to use unique tags for all sections, not just those labelled with `\label`.

```
1 \refstepcounter[secr]{#3}%
2 \SGML{sec id=#3.\@currentSlabel}%
3 \SGML{st}#5\SGML{/st}%
4 \SGML{p}%
5 }
```

List environments These can be dealt with straightforwardly, as the match between the DTD and \LaTeX is almost 100%.

```
1 \def\item{%
2 \ifnextchar [ {\@item}{\SGML{li}}%
3 }
4 \def\@item[#1]{\SGML{li id=#1}}
5 \renewenvironment{enumerate}
6 {\SGML{l type=ord}}{\SGML{/l}}
7 \renewenvironment{itemize}
8 {\SGML{l type=unord}}{\SGML{/l}}
9 \renewenvironment{description}
10 {\SGML{l type=def}}{\SGML{/l}}
11 \def\quote{\SGML{qd}}
12 \def\endquote{\SGML{/qd}}
13 \let\quotation\quote
14 \let\endquotation\endquote
15 \def\{\{\SGML{p}}
```

Font changes The Elsevier DTD supports a wide range of typeface changes in the same way as \LaTeX does:

```
1 \def\it{\SGML{it}\aftergroup\ENDTAG}
2 \def\bf{\SGML{b}\aftergroup\ENDTAG}
3 \def\s1{\SGML{it}\aftergroup\ENDTAG}
4 .....
5 \def\ENDTAG{\SGML{/}}
6 \let\/=\relax
7 \def\textrm#1{\SGML{rm}#1\SGML{/}}
8 \def\textsf#1{\SGML{ssf}#1\SGML{/}}
9 \def\texttt#1{\SGML{ty}#1\SGML{/}}
10 .....
11 \def\emph#1{\SGML{it}#1\SGML{/}}
```

Bibliographic citations

```
1 \def\citex[#1]#2{%
2 \SGML{bbr id="bib-#2"}#1%
3 }
4 \def\check@bb{%
5 \ifin@bb\SGML{/bb}\in@bbfalse\fi
6 }
7 \def\@lbibitem[#1]#2{%
8 \check@bb
9 \SGML{bb id="bib-#2"}%
10 \global\in@bbtrue
```

```

11 }
12 \def\@bibitem#1{%
13   \check@bb
14   \SGML{bb id="bib-#1"}%
15   \global\@in@bbtrue
16 }
17 \def\thebibliography#1{%
18   \SGML{bm}\SGML{bibl}%
19   \let\\relax
20 }
21 \def\endthebibliography{%
22   \check@bb
23   \SGML{/bibl}%
24 }

```

Cross-referencing Anything which can be referenced advances some counter; we overload this to put in an SGML id, and make a note of that for later use in `\label`. An extra parameter is written to the `.aux` file, adding an identifier to the literal page number and section number. This will fail badly if `\theS<name>` does not expand to a sensible reference. This means that classes or package which introduce new elements need to define an equivalent `\theS<name>` for every `\the<name>`

These shenanigans are to make sure section numbers, etc., are always arabic, separated by dots. Who knows how people will set up `\@current` label? If they put spaces in (quite legal) then the processor will get upset.

```

1  \def\pageref#1{\SGMLWarning{pageref}}
2  \let\vref\ref
3  \long\def\footnote#1{%
4    \refstepcounter{fnr}{Sfootnote}%
5    \SGML{fn id=Sfootnote.\@currentSlabel}%
6      #1\SGML{/fn}%
7  }
8  .....
9
10 \def\@setref#1#2#3{%
11   \ifx#1\relax
12     \protect\G@refundefinedtrue
13     ??
14     \@latexwarning{Reference '#3' on page
15       \thepage \space undefined}%
16   \else
17     \special{<}%
18     \expandafter\@secondoftwo#1{%
19       \expandafter#2#1\null}%
20     \special{>}%
21     \fi

```

But all this is very flaky, and open to abuse. Styles like `subeqn` will mess it up, for starters. Appendices are an issue, too. We just hope to cover most situations. We can at least cope with the standard sectioning structure, allowing for `\part` and `\chapter`.

```

1  \@ifundefined{thepart}{\%
2    \newcommand\theSpart{\arabic{part}}}%
3  \@ifundefined{thechapter}{\%
4    \newcommand\theSsection{\arabic{section}}%
5    \newcommand\theSfigure {\arabic{figure}}%
6    \newcommand\theStable {\arabic{table}}%
7  }{\%
8    \@ifundefined{thepart}%
9      {\newcommand\theSchapter
10       {\arabic{part}.\arabic{chapter}}}%
11     {\newcommand\theSchapter
12      {\arabic{chapter}}}%
13  \newcommand\theSfigure
14    {\theSchapter.\arabic{figure}}%
15  \newcommand\theStable
16    {\theSchapter.\arabic{table}}%
17  \newcommand\theSsection
18    {\theSchapter.\arabic{section}}%
19
20  }
21  ...
22  \newcommand\theSequation
23    {\theSsection.\arabic{equation}}%
24  \newcommand\theStheorem
25    {\theSsection.\arabic{theorem}}%
26  \newcommand\theSthm
27    {\theSsection.\arabic{thm}}%
28  \newcommand\theSenumi
29    {\theSsection.\arabic{enumi}}%
30  ...
31  \newcommand\theSSfootnote
32    {\arabic{Sfootnote}}%
33  ...
34  \let\theSHmpfootnote\theSSfootnote
35  \let\S@refstepcounter\refstepcounter
36  \def\refstepcounter{\@ifnextchar[%
37    {\@refstepcounter}%
38    {\@refstepcounter[]}}%
39  \def\@refstepcounter[#1]#2{%
40    \ifx\#1\\\edef\@sgmlname{#2}%
41    \else
42      \def\@sgmlname{#1}
43    \fi
44    \S@refstepcounter{#2}%
45    \S@makecurrent{\@sgmlname}{#2}%
46  }
47  \def\S@makecurrent#1#2{%
48    \edef\@currentSlabel{%
49      \csname theS#2\endcsname}%
50    \global\edef\@currentSref{%
51      #1 id="#2.\expandafter\strip@prefix
52        \meaning\@currentSlabel"%}
53  }
54  \def\label#1{%
55    \@bsphack
56    \protected@write\@auxout{%
57      {\string\newlabel{#1}{%
58        \@currentSref}}}%

```

```
59 \@esphack
60 }
```

Mathematics and tables Surprisingly, the majority of mathematics requires fairly trivial redefinition; powerful and complicated as TeX's math typesetting is, we simply ignore it.

```
1 \catcode'\^=13 % circumflex for superscript
2 \catcode'\_ =13 % underline for subscript
3 \def\frac#1#2{\SGML{fr}\SGML{nu}#1
4 \SGML{de}#2\SGML{/fr}}
5 \def _#1{\SGML{inf}#1\SGML{/inf}}
6 \def ^#1{\SGML{sup}#1\SGML{/sup}}
```

The real implementation of $\hat{\ }$ in our package is actually rather more complicated than presented here, because it has to cope with markup like this:

```
x^\frac{a}{b}
```

which is in fact only barely tolerated in L^AT_EX. In this situation, the parameter to $\hat{\ }$ ends up as `\frac` unless we take special precautions.² Many macros just output the right SGML entity.

```
1 \def\alpha{\ENT{alpha}}
2 \def\beta{\ENT{beta}}
3 .....
4 \def\vartriangleleft{\ENT{vartriangleleft}}
5 \def\vartriangleright{\ENT{vartriangleright}}
6 \def\vartriangle{\ENT{vartriangle}}
7 \def\veebar{\ENT{veebar}}
8 \def\left#1{\SGML{fen}\SGML{cp style="#1"}}
9 \def\right#1{\SGML{/fen}}
```

The remainder of the mathematical work, including equation labelling and numbering, and tables, involves quite straightforward macro programming, albeit somewhat byzantine at times.

