TUGboat

Volume 21, Number 3 / September 2000
2000 Annual Meeting Proceedings

TEX Users Group

Memberships and Subscriptions

TUGboat (ISSN 0896-3207) is published quarterly by the TEX Users Group, 1466 NW Naito Parkway, Suite 3141, Portland, OR 97209-2820, U.S.A.

2000 dues for individual members are as follows:

- Ordinary members: $75.
- Students: $45.

Membership in the TEX Users Group is for the calendar year, and includes all issues of *TUGboat* for the year in which membership begins or is renewed. Individual membership is open only to named individuals, and carries with it such rights and responsibilities as voting in TUG elections. A membership form is provided on page 000.

TUGboat subscriptions are available to organizations and others wishing to receive *TUGboat* in a name other than that of an individual. Subscription rates: $85 a year, including air mail delivery.

Periodical-class postage paid at Portland, OR, and additional mailing offices. Postmaster: Send address changes to *TUGboat*, TEX Users Group, 1466 NW Naito Parkway, Suite 3141, Portland, OR 97209-2820, U.S.A.

Institutional Membership

Institutional Membership is a means of showing continuing interest in and support for both TEX and the TEX Users Group. For further information, contact the TUG office (`office@tug.org`).

Printed in U.S.A.

Addresses

General correspondence, payments, etc.
TEX Users Group
P. O. Box 2311
Portland, OR 97208-2311
U.S.A.

Delivery services, parcels, visitors
TEX Users Group
1466 NW Naito Parkway
Suite 3141
Portland, OR 97209-2820
U.S.A.

Telephone

+1 503 223-9994

Fax

+1 503 223-3960

Electronic Mail (Internet)

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

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

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

To contact the Board of Directors:
`board@tug.org`

World Wide Web

`http://www.tug.org/`
`http://www.tug.org/TUGboat/`

Problems not resolved?

The TUG Board wants to hear from you:
Please email to `board@tug.org`

Calendar

2000

Nov 13 – Jan 6 — Gutenberg exhibit, including working replica of his original printing press, Louisville Free Public Library, Louisville, Kentucky.

2001

Feb 28 – Mar 3 — DANTE 2001, 24th meeting, Fachhochschule Rosenheim, Germany. For information, visit `http://www.dante.de/dante2001/`.

Mar 26 – 28 — XML World Europe, Amsterdam, Netherlands. For information, visit `http://www.xmlworld.org/`.

Apr 1 – Jun 15 — The Best of the Best: A traveling juried exhibition of books by members of the Guild of Book Workers. Ohio State University Library, Athens, Ohio. Sites and dates are listed at `http://palimpsest.stanford.edu/byorg/gbw`.

Apr 9 – 13 — Seybold Boston, Boston, Massachusetts. For information, visit `http://www.key3media.com/seyboldseminars/boston2001/`.

Apr 29 – May 2 — BachoTEX 2001, 9th annual meeting of the Polish TEX Users' Group (GUST), "Contemporary publishing TEXnology", Bachotek, Brodnica Lake District, Poland. For information, visit `http://www.gust.org.pl/BachoTeX/`.

May 14 – 17 — Congrès GUTenberg 2001, "Le document au XXIe Siècle", Metz, France. For information, visit `http://www.gutenberg.eu.org/manif/gut2001/`.

Jun 4 – 8 — Rare Book School Summer Session, University of Virginia, Charlottesville, Virginia. A series of one-week courses on topics concerning rare books, manuscripts, the history of books and printing, and special collections. For information, visit `http://www.virginia.edu/oldbooks`.

Jun 6 – 8 — Society for Scholarly Publishing, 23rd annual meeting, San Francisco, California. For information, visit `http://www.sspnet.org`.

Jun 13 – 17 — ACH/ALLC 2001: Joint International Conference of the Association for Computers and the Humanities, and Association for Literary and Linguistic Computing, New York University, New York. For information, visit `http://www.nyu.edu/its/humanities/ach_allc2001/`.

Jul 6 – Aug 18 — The Best of the Best: A traveling juried exhibition of books by members of the Guild of Book Workers. Columbia College Chicago Center for Book and Paper Arts, Chicago, Illinois. Sites and dates are listed at `http://palimpsest.stanford.edu/byorg/gbw`.

Jul 13 – 15 — TypeCon 2001, Rochester, New York. For information, visit `http://www.typecon2001.com`.

Jul 16 – Aug 10 — Rare Book School Summer Session, University of Virginia, Charlottesville, Virginia. A series of one-week courses on topics concerning rare books, manuscripts, the history of books and printing, and special collections. For information, visit `http://www.virginia.edu/oldbooks`.

TUG 2001

University of Delaware, Newark, Delaware.
For information, visit `http://www.tug.org/tug2001/`.

Aug 6 – 10 — Intermediate/Advanced LATEX training class.

Aug 12 – 16 — The 22nd annual meeting of the TEX Users Group, "2001: A TEX Odyssey".

Aug 12 – 17 — Extreme Markup Languages 2001: "There's Nothing so Practical as a Good Theory", Montréal, Canada. For information, visit `http://www.gca.org`.

Status as of 1 December 2000

For additional information on TUG-sponsored events listed above, contact the TUG office (+1 503 223-9994, fax: +1 503 223-3960, e-mail: `office@tug.org`). For events sponsored by other organizations, please use the contact address provided.

Additional type-related events and news items are listed in the Sans Serif Web pages, at `http://www.quixote.com/serif/sans`.

Aug 12 – 17	SIGGRAPH 2001, Los Angeles, California. For information, visit `http://www.siggraph.org/s2001/`.
Sep 8	WDA'2001: First International Workshop on Web Document Analysis, Seattle, Washington. For information, visit `http://www.csc.liv.ac.uk/~wda2001`.
Sep 10 – Oct 26	The Best of the Best: A traveling juried exhibition of books by members of the Guild of Book Workers. Dartmouth College, Hanover, New Hampshire. Sites and dates are listed at `http://palimpsest.stanford.edu/byorg/gbw`.
Sep 17 – 20	XML World 2001, San Francisco, California. For information, visit `http://www.xmlworld.org/`.
Sep 23 – 27	EuroTEX 2001, "TEX and Meta: the Good, the Bad and the Ugly Bits", Kerkrade, Netherlands. For information, visit `http://www.ntg.nl/eurotex/`.
Sep 29	29th Annual General Meeting of the Danish TEX Users Group (DK-TUG), Århus, Denmark. For information, visit `http://sunsite.dk/dk-tug/`.
Oct 24 – 26	4th International Conference on The Electronic Document, Toulouse, France. For information, visit `http://www.irit.fr/CIDE2001/`.
Nov 5 – Dec 21:	The Best of the Best: A traveling juried exhibition of books by members of the Guild of Book Workers. Smith College, Northampton, Massachusetts. Sites and dates are listed at `http://palimpsest.stanford.edu/byorg/gbw`.
Nov 9 – 10	ASM Symposium on Document Engineering, Atlanta, Georgia. For information, visit `http://www.documentengineering.org`.

Institutional Members

American Mathematical Society,
Providence, *Rhode Island*

Center for Computing Services,
Bowie, *Maryland*

CNRS - IDRIS,
Orsay, *France*

College of William & Mary,
Department of Computer Science,
Williamsburg, *Virginia*

CSTUG, *Praha*, *Czech Republic*

Florida State University,
Supercomputer Computations
Research, *Tallahassee*, *Florida*

Hong Kong University of
Science and Technology,
Department of Computer Science,
Hong Kong, *China*

IBM Corporation,
T J Watson Research Center,
Yorktown, *New York*

ICC Corporation,
Portland, *Oregon*

Institute for Advanced Study,
Princeton, *New Jersey*

Institute for Defense Analyses,
Center for Communications
Research, *Princeton*, *New Jersey*

Iowa State University,
Computation Center,
Ames, *Iowa*

Kluwer Academic Publishers,
Dordrecht, *The Netherlands*

KTH Royal Institute of
Technology, *Stockholm*, *Sweden*

Marquette University,
Department of Mathematics,
Statistics and Computer Science,
Milwaukee, *Wisconsin*

Masaryk University,
Faculty of Informatics,
Brno, *Czechoslovakia*

Max Planck Institut
für Mathematik,
Bonn, *Germany*

National Institute for Child
& Human Development,
Bethesda, *Maryland*

New York University,
Academic Computing Facility,
New York, *New York*

Princeton University,
Department of Mathematics,
Princeton, *New Jersey*

Space Telescope Science Institute,
Baltimore, *Maryland*

Springer-Verlag Heidelberg,
Heidelberg, *Germany*

Springer-Verlag New York, Inc.,
New York, *New York*

Stanford Linear Accelerator
Center (SLAC),
Stanford, *California*

Stanford University,
Computer Science Department,
Stanford, *California*

Stockholm University,
Department of Mathematics,
Stockholm, *Sweden*

University of Canterbury,
Computer Services Centre,
Christchurch, *New Zealand*

University College, Cork,
Computer Centre,
Cork, *Ireland*

University of Delaware,
Computing and Network Services,
Newark, *Delaware*

Universität Koblenz–Landau,
Fachbereich Informatik,
Koblenz, *Germany*

University of Oslo,
Institute of Informatics,
Blindern, *Oslo*, *Norway*

Università degli Studi di Trieste,
Trieste, *Italy*

Vanderbilt University,
Nashville, *Tennessee*

Vrije Universiteit,
Amsterdam, *The Netherlands*

TEX Consulting & Production Services

Information about these services can be obtained from:

TEX Users Group
1466 NW Naito Parkway, Suite 3141
Portland, OR 97209-2820, U.S.A.
Phone: +1 503 223-9994
Fax: +1 503 223-3960
Email: office@tug.org
URL: http://www.tug.org/consultants.html

North America

Loew, Elizabeth
President, TEXniques, Inc.,
675 Massachusetts Avenue, 6th Floor,
Cambridge, MA 02139;
(617) 876-2333; Fax: (781) 344-8158
Email: loew@texniques.com
Complete book and journal production in the areas of mathematics, physics, engineering, and biology. Services include copyediting, layout, art sizing, preparation of electronic figures; we keyboard from raw manuscript or tweak TEX files.