TeX accentuation is handled by a similar scheme in our DTD.

```
1 \def\acute#1{\SGML{a}\SGML{ac}#1
2 \SGML{ac}\ENT{acute}\SGML{/a}}
3 \def\grave#1{\SGML{a}\SGML{ac}#1
4 \SGML{ac}\ENT{grave}\SGML{/a}}
5 ...
```

DVI to ASCII We have now written a typeset page of text in a monospace font interspersed with `\special` commands relating to `<` and `>` characters; we are not at all interested in the layout, we just want the words, and all vertical and horizontal spacing turned to simple spaces. There have been a variety of 'dvi2tty' programs written over the years, but most of them are aiming to produce crude ASCII layout; after some experimentation, Geoffrey Tobin's excellent `dv2dt` program was found. This, and a

² We owe the solution to Alan Jeffrey, David Carlisle, Chris Rowley and Michael Downes, almost simultaneously. Seldom can such a concentration of L^AT_EX brain-power have been used to crack such a small nut.

companion `dt2dv`, provides a reliable, and easy to understand, text representation of a dvi file which can even be edited and turned back to dvi. Working with an ASCII representation means that it is easy to check and debug one's work, and writing a parser in Flex is simple. A flavour of the 'dt' language can be gleaned from this example; text is in round brackets, and the `w` commands are horizontal spacing.

```
special1 1 '<'
(fn)
(id=Sfootnote.1)
special1 1 '>'
(Everyone)
w3 218453
(likes)
w0
(cat)
w0
(meat)
special1 1 '<'
(/fn)
special1 1 '>'
w0
special1 1 '<'
(li)
special1 1 '>'
(dogs)
w0
special1 1 '<'
(/1)
special1 1 '>'
```

Our parser needs to read this, output the words and `\special` commands, insert spaces, and convert any non-ASCII characters to SGML entities. Readers familiar with Flex will see from the following fragment how trivial this is:

```
<SKIP>"\"( BEGIN TEXT ;
<SKIP>"special1 "[0-9]+" '>'
{ BEGIN SPECIAL ; } ;
<SPECIAL>"['']*" { printtext(yytext); };
<SPECIAL>"'" BEGIN SKIP ;
<TEXT>"\"( BEGIN SKIP ;
....
<SKIP>"s1 249"
printtext("<a<ac>u<ac>&grave;</a>") ;
....
<SKIP>^z { printtext("<P>"); };
<SKIP>^[rwy] { printtext(" "); };
<SKIP>. |
<SKIP>\n { } ;
```

Cleaning the SGML Unfortunately, L^AT_EX has a strange way with paragraphs (as indeed do SGML DTDs), and understanding the vertical and horizontal spacing in a dvi file is slightly fraught; therefore we run a simple cleanup to remove extraneous

<p>s and empty tags, and add some miscellaneous markup.

Transforming the SGML Now to the crux of any L^AT_EX to SGML program — what is the target DTD? Do we go for a translation all the way, or adopt the strategy of the ‘Rainbow’ tools, and work to an intermediate DTD from which we can transform to the desired result? It is important to distinguish between transforming, and upgrading; if we write ‘poor’ SGML, we cannot reliably upgrade it to ‘rich’ SGML, whereas ‘rich’ SGML in dialect A can certainly be transformed to dialect B with no problem. This is one of the tasks for which DSSSL will be useful in the longer term, but for this purpose we use a public domain Perl5 module for manipulating SGML instances. To ensure that the output is correct, the file is parsed using James Clark’s *nsgmls*, and the standard ESIS output from that program is then read by the transformer, writing the final output. This is of course again parsed against the real DTD before the program run is deemed successful.

A simple example may suffice to show why this last transformation stage is necessary. A typical piece of L^AT_EX often contains `eqnarray` structures, or multi-line equations. In L^AT_EX, lines end with a `\\`, and only then do we find if they are numbered. In the Elsevier DTD, this would be represented as a series of math displays nested inside an outer display. No doubt one *could* program this in L^AT_EX, but it is simpler to convert the end of line `\\` to a special tag, and transform the result later.

A harder problem, which we do not in fact face (since we know it can be fixed in the editing phase), is to deal with

```
1 {A \over B} 2
```

with T_EX programming, since in the simple redefinition of macros we do not realise we are in a ‘fraction’ until the group has started, and we have no way of back-tracking to put in a tag at the start of the construct. By contrast, the L^AT_EX notation

```
1 \frac{A}{B} 2
```

is easy to transform immediately to

```
1<fr><nu>A<de>B</fr>2
```

Where in the work-flow?

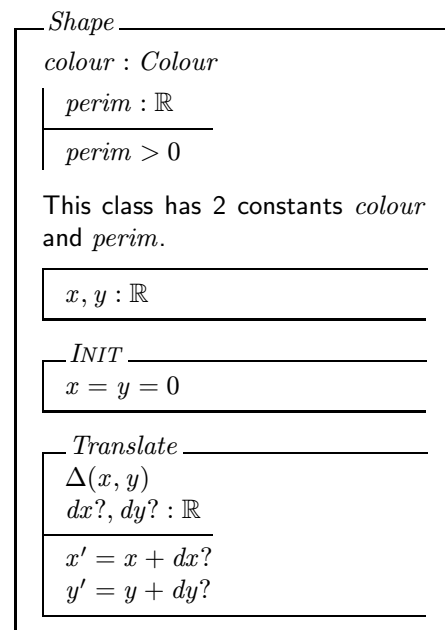
Once the technical problem is solved, we still have to consider where in the work-flow it is used. These are some of the issues we have to face in a production environment:

1. Who runs the converter? Disk administrator, desk editor or special L^AT_EX editor? Do we send it all to an outside contractor?

2. What if it goes wrong? Are problems fixed in L^AT_EX or SGML? Does the SGML editing tool permit illegal SGML to be imported?
3. Is copy-editing done in SGML or L^AT_EX? Some regular markup by the author may well be far easier to change once in L^AT_EX than the expanded form in SGML.

There are also some miscellaneous issues to consider:

1. What about author graphics created in T_EX? These can be very sophisticated. The best solution would be to run the article through L^AT_EX and extract the pages as PostScript output, but this requires some confidence with, and knowledge of, T_EX.
2. What do we do with L^AT_EX constructs which have no corollary in the Elsevier DTD? Two obvious examples are chemical structures, and the the formal Z schemas used in computer science, like this:



Conclusions

There remain, of course, three questions:

1. Does this approach work, other than as a toy?
2. Are we actually using it?
3. Where can it be found?

In our experience, no T_EX-related project works 100% reliably without manual intervention; this one is no exception. The approach works, but it is likely that a sophisticated T_EX programmer could quickly produce a file which produced inappropriate output. However, we do claim that it succeeds in its aim,

and that we have a usable, maintainable, and flexible tool.

To enable the generation of SGML from L^AT_EX, we are now seriously testing this tool, and using it in pilot projects; it remains to be seen whether or not it will move into full production, or whether the maintenance implications will be deemed too inefficient for the potential volume of material.

The approach outlined in this paper relies almost entirely on T_EX programming, and the principles followed are all demonstrated above. The complete style file has *not* been placed in the public domain, because it represents a not inconsiderable (ongoing) investment by Elsevier Science, and because it is not a general-purpose tool to translate L^AT_EX to arbitrary DTDs. However, those interested in discussing this approach are invited to contact Sebastian Rahtz directly. T_EX wizards will learn that many problems are still to be solved, or dealt with in a considerably more elegant way.

Acknowledgements

The genesis of this project was work done by Michel Goossens at CERN in 1993 to convert L^AT_EX documents to HTML, using the ‘dvi to ASCII’ program of Alexander Samarin and Basil Malyshev. At Elsevier, detailed discussion with Bart Wage turned the idea into reality, and he wrote the second half of the system. The L^AT_EX package shares material with the author’s *hyperref* package (see [2]). Peter Flynn and Jonathan Fine, in their different ways, have made contributions to the T_EX and SGML relationship which impacted on this work. Joachim Schrod read this paper and gave extremely helpful feedback on it, and the subject in general. Later versions of the package will hopefully take advantage of his many good suggestions.

References

- [1] Michel Goossens and Janne Saarela, “From L^AT_EX to HTML and Back”, *TUGboat*, 16(3), 1995.
- [2] Yannis Haralambous and Sebastian Rahtz, ‘L^AT_EX, hypertext and PDF, or the entry of T_EX into the world of hypertext’, *TUGboat*, 16(2), 1995.
- [3] Martin Key, ‘Theory into Practice: working with SGML, PDF and L^AT_EX at Elsevier Science’, *Baskerville* 5 (2), 1995.
- [4] Joachim Schrod ‘Towards interactivity for T_EX’, *TUGboat* 15(3), 1994 (Proceedings of the 1994 T_EX Users Group Annual meeting), 309–317.

Omega — why Bother with Unicode?

Robin Fairbairns

University of Cambridge Computer Laboratory

Pembroke Street

Cambridge CB2 3QG

UK

Email: `rf@c1.cam.ac.uk`

Abstract

Yannis Haralambous' and John Plaice's Omega system employs Unicode as its coding system. This short note (which previously appeared in the UKTUG magazine *Baskerville* volume 5, number 3) considers the rationale behind Unicode itself and behind its adoption for Omega.

Introduction

As almost all T_EX users who 'listen to the networks' at all will know, the Francophone T_EX users' group, GUTenberg, arranged a meeting in March at CERN (Geneva) to 'launch' Ω. The UKTUG responded to GUTenberg's plea for support to enable T_EX users from impoverished countries to attend, by making the first disbursement from the UKTUG's newly-established Cathy Booth fund. The meeting was well attended, with representatives from both Eastern and Western Europe (including me; I also attended with UKTUG money), and one representative from Australia (though he is presently resident in Europe, too).

The speakers at the meeting were Michel Goossens (the president of GUTenberg¹), as host for GUTenberg and as an expert on background to, and the use of Unicode), and Yannis Haralambous and John Plaice, Ω's two developers.

The meeting can be accounted a success; all that attended enjoyed themselves, and also learnt a lot.

This paper is a minor revision of an article I wrote for the UKTUG magazine *Baskerville*, volume 5, number 3, and outlines some of my views of (and support for) the choices that Haralambous and Plaice have made. I will consider the arguments for using Unicode as a foundation for future text processing, in particular (of course) T_EX-related processing.

What is Ω?

Ω (Haralambous and Plaice, 1994) is an extension of T_EX and related programs that has been designed

¹ And now (at the time of writing) president-elect of TUG itself.

and written by Yannis Haralambous (Lille) and John Plaice (Université Laval, Montréal). It follows on quite naturally from Yannis' work on exotic languages (see, among many examples, Haralambous, 1990; 1991; 1993; 1994), which have always seemed to me to be bedevilled by problems of text encoding.

Simply, Ω (the program) is able to read scripts that are encoded in Unicode (or in some other code that is readily transformable to Unicode), and then to process them in the same way that T_EX does. Parallel work has defined formats for fonts and other necessary files to deal with the demands arising from Unicode input, and upgraded versions of METAFONT, the virtual font utilities, and so on, have been written. Ω itself is based on the normal Web2C distribution that is at the base of most modern Unix implementations, and of at least one of the PC versions that is freely available.

Why Unicode?

There are something between 3000 and 6000 languages in use in the world, for which a writing system exists. (The set of languages is shrinking all the time as the deadening effect of cultural intrusion, primarily through the electronic media, overwhelms the desire to support existing cultures to the extent of teaching their language to the young.) The distribution of languages is by no means even throughout the globe (Michel showed us a map), and there are many that have not been and will presumably now never be formally recorded.

When we come to writing systems, we find almost every variation imaginable in use somewhere in the world. The Latin-like system (written left to right with modest numbers of diacritics simply arranged) has very wide penetration, not least because so many languages were first written down by

Western European missionaries or other explorers. Languages such as Vietnamese are classified as ‘complex Latin-like’, with ≥ 2 diacritics per character; an artificial example of the same effect is IPA (the International Phonetic Alphabet) which has sub- and super-scripts and joining marks. Languages such as Hebrew and Arabic are written right to left, and constitute another class. Then there are the multiple-ligature writing systems typified by the Indic languages such as Devanagari (of which we had a fascinating exposition at the 1993 UKTUG Easter meeting on ‘non-American’ languages, from Dominik Wujastyk), and finally the syllabic scripts (such as Korean Hangul and Japanese Hiragana and Katakana), and the ideographic scripts (Chinese and Japanese Kanji).

Encodings are needed for computer operations on language of any sort. There are differences between the coded representation and the written (or printed) representation. Everyone who’s read about T_EX at all will know about ligatures (the CM fonts, and most PostScript fonts, implement ligatures so that, for example, ‘f1’ typed appears as ‘fl’ printed). More significantly, almost all adults in Western cultures write ‘joined-up’, which is in itself application of a form of ligature. All these ligatures are for presentation, not for information, and so it is unreasonable for them to be represented in a character set. Other ligatures, however, form real characters in some languages (examples are æ in Danish and Norwegian, and œ in French).

Each encoding represents a ‘character set’ that is to be used by the computer; the history of how these character sets have developed is long and sorry indeed. In the ‘dark ages’ (in fact, as recently as the early 1960s, when I started computing), every make of computer system had its own character code, many of them based on the 5-bit teleprinter codes used in telex printers. Eventually, the rather more sophisticated teletypes appeared, which used seven bits of an eight-bit code; this 7-bit codification was standardised as ASCII (the American Standard Code for Information Interchange), which was (in the area of application it was designed for) an excellent code. It had all the properties needed for many of the significant development of computers in the 1960s, but it had one serious flaw: it was not able to encode diacritics, which are used in almost every language (but which your all-American information interchanger would seldom have a need for).

To regularise the resulting mess, ISO adopted the ASCII standard as the basis for an international 7-bit character set, ISO 646. ISO 646 is identical to ASCII in the code points that it specifies; however,

some of the characters that ASCII does specify are left “for national variation” in ISO 646; ASCII itself then became the USA national variation of ISO 646. An example of national variation is defined for the UK, which specifies that the code point that holds ‘#’ in ASCII should hold a pounds sign (£). There are versions for various Nordic languages that include characters such as æ or å in place of braces, a version for French with acute, grave and circumflex-accented letters, one for German that offers letters with umlauts and ‘sharp s’ (ß).

There were various attempts at mechanisms to assign different character sets for use by those who need to use characters from several different sets (for example, someone writing a Swedish-English Dictionary); an example is ISO 2022, which defines escape sequences for such switches. These efforts proved impractical (at least they seemed so to me), and 8-bit developments of ISO 646 arose, with the ability (comfortably) to express more than one language.

Thus were born the ISO 8859 character sets. The commonest of these (at least in the ken of most English speakers) is ISO Latin-1 (ISO 8859-1, that is, part one of the multi-part standard), which was designed for use by Western Europeans. As well as the ‘basic ASCII set’ in the first 128 characters, it has diphthongs and vowels appropriate to most Western European languages. Oddly, it omits the œ diphthong that French uses, and (perhaps less surprisingly²) it omits some of the accent forms used by Welsh. ISO 8859 didn’t stop with part 1, though; there are variants that accommodate Cyrillic (for Russian, Serbian, and several other languages of the old Soviet Union), Arabic, Hebrew, and so on.

This is all well and good, but it doesn’t answer the needs of a writer preparing multilingual documents, except in the case that the multiple languages are accommodated in the same part of ISO 8859: it will happen some of the time, but most ‘interesting’ combinations will require switches of character set whenever the language changes.

So ISO (by this time, jointly with IEC) started development of an all-encompassing character set, to be numbered ISO/IEC 10646 (the difference of 10 000 is no accident). ISO/IEC 10646 was to accommodate every possible language in the world by the simple expedient of allowing 32-bit characters. Of course, no-one can comprehend a 32-bit character set, and so the set was to be structured, as a hypercube of different repertoires; the (0, 0, 0, *) repertoire would

² Given that Wales would have been represented by the BSI in the standardisation process.

be the same ISO-Latin 1, but all the other sets could be accommodated, too.

Independently, Apple and Microsoft got together to found the Unicode consortium, whose aim was to define 16-bit characters that would cover all the economically important world. This criterion of economic importance could easily have brought down the whole edifice: the (increasingly important) languages of the Far East are at best syllabic (e.g., Korean; Korea claims 11 000 of the code points in Unicode), or even one character per word (e.g., Chinese; a full classical Chinese repertoire would require well in excess of 65 536 characters, thus sinking a 16-bit code single-handedly).

Unicode's sponsors therefore enforced a process called 'Han unification', which aims to put the 'same' character in any of Chinese, Japanese and Korean in the same slot in the table. This unification is a distinctly dubious exercise: the same character may have different significance in the different languages, but they are all represented by the same code point. Contrariwise, the Latin 'H', the Russian 'H' (which sounds as Latin 'N') and the Greek 'H' (capital 'η') all get different code points despite having the same paper representation. For this reason (among others), there remain doubts as to whether the Japanese, in important particular, will adopt Unicode as a long-term replacement for their own national standards.

In the shorter term, however, there remained the possibility that there would be two conflicting standards for the future of character codes—a *de facto* one (Unicode) and ISO/IEC 10646. The ISO/IEC standard reached its (nominal) final ballot without addressing the relation to Unicode ... but (fortunately) it failed at that hurdle, and for that reason. Standards people are notorious for ignoring the real world³, but this time, they conceded defeat. ISO/IEC 10646 was edited to have the whole of Unicode as its (0,0,*,*) plane, and it has thus passed into the canon of published standards.