Ogawa, Arthur
40453 Cherokee Oaks Drive,
Three Rivers, CA 93271-9743;
(209) 561-4585
Email: Ogawa@teleport.com
Bookbuilding services, including design, copyedit, art, and composition; color is my speciality. Custom TEX macros and LATEX 2ε document classes and packages. Instruction, support, and consultation for workgroups and authors. Application development in LATEX, TEX, SGML, PostScript, Java, and ßC++. Database and corporate publishing. Extensive references.

Outside North America

DocuTEXing: TEX Typesetting Facility
43 Ibn Kotaiba Street,
Nasr City, Cairo 11471, Egypt
+20 2 4034178; Fax: +20 2 4034178
Email: main-office@DocuTeXing.com
DocuTEXing provides high-quality TEX and LATEX typesetting services to authors, editors, and publishers. Our services extend from simple typesetting and technical illustrations to full production of electronic journals. For more information, samples, and references, please visit our web site: http://www.DocuTeXing.com or contact us by e-mail.

Editor's notes

Robin Fairbairns
Computer Laboratory
University of Cambridge
UK
rf10@cam.ac.uk

Oxford (and Cambridge)

Here, finally, we have the proceedings of TUG 2000: TUG's annual meeting in Oxford ("the other place"). It may seem odd to have a Cambridge-based editor of an Oxford conference: but you may not understand why this would be remarkable, so I shall bore you all with a little academic history (some of it decidedly personal...).

My decision, of which of the two "old" English universities[1] to grace with my presence as a student, was made on the basis of deep academic consideration. Cambridge, when I first visited over the final weekend of the Cuban missile crisis in 1962, was beautiful under a frosty, clear sky; I spent two nights at the home of a school friend, whose father was a don of long standing, and all was wonderful.

When I first visited Oxford in early 1963, the weather was foul — windy, wet and cold; I had hoped to spend time with a friend from school, but he had left town for the weekend. Over all the years since, I've never had the opportunity to "get the feel" of the city: on each visit I've either had no spare time, or the weather has been bad.

Which is all terribly ... characteristic. The University of Cambridge (in something approaching its present form) was probably established by dissident Oxford students getting on for eight hundred years ago. Ever since, Cambridge people have been *expected* not to know about Oxford, and to despise all of Oxford's doings. We're supposed to despise their style of poling their punts[2], and to disparage their academic achievements. And vice versa.

I've never really believed in this silly caricature, so a week to get to know Oxford, based in the centre of the city, in weather as good as we get nowadays (with global warming apparently already upon us and bringing even more rain in our summers), was a real treat.

[1] There are other universities almost as venerable as Oxford and Cambridge in these islands, but I didn't know about them in 1963

[2] Which in fact are of a completely different design from those we use in Cambridge, so *would* be poled differently

The meeting

We have to thank the local team (led by Sebastian Rahtz and Kim Roberts) for a splendidly run conference. The college seemed to me ideal for the sort of "small" conference that TUG runs: compact enough that everyone could feel they were on top of the whole event, and yet spacious enough that we could spread out and feel comfortable. The social events (notably the reception in the University Museum) were splendid, and Kim's arrangement for those that wanted to go to an open-air dramatisation of Carroll's "Through the Looking Glass" was inspired (for this member of the audience at least, despite less-than-ideal weather).

A strong cast presented an intriguing set of papers, with meat for every taste in the TEX world. For me, the highlights were Mike Vulis on his VTEX/GEX system, discussion of the achievements and future of the PDFTEX and Omega projects, Don Deland's demonstration of his interactive courseware, and (of course) Frank Mittelbach's paper on directions for the LATEX output routine (which was voted best paper of the conference).

The papers

I took over preparation of these proceedings at a late stage, and from the start I experienced problems. Neither I, nor the printers that Kim Roberts had chosen, could print one of the pages of Kostin & Vulis' paper for the preprints: I don't believe I've ever before seen conference proceedings with an apology for the absence of a page for that reason.

The papers that follow represent much of the best in the conference, but there are sad omissions: none of DeLand's presentation of his use of IBM's techexplorer, the AsTEX, the PDFTEX or the Omega presentations is here represented. *TUGboat* hopes to present a detailed account of PDFTEX in a future issue, but we can do little but hope for papers covering the other topics.

The *difficult* papers Two papers presented particular technical difficulties, since both of them demanded use of the technology they described.

Alex Kostin & Mike Vulis described VTEX, and their paper included demonstrations of the capabilities of VTEX/GEX. I suppose we could have faked the effects, but the simpler course seemed to be to install and use a copy of the free (Linux) version of VTEX. The installation itself proved very simple, but (as mentioned above) there were problems with the output. VTEX produces its output of the paper in PDF, and the problem with the output wasn't apparent in Adobe's Acrobat Reader, merely when that program produced printer output. Interestingly, PostScript output of the troublesome page also defeated Adobe's Acrobat Distiller, so that at the time of the meeting I suspected a bug (that I couldn't at the time characterise) in the Acrobat suite. The bug was in fact in VTEX's failure to detect some infelicity in the data for one of the paper's diagrams, and has long since been corrected; the preprint of the paper was an ordinary LATEX document that used a series of `\includegraphics` commands on images of the working pages of the paper, and had an apology in place of the rogue page. The paper presents itself in glorious TEXnicolour; which you can't see in these proceedings, but which will be visible when the paper appears on *TUGboat*'s web site.

Frank Mittelbach described the outcome of experimental work on the algorithms desirable for a future version of LATEX: this is another in a long series of papers on directions towards the mythic LATEX 3 and in most respects is the answer to the average maiden's prayer. However, the version I worked with couldn't deal with the (old LATEX) construct `\twocolumn`[⟨*stuff*⟩], which is how the *TUGboat* class creates paper titles in a proceedings issue. So after much agonising, we have decided to set the paper with the title set separately, so as to demonstrate that the code he describes "works" (in so far as it does!).

Regrets

I need hardly repeat that I'm deeply ashamed at how long it has taken me to produce these proceedings: even now they wouldn't be with you had it not been for editing support from Barbara Beeton and the continuing sterling work done by Mimi Burbank behind the scenes. (In particular, Mimi's taken on the rôle of "Robin's conscience", prodding me every so often on the necessity of getting on with it!) The time since the meeting has been very full (I've moved house, for example), but the delays have, I admit, been mostly of my own making.

Several of the presentations at the conference have not resulted in papers in these proceedings (we print the pre-conference abstracts in these cases). Most of the presentations are sorry omissions from these proceedings, but we will perhaps pick up matter to publish in future editions of *TUGboat*. The lack of the papers comes as an awful warning to us all: we *must* attend the annual meetings to keep up with what is going on with TEX and its related technologies.

TUG 2000
Programme of events

Sunday, August 13, 2000

Mimi Jett, President of TUG
The Tao of TEX

Barbara Beeton
Unicode and math, a combination whose time has come — Finally!

Frank Mittelbach
Formatting documents with floats: A new algorithm for LATEX 2ε

Phil Taylor
NTS Report

Jonathan Fine
Line breaking and page breaking

Monday, August 14, 2000

Benjamin Bayart
The description language chosen for FDNTEX

Michel Lavaud
The AsTEX Assistant and Navigator

Hirotsugu Kakugawa
A device-independent DVI interpreter library for various output devices

Donald DeLand
Developing Interactive, Web-based Courseware

Alexander Berdnikov
Some experience in converting LH Fonts from METAFONT to Type1 format

Michael Vulis
Mixing TEX & PostScript: The GEX model

Tuesday, August 15, 2000

Hans Hagen
What is PDF?