So we may now discuss Unicode without running out against the ISO/IEC standard: a splendid example of the behaviour known as "common sense prevailing".

Virtual Metafont and Fonts to Support Unicode


It is known that T_EX is a general-purpose programming language. In 'plain' text, we would type "hello world". For T_EX output we would type "‘hello

world’", which would be transparently converted to "hello world". Thus, the two grave accents and the two single quotes constitute 'programming'. In the last analysis, you can "do everything with T_EX".

When English is typeset, the convention is that the space, after the full stop is the end of a sentence, is expanded; T_EX makes provision for this to happen by way of the `\sfcode` mechanism. When French is typeset, the convention is that the space is not expanded; the `\sfcode` mechanism can provide this style of typesetting, as well (cf. the `\frenchspacing` macro of plain T_EX).

Other features of French typesetting are more difficult to provide in T_EX. For example, an exclamation mark is separated from the sentence: "en français!"; to program this, the exclamation mark needs to become an 'active character', which is always a tricky thing to do.

Setting the French quotation marks (known as guillemets) becomes even more tricky; the guillemets look like little << and >>, and the natural way to program them is by using repeated < or > characters; Bernard Gaulle's `french.sty` does this (also setting a space between the text quoted and the guillemets), but it's becoming more and more complicated; even more so when we consider the French rules for quotes within quotes.

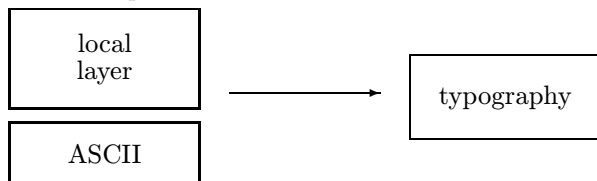
More problems arise when we consider the question of diacritics. English rather infrequently has diacritics, so it's not surprising that T_EX's method of dealing with them isn't perfect. To typeset an accented character, e.g. ä, one must type `"a`; which is typeset as two little boxes stacked on top of one another, rather like . This does work, but these composite glyphs no longer qualify (to T_EX) as something that it's willing to hyphenate—T_EX only hyphenates 'words' made up of sequences of letters. A language such as German, with hyphenation suppressed for many words, is hardly a language at all. These observations are what led to the definition of the Cork font encoding, in which a goodly proportion of Western European letters with diacritics appear as single characters; if they are thus represented, words containing them may be hyphenated.

With the Cork encoding, which is in effect an output encoding, we encounter a further problem relating to the nature of communication. The problem arises from the nature of character sets; while there are many well-established character sets, there are seriously different camps into which they fall. For example, the character 'P' (Thorn), appears in Microsoft Windows' character set but not in the Macintosh set, while 'Ω' appears in the Macintosh set but not in the Windows set; both of these sets are

³ The author has spent an unconscionably long period of his life on these things, and is therefore in a position to know.

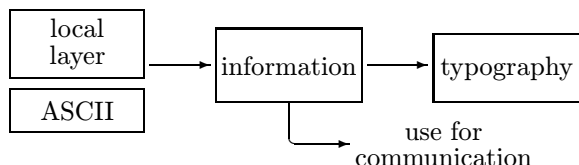
based on ASCII. To solve this problem, of encoding everything that appears in any character set, there has to be a super-encoding. This can be either a multi-character representation, as in the www encoding, html (for example the encoding would for é would be `é`), or a super-character set, as in Unicode.

In the present arrangement of typesetting technology, we have the situation where non-English users sit at a computer, and express their own language via a local layer in ASCII or a derivative of it – i.e., we have a picture like this:

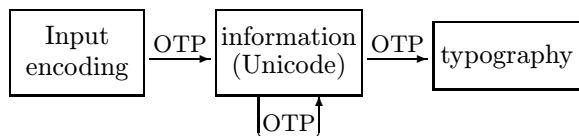


In this arrangement, the human interface allows the use of local characters, and the display will show what’s typed. The typography does the display job again (possibly differently); however, communication of the text to be typeset is difficult, because of the local nature of the interface.

The information to be transmitted needs to be encoded. There is no limit to the number of local encodings that may exist; equally, there is no constraint on the representations used by the typographic system. However, to facilitate the transmission of information, a common schema of its representation in the coded date must exist.



The ultimate mechanism for ensuring that such a schema exists is to require that everything be transmitted in a common encoding scheme; Ω employs ISO/IEC 10646/Unicode for this. Input text is transformed into Ω ’s internal ‘information’ by an Omega Translation Process (OTP); OTPs may also be used to transform the information during its processing withing Ω , and an OTP is also used to derive the coding of the font, to be used for typesetting, from the Unicode-encoded information within Ω :



Conceptually, at least, this is exactly what one wants. In practice, the usefulness of Ω remains to be seen; the implementors are promising a version (‘soon’) that progresses from the status of pre-test. I, for one, am eagerly awaiting its appearance.

References

Y. Haralambous. “Arabic, Persian and Ottoman \TeX for Mac and PC”. *TUGboat* **11**(4), 520–524, 1990.

Y. Haralambous. “ \TeX and those other languages”. *TUGboat* **12**(34), 539–548, 1991.

Y. Haralambous. “The Khmer script tamed by the Lion (of \TeX)”. *TUGboat* **14**(3), 260–270, 1993.

Y. Haralambous. “An Indic \TeX preprocessor — Sinhalese \TeX ”. *TUGboat* **15**(3), 301–301, 1994.

Y. Haralambous and Plaiçe, John. “First applications of Ω : Adobe Poetica, Arabic, Greek, Khmer”. *TUGboat* **15**(3), 344–352, 1994.

Modern Catalan Typographical Conventions

Gabriel Valiente Feruglio

University of the Balearic Islands

Mathematics and Computer Science Department

E-07071 Palma de Mallorca (Spain)

Email: valiente@ps.uib.es

Abstract

Many languages, such as German, English and French, have a traditional typography. However, despite the existence of a well-established tradition in scientific writing in Catalan, dating back to 1273 in the writings of Ramon Llull, and despite the early introduction of the Gutenberg printing press in Catalonia, where the first printed book in Catalan appeared in 1474, there are not yet any standard encompassing typographical conventions for scientific writing.

Typographical rules are proposed in this paper which reflect the spirit found in ancient Catalan scientific writings while conforming to modern typographical conventions. Some of these typographical rules are realized as a set of \TeX definitions, and they are meant to be incorporated in Catalan extensions of \TeX and \LaTeX . In addition, the proposal is also expected to contribute to the development of standard rules for scientific writing in Catalan.

Introduction

*Nothing in the design of \TeX
limits it to the American alphabet.*

*Fine printing is obtained
by fine tuning to the language
or languages being used.*

— D. E. KNUTH, *The \TeX book* (1990)

The widespread use of \TeX and \LaTeX in Catalan universities is causing the adoption of alien typographical conventions within documents written in Catalan. Even the `catalan` option of the `babel` package in \LaTeX advocates some typographical conventions which are foreign to Catalan. On the other hand, language-specific options in `babel` for languages such as German and French, which have well-established typographical conventions, accurately reflect such conventions.

There are not yet any standard encompassing typographical conventions for scientific writing in Catalan. While existing manuals of style deal with orthographic and typographical conventions for journalism (Coromina, 1993) and for literature (Joseph, 1991; Solà, 1990), books dealing with scientific writing focus primarily on linguistic aspects (Riera, 1993) and perhaps the most commonly accepted reference to scientific writing style is the way it is realized in an encyclopedia (DIGEC, 1989). A style book for authors who are themselves editors,

however, has just appeared (Pujol, 1995) and another style book is about to appear (Mestres, 1995) which may soon become another accepted reference source for scientific writing style as well.

A critical review of the degree of support of the Catalan alphabet in \TeX , as well as in international encoding standards, is presented in the following section. Different aspects of Catalan scientific writing style are discussed in the following sections, in the light of their (lack of) support in \TeX and \LaTeX , and they are grouped under the headings of spacing, quoting, hyphenation, and punctuation conventions. The paper concludes with a discussion of foreign typographical conventions currently encoded in the `catalan` option of the `babel` package in \LaTeX .

The Catalan alphabet

All Catalan characters belong to the ISO 8859-1 coding scheme, known as ISO LATIN-1, with the only exception (Valiente, 1993) of the digraph LATIN LETTER L MIDDLE DOT LATIN LETTER L in its uppercase (LL) and lowercase (ll) forms.