Sebastian Rahtz
The history of pdfTEX

Erik Frambach
Fonts in pdfTEX

Han The Thanh
How pdfTEX can improve your pages

Hans Hagen
Graphics in pdfTEX

Ed Cashin
pdfTEX in a workflow

Hans Hagen
Going beyond static documents

Ed Cashin
Setting up pdfTEX

Berend de Boer
Postprocessing PDF

Han The Thanh
The future of pdfTEX

TUG Annual General Meeting

John Plaice
Omega version 2

Wednesday, August 16, 2000

Pedro Palao Gostanza
Fast scanners and self-parsing in TEX

Włodek Bzyl
Typesetting TEX documents containing computer code

Timothy Murphy
The Penrose notation: a LATEX challenge

David Carlisle
xmltex: A non validating (and not 100% conforming) namespace aware XML parser implemented in TEX

Sebastian Rahtz and Michel Goossens
PassiveTEX: XSL processing using TEX

Bernice Sacks Lipkin
LATEX and the personal database

Michael Downes
The `amsrefs` LATEX package and the `amsxport` BIBTEX style

Alexander Berdnikov
Chess fonts and chess macros for chess games and puzzles

Olga Lapko
"Russian style" with LATEX and Babel: what does it look like and how does it work

Dick Nickalls and Apostolos Syropoulos
A Perl port of the mathsPIC graphics package

Thursday, August 17, 2000

Tutorials

John Plaice
Omega Tutorial

Hans Hagen, et al.
ConTeXt and PDF Tutorial

Friday, August 18, 2000

Tutorials

Frank Mittelbach et al.
LATEX3 Tutorial

Sebastian Rahtz and Michel Goossens
XML and XSL Tutorial

The description language chosen for FDNTEX

Benjamin Bayart
10, rue du Croissant
75 002 Paris
France
bayartb@edgard.fdn.fr

Abstract

In this paper I will introduce the package description language that has been chosen to be used in FDNTEX. In short, this language is XML with a dedicated DTD. First, I'll introduce FDNTEX, to obtain a good representation of its requirements. Then, I'll introduce the data that have to be contained in the description files. Finally, I'll introduce briefly the DTD itself.

This paper has to be understood for what it is: first thoughts on how to obtain the right language. What has to be expressed by this language is something definite, but the DTD itself is in an early stage development at the time of writing. Thus, if the goals described in the firsts sections and the DTD described later are in conflict, consider the DTD is wrong.

Vocabulary

The word "package" will be used with several meanings in this paper, this can be troublesome in certain places. A *package* can be, depending on the context:

- a LATEX style file, like `array.sty`;
- a part of the distribution, *e.g.*, the package containing TEX itself, or the one containing Ω;
- a binary version, in a given flavour, of the previous one, *e.g.*, the `rpm` file containing Ω, or TEX.

It's sometimes hard to distinguish between the three meanings since they often represent three stages in the life of the same object: `array` which is a package in the first meaning, as any LATEX user knows, is also a package according to the second meaning, since FDNTEX has a description of it, with dependencies, methods to build and install it, and so on; and will also be a package in the third meaning since the `rpm` flavour of the distribution will contain an `rpm` package called `array`.

FDNTEX is designed to exist under several "versions", *e.g.*, one for FreeBSD, one for Linux/RPM, one for Linux/Debian, one for HP-UX 9, and so on. Those versions will be named "flavours" in this paper, to avoid confusing the reader between "versions" of FDNTEX and versions of the packages that are part of it.

According to the context, "I" will refer either to the author, or to an hypothetical user's thoughts.

Introduction

FDNTEX is a new distribution of TEX, based upon different ideas from the previous ones.

Before teTEX appeared, a "TEX distribution" was, de facto, a pure distribution of "TEX, the program" and the strictly required files to build it and make it work in a standalone way. Any other tools, like fonts or formats or extensions, had to be installed by hand.

Since teTEX appeared, we have lived in a more user-friendly world: one can install the whole thing and obtain a rather complete TEX-based system including LATEX and lots of useful extensions.

But going on straight in the same way will lead us to a really heavy system, providing any available font, say Japanese fonts, to any user, even Russian-speaking ones. Thus distributing a hundreds-megabytes system, 85% of it being useless for each given end-user.

To avoid this problem, two ways can be studied: restrict distributions to a good subset of what is possible, and hope users will be successful in installing the missing parts by themselves; or use a different approach of the distribution problem. FDNTEX is an attempt to fulfill the second solution.

The basic ideas underlying FDNTEX design are briefly described as follow:

- Fully modular, and fine grained. *I don't want to use* `patgen`*, thus I don't install it.*
- Easy to upgrade a part without reinstalling the whole thing. *I upgrade my LATEX kernel every*

six months or so, and I don't want to change my Computer Modern fonts that often.

- Easy to install on the target system. *If I use a distribution instead of the root-source, it's just because I don't want to install a useless C compiler.*

The original idea was even simpler. We use TEX, in the leading team of FDN[1], for administrative purposes, and thus we all need to have it installed on our computers in a satisfactory way. Most of us are not good at using TEX outside of this restricted use (in fact, I'm the only one in the team who knows about the internals of TEX). teTEX didn't contain the packages we needed. Thus I started developing a TEX distribution that could satisfy those requirements. And this is also why this distribution is called "FDNTEX", originally, it was a TEX distribution to be used by FDN.

Let's have a look at some instructive examples to have a more precise view of what is standing behind those three simple ideas.

Fully modular By stating that "FDNTEX is to be modular", several problems are addressed.

The first problem has already been discussed: if I don't need `patgen`, since I don't want to generate hyphenation patterns for a new language, then I don't want it on my system. Simmilarly, if I don't use PostScript fonts at all, I don't want to have their metrics and the related software. It's useless, will slow down my system, and will obstruct the use of TEX on an old computer with a small disk.

The second problem is harder to understand. As I'm a French native, Ω is of some help to me. If I decide to use *only* Ω and Ω-based formats, then I don't want to install TEX itself, but Ω instead. It means that the minimal subset needed to start FDNTEX needn't contain TEX itself.

The third problem is that I want to be allowed to install only the minimal subset of the whole TEXware required to build the book I'm writing. That means that I don't want to install large things like "all the PS metrics" if I don't use PostScript fonts at all in my book. And, even more, if I only use Times for some titles, I don't want to install something too large on the poor old laptop that I have to use to write this book. Thus, something as large as "all the PS metrics" will not be a valid package for FDNTEX. It will have to be split into several parts, probably one per font.

All that leads us to a system with hundreds of packages. Just by splitting down the `web2c` bundle into distinct software units leads to several dozen packages. Each of the hundreds of extensions of LATEX is also an autonomous package, at least, and sometimes several, for large parts. This makes hundreds, or perhaps thousands, of packages as part of the final distribution.

One cannot afford to know all of them in enough details to be able to choose. Thus, there must be a reliable description of the dependencies between packages (who will ever remember that `tabularx` requires `array` to be installed, or that `concmath` uses `url`?), and there must be a way to choose a reasonable subset for a given use. *E.g.*, if I want to use TEX to typeset a paper about electronics, I need a way to say "everything related to electronics is of interest to me", and a second way to give more precise instructions later to add or remove packages by hand.

Easy to upgrade This point is easier to understand, and easy to automate. The only really hard thing is to handle strange cases.

Let's have a look at a hypothetical example. Let's say Mr. X wrote a few packages, `a`, `b` and `c` which are really small ones and thus are distributed as a single one. As those packages are small, and distributed as a single thing, they are in the same bundle in FDNTEX. But, a few months later, our good old Mr. X has worked a lot, and his packages have gained hundreds of features, and are now really large ones, each being composed of dozens of small independent parts. We would then need to re-bundle them separately. Thus, how to explain to the system that upgrading from the first `a&b&c` bundle means installing the three separate bundles, and that from this point each part can be upgraded separately?

Of course the symmetrical example is also troublesome. If two separate things are now unified in a single bundle, how can we explain this to the system?

Even more, one can mix the problems: three elements are replaced by two in the new version. How to upgrade easily?

The 18 months spent working on FDNTEX have not yet led to any general solution to this ugly problem.

Easy to install on the target This point is an interesting one since it's one of those which led to lots of discussions about the design of FDNTEX.

The main idea can be explained so: as I run a RedHat system, I want each FDNTEX package to be an `rpm` file, so that I can install it easily, using my usual system tools.

This means creating several flavours of the distribution, something like one per target system, and

[1] A non profit organization, which is an Internet access provider.

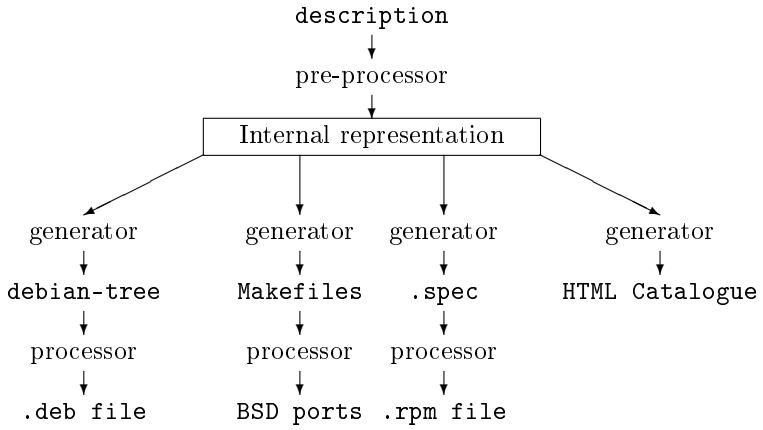


Figure 1: In this diagram, each teletype text identifies a physical representation of the information, and each roman text identifies a process to go from one representation to the other. The "Internal representation" is different since it only exists in the memory of a program, and not in a real file.

sometimes more, if several flavours of a system exist (as for Linux where Debian-based systems have almost no common point with RedHat-based ones).

Another way would be to develop a really standalone system, that would provide everything from the description language to the final binary format, including the building tools and all the other things needed. Such a system would be really easier for me (read "the distribution maker" here, instead of just "me") but would lead to something of less interest for most people. On the other hand, it would permit development of a reasonably complete distribution in a short period of time. The prototype was developed in only 6 months. And it would avoid developing all the general system that we are discussing here.

The ideas developed here are meaningful as long as there is a team working on the distribution, with people dedicated to each generator, but are meaningless if this is not the case, at least because I will never have all the variety of operating systems, and all the knowledge that is required to develop all of the flavours.