The digraph is not a letter of the Catalan alphabet, and it is not a ligature, although it could in principle be obtained in \TeX using ligatures as an input encoding mechanism; that is, ligatures `L.L` and `l.l` could be defined which would pick up the LL digraph and the ll digraph. It must be noted,



Figure 1: ISO/IEC 10646 prototypical glyphs

however, that the corresponding hyphenation point would then be missed.

Hyphenation of the *ll* digraph is 1-1, that is, it is replaced by two letters, and a similar phenomenon happens in other languages. For instance, in German *Drucker* becomes *Druk-ker*, in Swedish *tillaga* becomes *till-lage*, in Dutch *latje* becomes *lat-je* but *laatje* becomes *la-tje*, etc. All these cases fit in \TeX 's discretionary hyphenation mechanism.

Actually the characters are defined as special digraphs, with a raised dot as diacritic mark.

Encoding. The Cork encoding (Haralambous, 1992), an early attempt at extending the 7-bit Computer Modern encoding described in the five volumes of *Computers and Typesetting* to an 8-bit font encoding, covers the accented characters used in European languages, among them Catalan. It does not, however, include a character position for the *ll* digraph (Valiente, 1993).

An approximation to the *ll* digraph, the character combinations LATIN CAPITAL LETTER L WITH MIDDLE DOT and LATIN SMALL LETTER L WITH MIDDLE DOT, appears for the first time in the 32-bit encoding ISO/IEC 10646, in positions 0000013F and 00000140 (ISO, 1993, p. 20–21). Figure 1 shows the prototypical glyphs found therein.

The corresponding entity names `Lmidot` and `lmidot` already appear in ISO 8879 (ISO, 1986, p. 116) as Latin alphabetic characters used in Western European languages.

Furthermore, in Unicode, a 16-bit encoding which is a proper subset of the ISO/IEC 10646 32-bit encoding, the character combinations LATIN CAPITAL LETTER L WITH MIDDLE DOT and LATIN SMALL LETTER L WITH MIDDLE DOT appear again, in positions 013F and 0140 of the EUROPEAN LATIN block (Unicode, 1991, pp. 180–182). The prototypical glyphs found therein are shown in figure 2.

Strictly speaking, these ISO character combinations are not characters of the Catalan alphabet, and they should soon be replaced by the *ll* digraph, before they get implemented in the next generation of HTML browsers and in the next generation of



Figure 2: Unicode prototypical glyphs

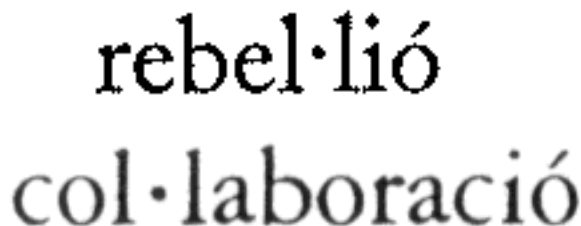


Figure 3: Diversity of graphical representation

typesetting systems. As a matter of fact, the Ω system (Haralambous, 1994) already implements the Unicode standard. A proposal for modification has been dealt with at length at the ISO 8859 and ISO 10646 discussion lists, and it will be submitted to ISO and Unicode as soon as it is agreed by academic authorities throughout the Catalan-speaking countries.

Prototypical rendering proposal. Parallel to the discussion of the best name for denoting such a digraph with a diacritic mark, is the definition of a corresponding graphical representation or glyph.

The standard which established the glyph to be used for representing the digraph was rather loose in the definition, only stating that “it will be written by putting a raised dot between the two letters” (Actes, 1906, pp. 194–200). Neither the distance between the two letters nor the raise and size of the middle dot were established by the standard.

Such an ambiguity, probably caused by the provisional character of the standard, has originated a great diversity in the practical application which can still be found nowadays, as can be seen in the samples of figure 3. Orthographic mistakes are not unusual: text processing systems lacking a discretionary hyphenation mechanism produce a wrong hyphenation of the *ll* digraph, while the use of a centered dot as middle dot may lead to unexpected results when changing fonts, as can be seen in the samples of figure 4.

Already in 1923 Pompeu Fabra complained about the excessive separation between the two letters being used by printers (Fabra, 1984, pp. 310–312) and interpreted the standard in the sense that the separation had to be the same as the normal

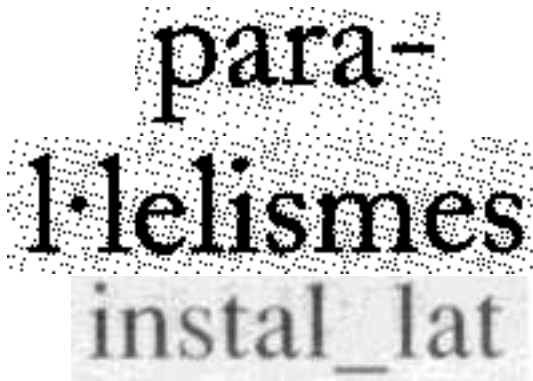
Figure 4: Non- \TeX text processing

Figure 5: Early graphical representation



Figure 6: First standard proposal

separation between the two letters when they are not written with a middle dot. The same interpretation can be found in Pujol (1995, pp. 339–340). As a matter of fact, this interpretation prevailed until digital typography displaced traditional typography.

Figure 5 shows samples of some of the first printed representations of the digraph, taken from a 1909 edition (Guasp Printer, Palma de Mallorca), a 1926 edition (Alcover Printer, Palma de Mallorca), and from a 1934 edition (Calatayud Printer, Sóller, Mallorca) of Alcover (1896–1931).

The lack of a clear standard has motivated two additional proposals. In the first one (Mestres, 1990) the size and raise of the middle dot is discussed, although the exact place where it should go is not established (see figure 6). In the second standard proposal (Valiente, 1993) the raise of the middle dot is established, but the distance between the two letters is not discussed.

The present proposal consists of defining the character as a digraph formed by two letters, with a middle dot as diacritic mark. The middle dot has to



Figure 7: Proposed prototypical rendering

be located between the two letters and equidistant from their stems. In order to facilitate reading, the middle dot has to be centered at half the height of uppercase letters. This is due to the fact that the digraph only appears between vowels. Moreover, between the two letters there has to be the same distance as there is between the two letters when they are not written with a middle dot, in order not to break continuity of words containing the digraph.

There seems to be no definitive need for the middle dot to be smaller than a period, contrary to the proposal in Mestres (1990). Boldface fonts are the only ones out of 27 standard \TeX fonts where a smaller middle dot would be, to the author's taste, aesthetically more pleasant. As a matter of fact, most (if not all) old and modern printed representations of the digraph share the fact that the middle dot and the period have the same size.

A prototypical glyph reproducing the standard proposal is presented in figure 7 and also in Appendix I for the main \TeX text fonts used in \LaTeX .

Implementing the proposal. In order to produce the *LL* and *ll* digraphs in \TeX , an input mechanism is first needed. Reasonable choices involve the definition of active characters, font ligatures, and control sequences.

- Type $\text{\L}\text{\L}$ and $\text{\l}\text{\l}$ and make the dot (period) active. This approach was developed by Claudio Beccari in the POLYGLOT package. Although such an input convention would be quite natural for most Catalan users, making the period (or any other punctuation mark) active is rather dangerous since it may have unexpected consequences when in math mode.
- Type $\text{\L}\cdot\text{\L}$ and $\text{\l}\cdot\text{\l}$ and make the raised dot active. This approach was developed by Xavier Gràcia and is another natural input convention for Catalan users, but it hinders portability, since the raised dot does not belong to the set of ASCII visible characters.

- Type `L.L` and `l.l` and define corresponding ligatures in all text fonts, which pick up the *LL* digraph and the *ll* digraph. Although being again a natural input convention for Catalan users, it also lacks portability and the corresponding hyphenation point would be missed when using this approach.
- Type a control sequence. The names `\LL` and `\ll` were proposed in Valiente (1993) as standard control sequence names for producing the *LL* digraph and the *ll* digraph, where it is discussed that `\ll` is already assigned to the \ll (much less than) relation in plain \TeX and \LaTeX and that it only occurs when in math mode, while the *ll* digraph is not supposed to be typed in math mode. Although these control sequence names could eventually be mapped to a more natural input convention in a local installation, Donald Knuth suggested preemption of the `\L` and `\l` macros (since it is not usual to have to include more than an occasional phrase in Polish into Catalan text) and to rewrite `\L` and `\l` as delimited macros with `.` as the delimiter. This solution is a quite natural input convention for Catalan \TeX users (it suffices to type `a\l.lot` in order to get *alot*) and it is robust, in the sense that the user gets a warning if the delimiter is omitted or if it is followed by anything other than `L` or `l`.

Typesetting the *ll* digraph involves either assembling together three characters, or picking up an existing *ll* character from a (real or virtual) font.

- Packing up three characters to form the *ll* digraph makes it quite difficult to get the right kerning, because the only metric information available about the characters in a font is the height, depth and width of the box enclosing them. For instance, the relative position of the stem of the two “l” letters within their enclosing box is needed in order to place the middle dot equidistant from the two stems, but this information can only be found in the METAFONT source code for the font. Even the actual diameter of a period is not guaranteed to be equal to the width of the box enclosing it.
- Since there is no *ll* character in any \TeX text font, not even in the DC fonts, which were supposed to cover all European languages, making a special text font for these two characters, as well as replacing some character in all text fonts by the *ll* digraph, would also hinder portability.

- Virtual fonts represent instead a portable solution, but the difficulty in getting the right kerning still exists.
- The technique of virtual METAFONT (Haralambous, 1995) comes then to rescue, since it allows to get the metric information needed for placing the middle dot at the exact position, according to the proposed prototypical rendering.

Assembling the *ll* digraph from three characters was the solution presented in Valiente (1993) and an improved version is given here, as a kind of *poor man’s solution* meant to be used at least until the techniques related to virtual METAFONT are available.

Implementing the rendering proposal at the \TeX or \LaTeX level, as opposed to the METAFONT level, means defining appropriate `\L.L` and `\l.l` control sequences that (a) produce the right hyphenation, (b) keep the normal separation between the two letters, (c) place the middle dot at half the distance between the stems of the letters, and (d) raise the middle dot to half the height of uppercase or lowercase letter `L` minus half the height of the middle dot.

Correct hyphenation is obtained by introducing appropriate discretionary break points, and normal separation between the two letters can be *approximated* by the definition

```
\newskip\zzz
\def\allowhyphens{\nobreak\hskip\zzz}
\edef\Lslash{\L} % save Polish \L
\def\L.L{\allowhyphens
\discretionary{L-}{L}{%
\hbox{L}%
\hbox to 0pt{\hbox{.}\hss}%
\hbox{L}}%
\allowhyphens }
```

and similarly for lowercase, where the horizontal and vertical displacement can be determined by trial and error for each particular font, after storing the component characters in boxes in order to know their dimensions *at macro expansion time*. By expressing such displacements in terms of, say, the width of letter `L`, displacements will be scaled together with the font and then the rendering proposal can be implemented for any font at any magnification.

Since such displacements depend on the current font in use at macro expansion time, appropriate values for `\leftdim` (horizontal displacement) and `\raisedim` (vertical displacement) can be determined and tabulated for each text font, and they have already been computed by the author for all

T_EX standard text fonts. However, a test is still needed in order to choose the right displacement values based on the *file name* of the current font. An intermediate solution, in the spirit of Valiente (1993), is the use of `\fam` (font family) values in the test, which are set by plain T_EX but no longer by L^AT_EX, and it is the solution adopted for typesetting this article. A provisional solution for L^AT_EX is also given in Appendix II which tests internal `\f@family`, `\f@series` and `\f@shape` values to deduce which font is in use.

Spacing conventions

French typographical conventions dictate that a little white space should be added before the so-called *double punctuation* characters (Gaulle, 1995), as for instance in:

Comptava que l'illa de Mallorca tenia
unes 300 milles que la circüen en torn;
i Menorca era cap a la banda de Sardenya.

These characters are COLON, SEMICOLON, EXCLAMATION MARK and QUESTION MARK. Similar conventions can be found in old Catalan texts (for instance in Reixac, 1749; Alcover, 1896–1934), also surrounding APOSTROPHE with extra white space. These conventions, however, are of no use in modern Catalan writing and they should not be encouraged by any Catalan style file.

Quoting conventions

Both double quotes (also called *saxon quotes*) and latin quotes (also called *guillemets*) find widespread use in modern Catalan texts, where their use is justified in solving the case of quotations inside quotations. While some authors prefer double quotes to latin quotes (Solà, 1990, p. 102; Pujol, 1995, p. 330), and only use latin quotes within text that is already enclosed in double quotes, other authors recommend their use the other way around, even suggesting the use of single quotes to enclose words which are already enclosed in double quotes (Joseph, 1991, pp. 149–150).

Although authors usually refer to particular aesthetic criteria justifying their choice of quoting conventions, an important aesthetic criterion seems to have been overlooked. Opening quotes may appear right after an apostrophe, and in such a case the apostrophe clashes with double quotes or with single quotes but not with latin quotes.

Therefore, the best choice in Catalan texts would be that of latin quotes, using double quotes

only within text already enclosed in latin quotes, and using single quotes only within text already enclosed in double quotes already enclosed in latin quotes. Maybe the development of French guillemets was already an answer to the aesthetic question of having an apostrophe being immediately followed by double quotes or by single quotes.

Hyphenation conventions

Catalan grammar, as well as current typographical conventions, establishes that syllables consisting of a single vowel should not be left alone at the beginning or end of a line, thereby reducing the number of possible break points in all but the smallest words. One exception to this rule is the case of a word preceded by an apostrophe, in which case hyphenation is allowed *after* the vowel.

Hyphenation of a word before or after an apostrophe has to be avoided, while keeping the break point after the vowel following the apostrophe. The solution given in Valiente (1993) is to set `\catcode'\ '=11` in order to include the pattern '2h in the language-specific pattern file, and to set `\left-hyphenmin=1` and `\righthyphenmin=3`. It is clear that such a solution is not sufficient, even when setting `\lefthyphenmin=2` and `\righthyphenmin=2`.

Another important exception to consider, found in modern publications but not yet covered by any style book, is the case of a word preceded by opening quotes or followed by closing quotes. In such a case the quotes should count as another character in the account for `\lefthyphenmin`, in such a way that, for instance, ‘‘u-na’’ could also be broken.

Punctuation conventions

When in math mode, T_EX treats the comma as a punctuation mark and the period (decimal point) as an ordinary symbol, putting a little extra space after the comma. In order to get the right spacing in a large decimal number such as 1,234,567.89

`$1{,}234{,}567.89$`

has to be written (Knuth, 1990, p. 134) instead of just

`$1,234,567.89$`

According to Catalan orthography, however, the roles of the comma and the decimal point are reversed, and a similar phenomenon happens in other European languages, among them Czech, French, German and Polish. The same number is expressed in Catalan by either 1.234.567,89 or 1 234 567,89 and would have to be written

`$1.234.567{,}89$`

or

`$1\,234\,567{,}89$`

in order to get the right spacing. This does not seem to be a practical solution, given the wider use of the comma as decimal separator in Catalan than as separator for large numbers in English.

Foreign typographical conventions

The `catalan` option of the `babel` package (release 3.4) in \LaTeX supports several typographical conventions that have been taken from the `spanish` option but which do not reflect modern Catalan typographical conventions.

Masculine and feminine ordinal indicators, which consist of a raised symbol such as “1st” or “1^a” in Spanish, correspond to abbreviations by contraction, without the final period, of their respective names in Catalan (Solà, 1990, p. 75; Pujol, 1995, p. 227), as shown in the following table.

Ordinal	Name	English	Spanish	Catalan
first	primer	primera	1 st	1 ^a 1 ^a 1r 1a
second	segon	segona	2 nd	2 ^a 2 ^a 2n 2a
third	tercer	tercera	3 rd	3 ^a 3 ^a 3r 3a
fourth	quart	quarta	4 th	4 ^a 4 ^a 4t 4a
fifth	cinquè	cinquena	5 th	5 ^a 5 ^a 5è 5a
sixth	sisè	sisena	6 th	6 ^a 6 ^a 6è 6a

The letter N TILDE does not belong to the Catalan alphabet. Whenever it appears in a Spanish word, for instance, temporarily switching to that language makes the right typographical conventions apply to that word. Hyphenation is perhaps the most important one of such conventions; switching to Spanish would produce the right hyphenation for the first last name of the author, `Va-lien-te`, and switching to Italian would produce the right hyphenation for the second last name of the author, `Fe-ru-glio`, instead of `Va-li-en-te` and `Fe-ru-gli-o` which would be produced under Catalan hyphenation rules.