Several technical problems arise from this intention to be close to the target system; we will expose them later on, in the section "Developers' tools".

Design, implementation, tools

In this section, we will discuss the way FDNTEX is to be developed, and what will be the development tools. That is to say, not what the resulting binary packages should be, nor which packages will be part of the distribution, but *how* packages will be created for any flavour of the distribution.

The system design The prototype of FDNTEX[2] was written directly in an `rpm` representation (technically, it is a large bundle of files in `.spec` format, which is that used by `rpm`), and from that automatically translated into the right bundle of files for FreeBSD ports. This approach is of course wrong, but at least has shown that writing such a distribution is feasible.

The right choice is, of course, as shown on figure 1, to have a complete and precise description of the package and to consider its representation in a given system (say `rpm`, `swtools`, or things like that) as a projection. Thus this description has to be a superset of what can be expressed in the various target languages.

Developers' tools The tools used to create the distribution itself are clearly shown in figure 1. The first one is the pre-processor (mainly a parser) that understands the original description of the distribution. This one is, in fact, strongly linked to the language used to describe the packages.

Discussions about TPM[3] led to the conclusion that writing a brand new language from scratch seems to be a bad choice, and that it is better to use XML as the representation of the knowledge

[2] This prototype can be downloaded from `ftp://ftp.lip6.fr/pub/TeX/FDNTeX/Prototype`. It's rather old but good enough to give a more precise idea of the goal to reach.

[3] The TEX Package Manager, an idea proposed by Simon Cozens, Sebastian Rahtz and Fabrice Popineau, grew independently of FDNTEX but leads to a really similar system. It was widely discussed, first on `comp.text.tex`, and then on a separate mailing list.

database underlying the distribution. Thus, the pre-processor in figure 1 is only a suitable XML-parser with some knowledge of the system.

The most interesting points of XML for this purpose is that good XML-parsers exist on most systems, written in various languages, so that one can use a Perl-based system on Unix systems, or an anything-else-based one on MacIntoshes, without troubles related to the description of the distribution.

The various processors are, of course, based on the tools dedicated to the various flavours of the distribution, but not only that. They will include tools to automatically manage the rebuilding, so each one might look like a `Makefile` which uses the system dedicated tools.

The most complicated part is the generators, since they know about the data in the description of the package, and are able to create the projection for a given flavour. This part is hard to achieve, due to the large number of flavours.

Thus, the evidence seems to indicate that the only way to obtain good results is to have people with good knowledge of the target systems writing those generators.

Users' tools As stated before in the introduction, there will be some needs for users' tools, some being provided by the target system, like the ones to install or upgrade a package, others not, like the ones that manage configuration of the whole thing. Those tools will probably be re-used from previous distributions, like the ones used in teTEX or TEXlive.

More information The main aim of this paper is not to describe the internals of FDNTEX in a full extent, but to expose how the description language used by it was designed, and to discuss the points that have led to this design. Thus, if one wishes to have more information about other points related to FDNTEX, the reference documents, like the *FDNTEX manifest* or the current version of the DTD, can be retrieved from `ftp` sites like `ftp://ftp.lip6.fr/pub/TeX/FDNTeX`

At the time of writing, the *FDNTEX manifest* is no longer up-to-date but will probably be updated before you read it.

Describing a package

The description of a package should contain several parts.

First, its full identification, providing the version number, information about its author, a short description in various languages whenever possible, and other things of the like. All this information can be taken from the well-known Catalogue[4].

Second, information to locate it, like where it is located, in source form, on CTAN sites, or any other source-location for non-CTAN packages. This is different from the list of source files, and is to be used, *e.g.*, to show the packages in a tree for download. Other kinds of locations are of interest. A good one will be to place the package in a tree based on the functionality it provides instead of its location on CTAN, in such a tree, algebra would be a subset of mathematics, and any package related to typesetting algebra would be in this subtree, ignoring whether it's a package for plain TEX or for LATEX.

Third, its content, *i.e.*, the list of files contained in the package, and their position in a TDS conforming structure.

Fourth, information on how to build it, that is, going from the source available on the Internet to a representation that can be directly used.

Fifth, information on how it's linked to other packages, that is, dependencies.

Sixth, information on how to manage it on the target system (procedures for installation, deinstallation, upgrade, and so on).

Contents of the package The only technical choice is to decide if it only lists files relative to a supposedly well-known root, or if it lists them in a more parametric way. For example, in the first case, the font metric of `cmr10` can be listed as `texmf/fonts/tfm/public/cm/cmr10.tfm`, while in the parametric form it might be expressed as `%TEXTFMS%/%MYDIR%/cmr10.tfm` so that it can be automatically adjusted on the fly to a given target.

Since this kind of re-mapping of trees seems easy to obtain from real paths, the first choice has been retained. If one needs to re-map a TDS conforming tree to another one, it will be done in the "generator" rather than in the "pre-processor".

Building a package During the discussions about this description language, two different ways of describing the building have been studied.

The first one, the easier one from the developer point of view, and the more powerful, is to use a powerful language like a scripting one (Bourne Shell would be a perfect candidate) and to write the script in a well parameterized form, so that it can be used on a large variety of systems. This is simple, since the tools just have to use the script as is, and powerful, since one can express in this language exactly

[4] The Catalogue can be retrieved from any CTAN site or mirror at `CTAN:/help/Catalogue` and is written with XML, which will help re-using its contents.

all of what is feasible on a real system. But a burden is created for the person who writes the description of the package, and it creates a lack of portability: Bourne Shell would be of no interest on non-Unix systems. And moreover, there are large differences between different flavours of this language.

The second one, harder to achieve from the developer's point of view, and less powerful, is to express it in a high-level language. To build the `tools` bundle, it can be something like:

```
LATEX tools.ins
DTXTODVI afterpage IND GLO
DTXTODVI array IND GLO
...
```

stating that the `.ins` file to be processed to build the packages is `tools.ins`, and that each item of documentation has to be compiled twice, then has a run of `makeindex` to produce the index, then another one to produce the history of changes (a kind of glossary, technically), and another last run through LaTeX.

The second way is far better for people who have to describe a package, and is easily translated on any system by the "generator". But if one wants to have a complete enough language to handle any and every case, one will create a language as complex as a traditional scripting one. Such a complex high-level language would then be useless, since it would be too hard to understand.

The good choice is, then, to have both. If the high level language can express what is needed to build the package for most flavours, then just use it, and, if a given flavour needs to express it in a more dedicated form, then just override the first high-level description.

Moreover, if a given package is too hard to express its building in the high level language, then just use the low-level one, and discard the generic description of the building.

This model sounds good, since 95% of the packages will be described in the easy way, and only the the most problematic 5% will be described in the hard one. Thus porting the distribution to a new system will be: 1) providing the right generator, 2) porting those 5%.

Some other information will be of use to build the package, like a knowledge of what is required to be installed before starting to build anything.

An example can be "one cannot build `web2c` without `make` and a C compiler", but this is an easy one, and external[5] so that we can avoid saying it.

Another strange example is that, to build Ω one needs LaTeX, which is strange since LaTeX can be built on top of Ω. In fact, LaTeX is required only during the *building* stage, to produce the documentation in a suitable `dvi` form; but LaTeX is not required to *install* Ω on a target system.

Thus, in the dependencies section of the description, one will have to pay attention to the building stage. Of course, conflicts can arise (*e.g.*, one cannot build the documentation if a given flavour of a package is installed).

Another point is that a single "building" can produce several packages, *e.g.*, compiling the `web2c` bundle produces dozens of packages. Thus, there must be stated in some way what building is required for which package.

Links between packages As we have already seen, several kinds of links can exist between packages. The next few sections will list them, and will examine the level of complexity required.

Installation dependency This kind of link is the most evident one: listing in a package description all the packages that need to be installed for this one to work correctly.

A first hard point is to determine this list, by reading the package documentation, by reading its source, by examining its behaviour closely, and so on. It is easy to state these dependencies in the description file, even if the information is hard to obtain. Just stating that `tabularx` requires `array` is easy.

The second hard point, and the really hard one, is that not all dependencies are that easy to give. Let's study two interesting examples.

The first example is the "functionality" one.

It seems clear that `array` has no meaning if LaTeX is not installed, and that LaTeX has no meaning without TeX. Now, consider that `array` *is* of interest if Ω and Λ are installed[6]. Such a case can be solved in two ways — either put the burden on the side of `array`, by stating "this requires LaTeX or Λ to be installed" (which places burdens on lots of package descriptions), or put the burden on LaTeX and Λ to state "this provides a LaTeX-like format" (so that the `array` description may say "this requires a LaTeX-like format").

In fact, the second alternative requires that we have a list of defined *functionalities* that packages

[5] Indeed, the requirements expressed here are not in the scope of this distribution. We will summarise those requirements as "one needs a complete and working operating system to build/install FDNTEX".

[6] Ω is an evolution of TeX adding features useful for internationalisation, like Unicode; and Λ is the name of LaTeX 2ε when built on top of Ω.

may provide (or require), and to be sure to express functional dependencies only according to this list, if there is a way to do so.

On targets that already use this kind of thing, it's easy to realise the projection of this information by direct translation. On other targets, it will be a little harder — we will need to express "A or B or C or..." by listing all the alternatives for a given functionality. Fortunately the job can be done by the generator, since it has a knowledge of all the packages before it starts to generate anything.