Uppercase and lowercase mappings differ from language to language. For instance, in French and Spanish LATIN SMALL LETTER E WITH ACUTE ACCENT may map to LATIN CAPITAL LETTER E. In Catalan, however, diacritic marks have to be preserved in uppercase form, although the influence of Spanish press in Catalan-speaking countries has resulted in non-accented uppercase characters found in Catalan texts as well (Joseph, 1991, pp. 103–132).

In \TeX the argument of `\uppercase` and `\lowercase` is converted to uppercase form or to lowercase form using the `\uccode` and `\lccode` tables, which only cover characters from a font and not composite characters obtained with macros such as `\‘a`. In the latest versions of \TeX , however, `\uppercase{\‘a}` becomes `\‘a` but `\uppercase{\‘{i}}` and `\lowercase{\‘I}` both become `\‘i` instead of `\‘{i}`. The situation does not really get solved unless fonts containing accented characters are used, which constitutes just another reason to switch to DC fonts.

Typographical rules

The previous discussion can be summarized in the following ten typographical rules.

1. Do not type 1.1 or 1-1 for the *ll* digraph. Use the control sequences `\L.L` (for uppercase) and `\l.l` (for lowercase) instead.
2. Do not add extra white space before COLON, SEMICOLON, EXCLAMATION MARK, QUESTION MARK and APOSTROPHE.
3. Use French guillemets as quotation marks, leaving double quotes and eventually also single quotes to quote words which already belong to a quotation.
4. Hyphenate accented and diacritized words as well. Either use the multilingual \TeX patch or switch to the T1 font encoding in \LaTeX .
5. Do not use the period as decimal separator. Use the comma instead, as in `3{,}14159`.
6. Do not use foreign ordinal indicators. Type `1r`, `2n`, `3r`, `4t`, `5\‘e`, `6\‘e`, etc. to get masculine ordinal indicators, and type `1a`, `2a`, `3a`, `4a`, `5a`, `6a`, etc. to get feminine ordinal indicators.
7. Switch languages to get hyphenation and particular typographical conventions apply to foreign words.
8. Do not forget accents and diacritics in uppercase words.
9. Check out style manuals when in doubt. Authors are increasingly becoming editors and sometimes even printers ourselves.
10. Join the CATALA-TEX discussion list to further discuss these typographical rules and to get them implemented at your local site. Send the message

`SUBSCRIBE CATALA-TEX Name Surnames`

to the address

`LISTSERV@CESCA.ES`

and an introductory message will be sent back to your account.

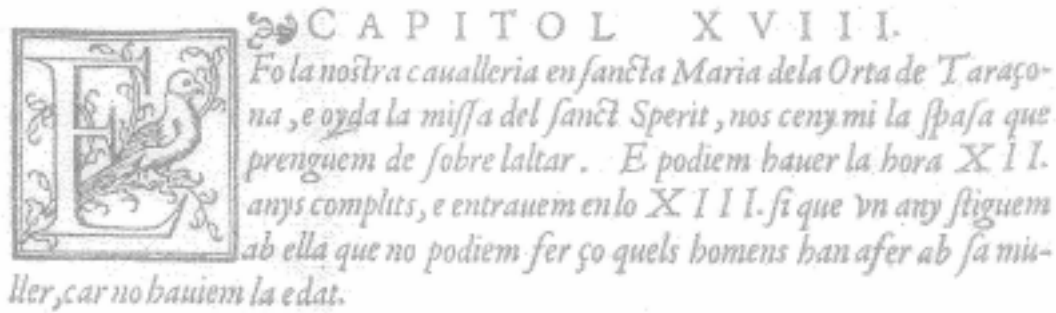


Figure 8: Medieval Catalan writing

Some of these typographical rules are illustrated by the following chapter of the *Libre dels feits del rei En Jacme*, the book of facts by Jaume the Conqueror (1208–1276):

E fo la nostra cavalleria en sancta Maria de l’Horta de Taraçona, que, oïda la missa de sent Espirit, nós cenyim l’espasa que prenguem de sobre l’altar. E podíem llaora haver dotze anys complits e entràvem en lo tretzè, sí que un any estiguem ab ella que no podíem fer ço que els hòmens han a fer ab sa muller, car no haviem l’edat.

The quotation has been taken from a modern edition (Jaume I, 1982), whereas the oldest preserved manuscript dates back to 1343. Compare it with the printed edition presented in figure 8, taken from (Chronica, 1557).

Acknowledgements

Much of the material related to the Catalan alphabet resulted from discussions with Francesc Comellas and Xavier Gràcia (Universitat Politècnica de Catalunya), Josep M. Mestres (Institut d’Estudis Catalans), Joan Solà (Universitat de Barcelona), Josep M. Pujol (Universitat Rovira i Virgili), and Joan Alegret (Universitat de les Illes Balears). Last, but not least, Pierre MacKay has presented this paper at the 16th Annual Meeting of the T_EX Users Group. Sincere thanks to them all.

Bibliography

- Actes del Primer Congrés Internacional de la Llengua Catalana. Barcelona, 1906. Facsimile edition by Vicens-Vives, Barcelona, 1985.
- Alcover, A. M. *Rondaies Mallorquines d’en Jordi des Racó*. Palma de Mallorca, 1896–1931.
- Badia, L. *Rudiments de tipografia*. Ed. Patronal d’Assistència Social, Barcelona, 1934.
- Chronica, o commentari del gloriosissim e invictissim Rey en Iacme. València, 1557. Facsimile edition by Ajuntament de València, 1994.
- Coromina, E. *El 9 Nou: Manual de redacció i estil*, 3rd Edition. Ed. Eumo, Vic, 1993.
- DIGEC. *Gran enciclopèdia catalana*, 2nd Edition. Ed. Enciclopèdia Catalana, Barcelona, 1989.
- Fabra, P. *Converses filològiques II*. EDHASA, Barcelona, 1984.
- Gaulle, B. *Notice d’utilisation du style french multilingue*. Electronic document distributed with the french package, Version 3.36, 1995.
- Haralambous, Y. “T_EX Conventions Concerning Languages.” *T_EX and TUG NEWS* 1(4), 1992, pp. 3–10.
- Haralambous, Y. and Plaice, J. “Ω, a T_EX Extension Including Unicode and Featuring Lex-like Filtering Processes.” In *EuroT_EX Proceedings*, 1994, pp. 154–167.
- Haralambous, Y. and Plaice, J. “Ω + Virtual METAFONT = Unicode + Typography.” *Cahiers GUTenberg*, 21, 1995, pp. 1–13.
- ISO, International Organization for Standardization. *Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*. Geneva, Switzerland, 1986. International Standard ISO 8879.
- ISO, International Organization for Standardization. *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*. Geneva, Switzerland, 1993. International Standard ISO 10646-1.
- Jaume I. *Crònica o llibre dels fets*. Ed. 62, Barcelona, 1982.
- Joseph, M. *Com es fa un llibre*. Ed. Pòrtic, Barcelona, 1991.
- Knuth, D. E. *The T_EXbook*. Addison-Wesley, Reading, Massachusetts, 9th printing, 1990.

- Mestres, J. M. “A l’entorn de la ela geminada.”
Escola Catalana, 2(66):7–8 and 2(70–72):7–10,
1990.
- Mestres, J. M., Costa, J., Oliva, M. and Fité, R.
Manual d’estil. Ed. Eumo, Barcelona, 1995. To
appear.
- Pujol, J. M. and Solà, J. *Ortotipografia*. Ed.
Columna, Barcelona, 1995.
- Reixac, B. *Instruccions per la ensenyança de mi-
nyons*. Anton Oliva, Girona, 1749. Facsimile edi-
tion by Edicions de la Universitat de Barcelona,
1983.
- Riera, C. *Manual de català científic*, 2nd Edition.
Ed. Claret, Barcelona, 1993.
- Solà, J. and Pujol, J. M. *Tractat de puntuació*, 2nd
Edition. Ed. Columna, Barcelona, 1990.
- The Unicode Consortium. *The Unicode Standard:
Worldwide Character Encoding. Version 1.0, Vol-
ume 1*. Addison-Wesley, Reading, Massachusetts,
1991.
- Valiente, G. and Fuster, R. “Typesetting Catalan
Texts with T_EX.” *TUGboat* 14(3), 1993, pp. 252–
259.

Appendix I

Prototypical rendering in font `cmr10`.

L̇L ḢH

Prototypical rendering in font `cmr10`.

L̇L ḢH

Prototypical rendering in font `cmsl10`.

L̇L ḢH

Prototypical rendering in font `cmbx10`.

L̇L ḢH

Prototypical rendering in font `cmtt10`.