The second example is "soft" dependencies.

Imagine a document class that has an option `psfonts` that can be called to adapt the layout to PostScript fonts instead of traditional Computer Modern ones, and another option `concrete` that uses beton instead of Computer Modern. Does this class depend on Computer Modern, beton, and PostScript fonts?

The conundrum is, that the class really depends only on Computer Modern *or* beton *or* PostScript, yet if only one of these sets of fonts is installed, the class is not fully usable. Dependency checking should not report "everything good" if in fact a part of the system cannot be used for lack of another.

We can describe this situation by introducing the concept of *soft dependencies* — that is "this package prefers this other one to be installed, but can be used without it, at your own risk".

It seems useful to define several levels of softness for this situation, *e.g.*, strong if the default behaviour, or the one the most used, needs the dependency, and weak if the dependency is needed for a weird use of the package, or by an option of little interest.

If the softness is expressed on a range from 0 to 10, 0 representing "not required at all" and 10 "strictly required" (like `array` for `tabularx`), the system could be controlled by specifying a single value n, to express "install all dependencies higher than n". The system may set a default value $n = 3$, so that the end-user can care only about the points he wants to, or drop to $n = 1$, if he doesn't want to bother at all about choosing, or raise n to 10 if he wants a *really minimal* system.

Same source bundle Two packages produced from the same source bundle, such as `patgen` and `gftopk` which are both produced from `web2c`, do have a link between them; this link has to appear clearly in the description.

One way to describe the situation is to use a purely object oriented representation. One has a source-bundle `web2c`, a building method `build-web2c` and a package `patgen`, then one states that the method `build-web2c` has to be applied to the source `web2c` in order to produce the package `patgen`[7].

A second way is to consider that the real object is the source-bundle and that the packages that are created from it are pieces of information related to it, and only to it, that is, in a structural way, the package description is a part of its parent source-bundle description.

The first technique sounds really powerful, but will quickly become hard to handle, and will create a heavy burden for a lot of people, only to handle extremely rare cases (no such cases have arisen in the 472 packages in the Prototype that would require such a complex system, even if some extremely rare ones have been met in the real world).

Thus, currently, the second technique is used.

Stage dependencies A case of "stage dependency" that was considered in the previous section is the "installation dependency". Installation dependencies are complex, and are probably the only ones that need the idea of soft dependencies.

Some other cases have to be handled.

One, also considered earlier, is the "building dependency", which states that one cannot build a given package unless another is installed. We can insist that building dependencies should all be strict, though one can find funny cases when soft dependency might be useful (the best one is METAFONT which requires X11, but can be built without windowing support). Forcing strict dependencies for the building stage is a strategic choice: FDNTEX has to behave everywhere in the same way, independently of the system on which it is built. Thus if one wants to rebuild the METAFONT binary package, X11 libraries *have to* be installed first since it have been decided that METAFONT has windowing support in FDNTEX.

Another interesting case would be to list documentation dependencies in a separate list, so that the system could ask the user "`XXX` is required in order to read the documentation of the package you're installing, do you want to install it?". This is of importance, since on a minimal system, the user can decide not to install a large set of fonts that is only required by the documentation of a tiny package.

[7] The way it's expressed is of no importance here, it can be `patgen` that states how it wants to be built, or the building method which states what it's able to build, but both methods are equivalent, given that when a generator runs, all the descriptions of all the packages have been loaded to permit consistency checking.

Conflicts The last kind of relationship between two packages is *conflict*: that is, a package cannot be installed if another one is already. The most evident case is the `tree` LaTeX-package, since three different implementations exist, have the same name (`tree.sty`), have different syntaxes and different behaviours, and installing more than one would lead to non-predictable results (one cannot say which one would be called during the building of a document). Those packages clearly conflict. (The difficulty is not in finding packages that conflict, that's rather easy, but in describing the conflict.)

The technique used in other similar languages is to have each package state which others would conflict, *e.g.*, if `a` and `b` are in conflict, then `a` states in its description "There is a conflict with `b`", and vice versa. There is a serious drawback to this technique: if a third package (say `c`) arises that conflicts with the two previous ones (this is the most probable case, if it conflicts with one, there are great probabilities it will also conflict with the other), then it has to state "There is a conflict with `a` and with `b`", and then the descriptions of `a` and `b` have to be corrected to indicate this new conflict. This is clearly troublesome, not least because `a` and `b` have to be rebuilt and to change their version numbers while their descriptions otherwise remain the same.

Another technique would be to have a separate list of mutual exclusions that is the only document that has to be corrected to handle those cases, given that the generator will easily translate the information provided by this list into the form described in the previous paragraph.

At the time of writing, the traditional way is used in FDNTEX; but we plan to switch to the other technique when a robust solution to the release number automatic increment is found.

Automatic behaviour from dependency and conflict information As already discussed, we aim to have tools, driving the system processor, that create the final package if the system is not able to do the job fully automatically.

The easy part is to use the building dependencies to automate this process. Whenever the automated system wants to build a given package, it first checks that all the building dependencies are satisfied.

The hard part may be explained with an example. Let's say that a package `a` needs the first flavour of `tree` to build its documentation, and that `b` needs the second flavour. When attempting to build `a`, the system will build and install the first flavour of `tree` if it's not already present. But, when the system tries to build `b`, it will notice the lack of the second flavour of `tree` and try to install it, which will fail due to the conflict.

The right behaviour would be to remove the conflicting flavour, then to install the required one, taking care of the dependencies while doing it (that is, remove everything that requires the conflicting flavour).

This is not linked to the way the information is provided, but to the way it is used.

Management information The management information is that which needs to be bundled with each package to allow good management of the whole system in a consistent way (like checking the dependencies, configuring the various elements, allowing one part to use another if both are installed on the target, and so on); and to allow management of the package itself (when installing, uninstalling, upgrading, and so on).

Most packages have minimal and recurrent requirements of management: rebuilding the `ls-R` database; handling the configuration files (a dozen or so cover a large majority of the packages); adding the right symbolic links at the right place; rebuilding the formats; and so on. Building stage information will be given in a generic fashion, using a high-level language, but can be given in fully user-controlled fashion too, if required.

Optimizations can almost certainly be described here too, like the fact that, if one installs 6 packages, it will probably be enough to rebuild the `ls-R` database only once, after the last package. This optimization is important, since repetitive rebuilding of the database is time consuming, and can be handled easily: the rebuilding is delayed until either the end of the installations, or an instruction that an up-to-date database is needed immediately. Then, the rebuilding will take place only when strictly needed and at the end of all the installations.

Some subtle cases can arise, because the actions to be accomplished are complex to describe. For example, in the first stages of an upgrade, the actions relate to the previous version of the package, and thus should be provided by it, and during the final stages, the actions relate to the new version. Describing the upgrade of a package requires description in each version how to install and how to un-install it for an upgrade. (These actions might be slightly different from a normal (un)installation procedure.)

Further subtleties can arise when packages evolve strangely, as in the previous case of 3 packages that are replaced by 2. We must consider how packages can describe an evolution that has not been

planned; moreover, the description of the last stages may have to care about the previously installed versions of the packages to decide how to handle the whole operation smoothly.

A good model to be reused, at least for the functionalities it provides, is that used by Debian since it takes into account all the subtleties outlined above. Please refer to the "Debian packaging manual" to have a more precise idea of what can be of interest.

A DTD to store this information

We will now introduce the first draft of the DTD that will be used by FDNTEX to store the descriptions of the packages. It's not yet, at the time of writing, used to produce any package. It will evolve quickly to a first release version, used to produce the first flavours, and then, probably, evolve again to a more mature version when new flavours appear.

Of course, any comments, improvements or suggestions are welcome, as far as they improve the DTD and make it closer to the description given in the previous sections, as this description is more mature than the DTD itself.

General structure A full document, validated according to this DTD, is a series of `Author`, `License` and `BPackage`. The description of the whole distribution can be seen, for convenience, as a single very large document, since one will need the descriptions of all the packages to generate the description in a given flavour (it's required for some flavours, like the FreeBSD one, and harmless for others, like the RPM one). Physically, there will more probably be one external document for each entity in the document: one per `Author`, per `License` and per `BPackage`, and maybe even one per `Package` (an internal element of `BPackage`).

`Author` and `License` are top-level objects just because there are relatively few of them, and there is a need for consistency, thus instead of "describing" dozens of times who is David P. Carlisle, it seems more efficient to describe him in a unique entity, and then give a reference wherever it's needed, in the description of all the packages he has written. Similarly for licenses: since there are only a dozen or so of them (according to the Catalogue), it seems useless to describe them separately for every package.

Thus, if considered as a single document, the description of the whole distribution might look like figure 2.

If we want to consider it as more modular, then we have to choose a method to aggregate all the information. One approach is to believe in XML even more strongly and use it to aggregate the whole document, using entities; this would lead us to a document which looks like the one in figure 3. An advantage is that the XML-parser can perform some consistency checking, such as one on the ids of the objects (*e.g.*, check that `Author` has been defined when referred to in a package description). The disadvantage is that the aggregating document will change each time a new package or author or license is added to the system.

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE doc SYSTEM "./fdntex.dtd">

<doc>
 <Author Id="Carlisle">
  <Name>David P. Carlisle</Name>
  <EMail>david@latex-project.org</EMail>
 </Author>

 <License Id="LPPL">
  LaTeX Project Public License
  ...
 </License>

 <BPackage>
  <BPIdentification>
   ...
  </BPIdentification>
  ...
 </BPackage>
 ...
</doc>
```

Figure 2: The whole description seen as a single file: this is the way it will be seen by the parser and by the various "generators", not the way it has to be written.

Another approach is to use a full standalone document for each part, each having its own DTD, and then to rely on the top-system to parse all the needed documents in the right order. In such a case, when the system needs a reference to the `Author`-id "Carlisle" it looks for a file named `./authors/carlisle.xml`, parses it, and then performs the consistency checking. This does not use the XML-based mechanism, and needs no top-file to handle the list of all other files, but it requires that we develop another parser on top of the XML one, and also requires more work when porting the distribution to a new target.

The third solution, while being more complex, seems the most interesting and will probably be used in the final system. In the DTD presented here, only

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE doc SYSTEM "./fdntex.dtd" [
 <!ENTITY carlisle
       SYSTEM "./authors/carlisle.xml">
 <!ENTITY bayart
       SYSTEM "./authors/bayart.xml">
 ...
 <!ENTITY lppl
       SYSTEM "./licenses/lppl.xml">
 ...
 <!ENTITY tools
       SYSTEM "./packages/tools.xml">
 ...
] >

<doc>
 &carlisle;
 &bayart;
 ...
 &lppl;
 ...
 &tools;
 ...
</doc>
```

Figure 3: A multi-part document based upon entities: this allows the XML-parser to check the ids and id-references in the whole system.

the first two alternatives (which are equivalent for the XML-parser) are available.

Authors and licenses Top-level objects `Author` and `License` are both identified by a unique id; the *id* system provided by XML is used for this purpose.

An `Author` refers to a physical person, or to a well-defined group. It might be the author of a package, or of a package description, depending on when it's referred to, or even of both. Up to now, this contains little information, the aim being not to collect personal data about people, but merely to cite them to allow anybody to contact them in case of troubles like bug reports, or license problems.

Currently, the three basic elements used here, namely `Name`, `EMail` and `License` are, in the XML typing system 'ANY', that is any non-parsed text. In the near future it will become a bit more structured with markups for the license text, and a more formal way to give an e-mail address. An example of the current structure can be seen in figure 2. The `EMail` markup in the `Author` entity is optional and might be repeated if needed.

About BPackage A `BPackage` is a bundle of packages (or big-package) which are all built from the same sources, as TEX and METAFONT are both built from the same `web2c` source-tree. `web2c` will then likely be a `BPackage`, producing several packages, including `tex` for TEX and `metafont` for METAFONT.

A `BPackage` contains a `BPIdentification` tag that identifies it in the distribution, a `SourceBundle` to help retrieve the sources, a `Building` section saying how to build the packages from it, and a non-empty list of `Packages`.

The `BPIdentification` contains a mandatory `BPName`, which is used to identify the bundle when producing a packaged version of sources in a given flavour[8], an optional version number, two textual descriptions: a `ShortDescription` (maybe several, in several languages), a `LongDescription` (may also be in several languages), a mandatory `LicenseId` and a non-empty list of `AuthorId`.

The `LicenseId` is the license under which the bundle is distributed, independently from FDNTEX. If the packages are bundled together only for convenience in the distribution, but are not when referring to the original sources, then the license will be the same as the one for the whole distribution. The license under which the description file is distributed is the one of FDNTEX, if the author of the description wants to claim that it's part of FDNTEX, and then it doesn't need to be exposed there.

The `AuthorId` can be the author of the bundle, that is the people who put all those things together, or the author of the description. There is currently no way to distinguish between them.

The descriptions are supposed to be identical, when several are provided, but in various languages. That is, if four `ShortDescriptions` are provided, they are supposed to be four times the same text, but in different languages, the language being specified as usual in an XML document, and defaulting to English, *e.g.*:

```
<ShortDescription xml:lang='fr-FR'>
 Description en français
</ShortDescription>

<ShortDescription>
 English description
</ShortDescription>
```

A bundle of sources A `SourceBundle`, as required by a `BPackage`, is a non-empty list of `SourceFile` and a `Prepare` directive that describes how to unpack all of those sources in a suitable tree for the building stage.

[8] Some flavours, like RPM and Debian, have their own source distribution format, and thus will need a name for the source package; others, like FreeBSD, give references to the real-world source, and then might not need all of that information. Since it's required by some, it's provided to all.

```
<SourceBundle>
 <SourceFile FileId="esieecv-1">
  CTAN:/macros/latex/contrib/supported/ESIEEcv.tar.gz
 </SourceFile>
 <Prepare UnpackTo="./ESIEEcv/">
  <untgz FileId="esieecv-1""/>
 </Prepare>
</SourceBundle>
```

Figure 4: Example of a `SourceBundle` tag for a simple package with sources from CTAN.

A `SourceFile` needs an id, to be referred to by the `Prepare` directive, and is supposed to be URL-like, that's either a URL, or something like `CTAN:/systems/knuth/web.tar.gz`, which is not a true URL. Several pseudo-protocols, which actually are default generic locations, will be defined. The need for two of them (`CTAN` to refer to any CTAN mirror, and `FDNTEX` to refer to any mirror of the whole distribution sources[9]) is already plain.

'Composite' generic locations may also be defined: for example, a package may be derived from `CTAN`, but a patch (maybe as simple as a Makefile) to facilitate its handling within FDNTEX may come from `FDNTEX`.

The `Prepare` directive is used for the usual sorts of files (tar, zip, and so on archives, patch files, etc.). Its variant `PrepareCust` (customize) is not yet well defined; its intent is to provide extensions for system-specific requirements, as when a particular flavour requires special treatment for unpacking the sources.

The preparation directive has an attribute that gives the place where it will unpack the sources relative to a supposed well-known root directory. *E.g.*, for the RPM flavour, the sources are supposed to be unpacked somewhere under `/usr/src/redhat/BUILD`, usually in a directory called `source` if the source archive is `source.tgz`, but maybe elsewhere if several sources are provided. In the default case, for this example, the attribute will be `"source"`. The directive is a list of actions, in the right order, that are to be accomplished to obtain a full source tree.

An example is shown at figure 4.

When retrieving the source file, the system is supposed to issue FTP commands like:

```
[ whatever is required to connect to the
  local CTAN mirror and go to the root
  of the mirror ]
cd macros/latex/contrib/supported
get ESIEEcv.tar.gz
```

This is of importance, since, when a tarball is built on the fly, like this one, it will most probably have the same structure as it has on the archive disk.

At the time of writing, several actions are defined, but others will probably be added:

untar expands a tarball, has a mandatory attribute which is the `FileId`, and an optional one named `Offset`, used if an archive has to be expanded somewhere in the tree provided by a previous tarball (like `xdvik` within `web2c`);

untgz which behaves exactly in the same way with archives compressed by `gzip` or by the traditional `compress`;

untbz which behaves in the same way with archives compressed by `bzip2`;

patch which applies a patch, as provided by a "unified diff" to a source tree; it takes at most 4 attributes: the `FileId` (mandatory), an `Offset` (optional) which allows a patch to be applied to a subtree, a `Depth` (optional) which is used as the `-p` argument to the `patch` command that is issued, and a `Compression` (optional, default to `none`) which specifies how the patch was compressed, can be `gz`, `bz2`, `Z` or `none`.

The `PrepareCust` directive will be used to provide full control to the author of a description for a given flavour of the distribution. For example, it's not clear how a `zoo` archive can be unpacked on a Windows-like system, so it would be better to provide full control to the author, instead of a too fragile action in the generic directive. A way to mix both the `Prepare` and the `PrepareCust` might be provided in a future version.

After all the actions have been accomplished, the sources have to be ready in the directory specified by `UnpackTo` relative to the well-known root defined by the flavour of the distribution.

Description of the building stage As for the preparation stage, the building stage may be written in a generic way, using pre-defined actions, in

[9] Even if `FDNTEX` will probably become equivalent to something like `CTAN:/systems/unix/fdntex`.

a `Building` directive. Otherwise (when the predefined actions are not suitable), it may be written in a system specific way with any kind of scripting language, using a `BuildingCust` directive. The kind of scripting language used will be chosen according to the target system.

There must be, at most, one `Building` directive, and there may be several `BuildingCust` ones. If so, they all have to have different targets. The `Building` directive is optional since some packages might not be built in any generic way. *E.g.*, a `dvi` viewer is strongly system-dependent and then has no generic building description, since it's not to be built on unknown targets.

Up to now, only 6 actions have been provided for building a package, but, of course others will be in the first non-alpha release of the DTD.

tex to call TEX on a file, which has two attributes, `file` (mandatory) is the name of the file to be processed, and `format` (optional, default to `plain`) is the name of the format to be used.

latex to call LATEX on a file, which has a `file` mandatory argument.

dtxtodvi provides a high level interface to build the documentations of LATEX packages (which is, *de facto*, most of the work when building the distribution). It has a mandatory `file` attribute which is supposed to be a suffix-less file name (suffix has to be `dtx`), and 5 optional attributes. `idx` (default to no) saying if there is an index to process. `glo` (default to no) saying if there is a glossary (history of changes, usually) to process. `bib` saying if a bibliography requires a BIBTEX run. Those 3 attributes can be either `yes` or `no`. `pre-runs` (default 2) says how many times LATEX has to be run before the index, glossary and bibliography are processed. `post-runs` (default 1) says how many times LATEX has to be run after that.

mktexlsr which has an optional attribute named `mandatory` which states if the action can be delayed or not, and rebuilds the `ls-R` (or equivalent on the target) database. Today, it is useless, but will be useful in a near future when the necessary actions are created to install a font while building a package. This is required for packages which are composed of a font itself and the LATEX package to handle it, since they usually have to be installed before building the documentation.

move takes two mandatory attributes, `from` and `to`, and is used to move a file or a directory from a place to another. No wildcard is allowed here. It's not to be used here for the installation of the package, but only for moving files while building the package.

cd takes a mandatory argument `to` and is used to move into the tree while building. Building is supposed to start in the directory specified in the `UnpackTo` attribute of the preparation stage.