L̇L ḢH

Prototypical rendering in font `cmss10`.

L̇L ḢH

Prototypical rendering in font `cmcsc10`.

L̇L L̇L

Appendix II

Horizontal displacement values for L^AT_EX font-switching internals

	\f@family	\f@series	\f@shape	file name	uppercase	lowercase
\rm	cmr	m	n	cmr10	0.44	0.50
\it	cmr	m	it	cmti10	0.32	0.40
\sl	cmr	m	sl	cmsl10	0.36	0.30
\bf	cmr	bx	n	cmbx10	0.39	0.48
\tt	cmmt	m	n	cmmt10	0.73	0.50
\sf	cmss	m	n	cmss10	0.50	0.59
\sc	cmr	m	sc	cmcsc10	0.43	0.51

Poor man's L_L and H definitions for L^AT_EX

```

\newskip\zzz \def\allowhyphens{\nobreak\hskip\zzz}
\newdimen\leftdim \newdimen\raisedim
\def\LDOTL#1#2#3{%
  \setbox0\hbox{#1}%
  \setbox1\hbox{#2}%
  \leftdim=0pt
  \raisedim=0pt
  \advance\leftdim by -#3\wd0
  \advance\raisedim by \ht0
  \divide\raisedim by 2
  \advance\raisedim by -0.5\ht1
  \allowhyphens \discretionary{#1-}{#1}{\copy0
  \hbox to 0pt{\hskip\leftdim\raise\raisedim\copy1\hss}\copy0}\allowhyphens
}
\makeatletter
\edef\cmtt@family{cmtt} \edef\cmss@family{cmss} \edef\bx@series{bx}
\edef\it@shape{it} \edef\sl@shape{sl} \edef\sc@shape{sc}
\edef\Lslash{\L} \edef\lslash{\l} % save Polish \L and \l
\def\L.L{%
  \ifx\f@family\cmtt@family \LDOTL{L}{.}{0.73}% cmtt10
  \else\ifx\f@family\cmss@family \LDOTL{L}{.}{0.50}% cmss10
  \else\ifx\f@shape\sc@shape \LDOTL{L}{.}{0.43}% cmcsc10
  \else\ifx\f@series\bx@series \LDOTL{L}{.}{0.39}% cmbx10
  \else\ifx\f@shape\sl@shape \LDOTL{L}{.}{0.36}% cmsl10
  \else\ifx\f@shape\it@shape \LDOTL{L}{.}{0.32}% cmti10
  \else \LDOTL{L}{.}{0.44}% cmr10
\fi\fi\fi\fi\fi }
\def\l.l{%
  \ifx\f@family\cmtt@family \LDOTL{l}{.}{0.50}% cmtt10
  \else\ifx\f@family\cmss@family \LDOTL{l}{.}{0.59}% cmss10
  \else\ifx\f@shape\sc@shape \LDOTL{l}{.}{0.51}% cmcsc10
  \else\ifx\f@series\bx@series \LDOTL{l}{.}{0.48}% cmbx10
  \else\ifx\f@shape\sl@shape \LDOTL{l}{.}{0.30}% cmsl10
  \else\ifx\f@shape\it@shape \LDOTL{l}{.}{0.40}% cmti10
  \else \LDOTL{l}{.}{0.50}% cmr10
\fi\fi\fi\fi\fi }
\makeatother

```

The 17th Annual T_EX Users Group Meeting

$$\text{П} \text{O} \text{A} \text{T} \text{T} \text{E} \text{X} = \left\{ \begin{array}{l} \text{Polytechnic} \\ \text{Polymath} \\ \text{Polyglot} \end{array} \right.$$

The Joint Institute for Nuclear Research
July 28 – August 2, 1996



Dubna: what is it?

Dubna was founded in 1956 when the Convention establishing the Joint Institute for Nuclear Research was signed. The town is situated on the picturesque banks of the Volga river and the Moscow sea 120 km to the north of Moscow.

There is no harmful environmental impact of the plants, this together with the large tracts of forest in the environs of Dubna, vast water area with small islands is quite favorable for the sphere of tourism and rest. The Volga embankment is one of the prettiest parts of the town that was built in the midst of a forest. It takes just a few minutes to get to the forest from the shopping centre on foot.

Small as it is, Dubna is a real scientific metropolis. The Joint Institute for Nuclear Research (JINR) plays an important role as a coordinator of the investigations of the scientists from 18 JINR member-state institutes. The wide international scientific and technical cooperation is one of the fundamental concepts of the JINR. The town has great experience in holding international conferences, and the exchange of delegations between countries in the sphere of science, education and culture . . .

There will be more information about Russia, the town of Dubna itself, and the Conference in the *TUGboat* 16(4).

The T_EX Users Group would very much like to invite you to take part in our seventeenth annual meeting П_ОА_ТТ_ЕX to be held at the Joint Institute for Nuclear Research (Dubna, Russia) between July 28th – August 2nd, 1996. This is the second annual TUG meeting outside North America, and the first meeting that takes place in a country with a non-latin alphabet. TUG, *CyrTUG*, and the Conference Committee hope to see a variety of techniques, knowledge and use of different languages presented at the meeting. As usual, courses will be offered in the days before or after the conference.

The Conference Committee foresees a cost in the range \$550–600 USD. This sum includes the registration fee, lodging (6 nights with 6 breakfasts, 6 lunches, 6 dinners), coffee/tea breaks, social events, and transport between Sheremetsevo Airport and Dubna. The payment can be made in the following way: \$100 of the payment will be transferred via a bank before June 1st, and the rest will be payable upon arrival at the Conference. We hope to get support from some funds sponsoring participants from Eastern Europe and students.

T_EX Consulting & Production Services

Information about these services can be obtained from:

T_EX Users Group
 1850 Union Street, #1637
 San Francisco, CA 94123, U.S.A.
 Phone: +1 415 982-8449
 Fax: +1 415 982-8559
 Email: tug@tug.org

North America

Anagnostopoulos, Paul C.

Windfall Software,
 433 Rutland Street, Carlisle, MA 01741;
 (508) 371-2316; greek@windfall.com

We have been typesetting and composing high-quality books and technical Publications since 1989. Most of the books are produced with our own public-domain macro package, ZzT_EX, but we consult on all aspects of T_EX and book production. We can convert almost any electronic manuscript to T_EX. We also develop book and electronic publishing software for DOS and Windows. I am a computer analyst with a Computer Science degree.

Cowan, Dr. Ray F.

141 Del Medio Ave. #134, Mountain View, CA 94040;
 (415) 949-4911; rfc@netcom.com

Twelve Years of T_EX and Related Software Consulting: Books, Documentation, Journals, and Newsletters
 T_EX & L^AT_EX macropackages, graphics; PostScript language applications; device drivers; fonts; systems.

Hoening, Alan

17 Bay Avenue, Huntington, NY 11743; (516) 385-0736
 T_EX typesetting services including complete book production; macro writing; individual and group T_EX instruction.

NAR Associates

817 Holly Drive E. Rt. 10, Annapolis, MD 21401;
 (410) 757-5724

Extensive long term experience in T_EX 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

40453 Cherokee Oaks Drive,
 Three Rivers, CA 93271-9743;
 (209) 561-4585

Experienced in book production, macro packages, programming, and consultation. Complete book production from computer-readable copy to camera-ready copy.

Quixote Digital Typography, Don Hosek

555 Guilford, Claremont, CA 91711;
 (909) 621-1291; Fax: (909) 625-1342;
 dhosek@quixote.com

Complete line of T_EX, L^AT_EX, and METAFONT services including custom L^AT_EX style files, complete book production from manuscript to camera-ready copy; custom font and logo design; installation of customized T_EX 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

T_EX macro consulting.

Type 2000

16 Madrona Avenue, Mill Valley, CA 94941;
 (415) 388-8873; Fax: (415) 388-8865
 pti@crl.com

\$2.50 per page for 2000 DPI T_EX and PostScript camera ready output! We provide high quality and fast turnaround to dozens of publishers, journals, authors and consultants who use T_EX. Computer Modern, PostScript and METAFONT fonts available. We accept DVI and PostScript files only and output on RC paper. \$2.25 per page for 100+ pages, \$2.00 per page for 500+ pages; add \$.50 per page for PostScript.

Outside North America

TypoT_EX Ltd.

Electronical Publishing, Battyány u. 14. Budapest,
 Hungary H-1015; (036) 11152 337

Editing and typesetting technical journals and books with T_EX from manuscript to camera ready copy. Macro writing, font designing, T_EX consulting and teaching.