A call to

```
<tex file="myfile.tex" format="latex"/>
```

is of course equivalent to

```
<latex file="myfile.tex"/>
```

as far as the target system handles the two following commands in the same way:

```
tex '&latex' myfile.tex
latex myfile.tex
```

which is 'not always'.

An example of such a `Building` directive is shown at figure 5.

Description of the package itself As one can guess, this is the most complex part of the description, or at least the really interesting one.

A package description is composed of 6 parts: `Identification`, `Installation`, `UnInstallation`, `FileList`, `Methods`, and `Dependencies` directives. The definition of none of these is final, but we will discuss what we believe is a good prototype.

A `Package` has a mandatory `Id` attribute that will be used when one needs to refer to it in dependencies of other packages. The `Id` should be the name of the package, but it's permissible to use anything else.

Identifying a package Just as a `BPackage` has a `BPIdentification`, a `Package` has an `Identification`, which is to be systematically used (the `BPIdentification` will only be used by some flavours of FDNTEX).

The `Identification` has to provide a `Version`, which is the one provided by the main file of the package. If there are several important files which all have their version number, then the version number provided here can be an aggregate of those, or a date. *E.g.*, if the two important files are numbered 1.2 and 5.6, then the resulting package can be numbered 1.2.5.6 or 5.6.1.2, both being valid. When a date is provided, *e.g.*, for the LATEX kernel, it should be like 20000403 to ease the comparison of two version numbers when upgrading the system.

If a version number is provided for the bundle (in the `BPIdentification` of the corresponding `BPackage`) then it's appended to the version

```
<Building>
 <latex file="ESIEEcv.ins"/>
 <dtxtodvi file="ESIEEcv" idx="yes" glo="yes" />
 <latex file="test.tex"/>
</Building>
```

Figure 5: Example of a `Building` directive for a simple LaTeX package with a full documentation including index and history of changes.

number of the package itself, so that TeX 3.14159, bundled in `web2c` 7.3a will be in a package numbered 3.14159.7.3a in the final binary version of each flavour of the distribution.

A `Release` is mandatory to indicate if the package has evolved in its FDNTeX port but not in its source version, as when a forgotten dependency is added to the description. Currently, the release is a single integer. The release number can by increased either by the author of the description, when it evolves, or by the system, in cases of automatic dependency handling (see section "Conflicts", above, for an example of such a case).

In future releases of the DTD the `Release` will probably be more informative, perhaps in a 2- or 3-integer system, like 1.0.0 for the first release, then, increasing the first one if the description of the package has evolved in an important way (*e.g.*, mending broken building directives), increasing the second digit if it evolved in a harmless way (*e.g.*, added a dependency) which means there is probably no need for upgrading, or the third digit if it evolved in a minor way (*e.g.*, to fix a typo in a `LongDescription`) in which case there is absolutely no reason to upgrade.

Textual descriptions use exactly the same structure as those for `BPackage`s, as do the `LicenseId` (which is the license under which the package itself is distributed) and `AuthorId` (which is the author of the package). Here, there is no confusion between 'author of the package' and 'author of the description': It's systematically the author of the package; the author of the description has already been cited in the identification of the bundle.

Installing a package Some target systems may not be able to deal correctly with the case where a single source-bundle provides several packages. On such systems, the `BPackage` acts as a 'virtual' package which has no real existence, and which installs no files on the system. All the related packages will require the virtual package to be built and installed (through the dependencies mechanism) before building themselves. The 'building' stage of such packages will be empty, and their 'installation' phase will install the part of the virtual package that is required.

Of course, such a subtlety need only be used when there are several `Package`s in a `BPackage`. In such cases, the building and installation stages will be system-specific only for those targets that experience difficulties, and for that class of packages.

In most cases, when the bundle has to be installed in a single run, the installation stage will be handled within the building stage, and the per-package installation stage will be empty.

1. When we build `web2c` on a system that can handle multiple packages, the building stage builds and installs the whole thing, and the installation stage does nothing.
2. When we build `web2c` on a system that cannot handle multiple packages, the building stage only builds the bundle, and the per-package installation system installs that package's part (*e.g.*, the `dvitype` binary for the `dvitype` package).
3. The `tools` bundle of packages for LaTeX can be handled in its entirety by generic directives, but is inherently a multi-package bundle; for such bundles the building stage will only build, and the per-package installation will install each package, so that the directives have to be written only once.
4. When we build a single small package (such as `ESIEEcv`) for any system, the building stage builds and the installation stage installs the package. This is the most frequent case.

The case of really complex bundles like `web2c` is handled like this because re-writing the installation stage for systems which cannot handle multi-packages is *really* hard, and error-prone. Using this method, the errors will appear only for truly minimalist systems, and not for all.

So, just like the building stage, the installation stage will use a high level language to describe things to be done, and this generic description can be overloaded when a target needs some special things to be performed that cannot be described by the generic language.

It should also be noted that the first installation — the one performed just after the building stage has been completed — is likely to differ from a 'normal' one — installing the final package on the target system. Thus, two tags are provided: `WhileBuild` to describe the installation while we are building the binary package, and `OnTarget` for the other one. Both of them have a "cust" variant, to allow overriding.

The high level language, at the time of writing, is quite poor and will evolve a lot. Some information is given as attributes of the `Installation` tag:

bindir for the directory where the binary executables have to be installed, this is supposed to be relative to the root of the target system, or at least to another root than the one used for the other directories; in fact it will often be `bin`, which will be concatenated to any prefix given while installing the real thing, *e.g.*, `/usr/local`.

libdir is the same thing for binary system libraries (mostly `libkpathsea`).

incdir is for the system include files (mostly the `.h` files related to `libkpathsea`).

docdir is the directory where the documentation for this package should stand, relative to the root-directory of the TEX system, usually something like `doc/latex/ESIEEcv` to be concatenated with *e.g.*, `/usr/local/share/texmf`, where `/usr/local` is the prefix specified during the installation and `share/texmf` is the "well-known" root.

stydir is the directory for `.sty` files provided by the package.

bstdir same thing for BibTEX styles;

bibdir for bibliographic databases;

tfmdir for `tfm` files;

mfdir for METAFONT sources;

mapdir for files related to the `map` system for PS fonts;

istdir for makeindex styles.

Instructions (defined so far) are as follows:

mkdir to create a directory, out of the ones specified previously in the attributes.

docfile is a file to be installed in `docdir`, and the same for `binfile`, `libfile`, `bstfile`, `bibfile`, `styfile`, `tfmfile`, `mffile`, `mapfile`, and `istfile`. The only one to have an attribute is `binfile`, which has an attribute `strip` which can be either `yes` or `no`, is optional, and defaults to `yes`, and says if the binary is to be stripped.

mktexlsr (see description in section "Description of the building stage", above).

format which is empty and has a mandatory attribute `name` giving the name of the format to rebuild. There is still no way to say that all the formats need to be rebuilt or that several of them have to be; this facility will be provided in future versions of the description language.

In practice, in most cases, the `WhileBuild` installation method will use those instructions, while the `OnTarget` will only state a single `mktexlsr`, since the system already takes care to move all the files listed in the `FileList` to the right place.

The default behaviour for uninstalling a package is to perform the converse of the the same actions as for installing, that is remove the file instead of installing, remove the directories if empty, and so on. Formats are also rebuilt as necessary.

The list of files The `FileList` contains a list of `docfile`, `cfgfile`, `file` and `dir`, each being part of the archive to create. Configuration files are isolated so that the uninstall and upgrade systems can handle them smoothly and save them. Directories are removed while uninstalling, if they are empty.

The dependencies The `Dependencies` tag contains a list of `BuildDep` which gives the name of a package that has to be installed in order to build this one, `Dep` which gives the name of a package that has to be installed for this one to be used properly, and `Conflict` giving the name of a package that creates a conflict if installed on the same system as this one.

The `Dep` tag has an integer attribute that gives the softness level (see section "Installation dependency" on page 164), whose default value is 10 (hard dependency). A value of 0 is legal but useless since it means no dependency at all.

A more fine-grained system will be used for future versions of the description language, at least for the conflicts since this model is really far from perfect. The new system will most probably be based on an external list of packages that are known to be in conflict. An analagous softness level will be used for conflicts, to say if it is legal to override a conflict directive or not.

A complete example

The package described here is a small one that permits the typesetting of a curriculum vitæ as French companies like to see them. It's a LATEX package, with a test file and the documentation included with the source in the `.dtx` file. The description given here has been validated against the DTD we have

just defined, but not yet used to build a real package since currently there is still no generator written.

Some parts of the document have been deleted (like the empty description of the 4 packages listed in the dependencies list).

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE doc SYSTEM "./fdntex.dtd" [
<!-- used to shorten the file name later
     since this XML document is typeset in
     two columns mode -->
<!ENTITY CTANmlcs
 "CTAN:/macros/latex/contrib/supported">
] >

<!--     maximum length of a line      -->

<doc>
 <Author Id="bayart">
  <Name>Benjamin B. Bayart</Name>
  <EMail>bayartb@edgerd.fdn.fr</EMail>
  <EMail>bayartb@guinness.domicile.fr
                               </EMail>
 </Author>
 <License Id="lppl">
  LaTeX Project Public License
 </License>
 <License Id="fdntexl">
  The FDNTeX license
 </License>

 <BPackage>
  <BPIdentification>
   <BPName>BP-ESIEEcv</BPName>
   <LicenseId Id="lppl"/>
   <AuthorId Id="bayart"/>
  </BPIdentification>

  <SourceBundle>
   <SourceFile FileId="esieecv-1">
    &CTANmlcs;/ESIEEcv.tar.gz
   </SourceFile>
   <Prepare UnpackTo="./ESIEEcv/">
    <untgz FileId="esieecv-1"
           Offset=".."/>
   </Prepare>
  </SourceBundle>

  <Building>
   <latex file="ESIEEcv.ins"/>
   <dtxtodvi file="ESIEEcv"
             idx="yes"
             glo="yes" />
   <latex file="test.tex"/>
  </Building>
```

```
<Package Name="ESIEEcv">
 <Identification>
  <Version>2.0a</Version>
  <Release>1</Release>
  <ShortDescription>
   ESIEEcv to typeset French
   curriculum vitae
  </ShortDescription>
  <ShortDescription xml:lang="fr-FR">
   ESIEEcv pour mettre en forme un
   curriculum vitae a la française
  </ShortDescription>
  <LongDescription>
   This package allows one to typeset
   a curriculum vitae as a French
   company would expect to receive it.
  </LongDescription>
  <LongDescription xml:lang="fr-FR">
   Ce package permet la mise en forme
   d'un curriculum vitae tel qu'une
   entreprise française s'attendra a
   le recevoir.
  </LongDescription>
  <LicenseId Id="lppl"/>
  <AuthorId Id="bayart"/>
 </Identification>
 <Location>
  macros/latex/contrib/supported
 </Location>
 <Location kind="func">
  lang/french
 </Location>
 <Installation
  stydir="tex/latex/ESIEEcv"
  docdir="doc/latex/ESIEEcv">
  <WhileBuild>
   <styfile>ESIEEcv.sty</styfile>
   <docfile>ESIEEcv.dvi</docfile>
   <docfile>test.tex</docfile>
   <docfile>test.dvi</docfile>
   <mktexlsr/>
  </WhileBuild>
  <OnTarget>
   <mktexlsr/>
  </OnTarget>
 </Installation>
 <FileList>
  <file>
   tex/latex/ESIEEcv/ESIEEcv.sty
  </file>
  <docfile>
   doc/latex/ESIEEcv/ESIEEcv.dvi
  </docfile>
```

```
    <docfile>
     doc/latex/ESIEEcv/test.tex
    </docfile>
    <docfile>
     doc/latex/ESIEEcv/test.dvi
    </docfile>
   </FileList>
   <Methods>
    Not yet defined
   </Methods>
   <Dependencies>
    <Dep name="a-LaTeX-format"/>
    <Dep name="tabularx"/>
    <BuildDep name="a-LaTeX-format"/>
    <BuildDep name="tabularx"/>
    <BuildDep name="babel"/>
    <BuildDep name="fnt-ec"/>
   </Dependencies>
  </Package>
 </BPackage>
</doc>
```

The full DTD

As has already been explained, this Document Type Definition (DTD) is not final. It is the DTD that this paper has described, and it has been used to validate the description. A more up to date version might be available at `ftp://ftp.fdn.fr/pub/FDNTeX/Develop/fdntex.dtd`, and a more up-to-date version of this paper (or at least something describing the corresponding version of the DTD) should be available at the same place.

```
<!ELEMENT doc (Author|BPackage|License)*>

<!ELEMENT Author (Name,EMail*)>
<!ATTLIST Author Id ID #REQUIRED>

<!ELEMENT Name ANY>
<!ELEMENT EMail ANY>

<!ELEMENT License ANY>
<!ATTLIST License
        Id ID #REQUIRED
        xml:lang NMTOKEN 'en'>

<!ELEMENT BPackage (BPIdentification,
                    SourceBundle,
                    Building,
                    BuildingCust*,
                    Package+)>

<!ELEMENT BPIdentification
        (BPName,
         Version?,
         ShortDescription*,
         LongDescription*,
         LicenseId,
         AuthorId+)>

<!ELEMENT BPName ANY>
<!ELEMENT Version ANY>
<!-- ShortDescription and LongDescription
     are defined later on,
     when defining Package -->
<!ELEMENT LicenseId EMPTY>
<!ATTLIST LicenseId Id IDREF #REQUIRED>
<!ELEMENT AuthorId EMPTY>
<!ATTLIST AuthorId Id IDREF #REQUIRED>

<!--
 Identification
  BPName
  Version?
  ShortDescription
  LongDescription
  License
  AuthorId
-->

<!ELEMENT SourceBundle
        (SourceFile+,
         (Prepare|PrepareCust))>
<!ELEMENT SourceFile ANY>
<!ATTLIST SourceFile
        FileId ID #REQUIRED>
<!ELEMENT Prepare
        (untar|
         untgz|
         patch)+>
<!ATTLIST Prepare
        UnpackTo CDATA #REQUIRED>
<!ELEMENT untar EMPTY>
<!ATTLIST untar
        Offset CDATA "."
        FileId IDREF #REQUIRED>
<!ELEMENT untgz EMPTY>
<!ATTLIST untgz
        Offset CDATA "."
        FileId IDREF #REQUIRED>
<!ELEMENT patch EMPTY>
<!ATTLIST patch
        Offset CDATA "."
        Depth CDATA "1"
        FileId IDREF #REQUIRED
        Compression (gz|bz2|Z|none)
                              "none" >
<!ELEMENT PrepareCust ANY>
```

```
<!ATTLIST PrepareCust
        UnpackTo CDATA #REQUIRED
        >



<!--
 SourceBundle
  SourceFile*
  Prepare | PrepareCust
   (UnpackTo?) implied
-->
<!ELEMENT Building
        (tex|
         latex|
         dtxtodvi|
         mktexlsr|
         move|
         cd)+>
<!ELEMENT tex EMPTY>
<!ATTLIST tex
        file CDATA #REQUIRED
        format CDATA "plain">
<!ELEMENT latex EMPTY>
<!ATTLIST latex
        file CDATA #REQUIRED>
<!ELEMENT dtxtodvi EMPTY>
<!ATTLIST dtxtodvi
        file CDATA #REQUIRED
        idx (yes|no) "no"
        glo (yes|no) "no"
        bib (yes|no) "no"
        pre-runs CDATA "2"
        post-runs CDATA "1">
<!-- mktexlsr will be defined later -->
<!ELEMENT move EMPTY>
<!ATTLIST move
        from CDATA #REQUIRED
        to CDATA #REQUIRED>
<!ELEMENT cd EMPTY>
<!ATTLIST cd
        to CDATA #REQUIRED>

<!ELEMENT BuildingCust ANY>
<!ATTLIST BuildingCust
        Target (i386|ppc|sparc|alpha)
                                #REQUIRED
        System (linux|freebsd|solaris|
                hpux9|hpux10)   #REQUIRED>

<!ELEMENT Package (
        Identification,
        Location+,
        Installation,
        UnInstallation?,
        FileList,
        Methods,
        Dependencies?,
        Provides?)>
<!ATTLIST Package
        Name ID #REQUIRED>

<!ELEMENT Identification (
        Version,
        Release,
        ShortDescription+,
        LongDescription+,
        LicenseId,
        AuthorId+)>
<!ELEMENT Release ANY>
<!ELEMENT ShortDescription (#PCDATA)>
<!ATTLIST ShortDescription
        xml:lang NMTOKEN 'en'>
<!ELEMENT LongDescription (#PCDATA)>
<!ATTLIST LongDescription
        xml:lang NMTOKEN 'en'>
<!-- LicenseId and AuthorId are
     already defined -->

<!ELEMENT Location (#PCDATA)>
<!ATTLIST Location
        kind (ctan|func) "ctan">

<!ELEMENT Installation
        (WhileBuild,
         WhileBuiltCust*,
         OnTarget,
         OnTargetCust*)>
<!ATTLIST Installation
        bindir CDATA "."
        libdir CDATA "."
        incdir CDATA "."
        docdir CDATA "."
        stydir CDATA "."
        bstdir CDATA "."
        bibdir CDATA "."
        tfmdir CDATA "."
        mfdir  CDATA "."
        mapdir CDATA "."
        istdir CDATA ".">
<!ELEMENT WhileBuild
        (mkdir|
         docfile|
         binfile|
         libfile|
         bstfile|
         bibfile|
         styfile|
```

```
         tfmdile|
         mffile|
         mapfile|
         istfile|
         mktexlsr|
         format)+>
<!ELEMENT WhileBuildCust ANY>
<!ELEMENT OnTarget
        (mkdir|
         docfile|
         binfile|
         libfile|
         bstfile|
         bibfile|
         styfile|
         tfmdile|
         mffile|
         mapfile|
         istfile|
         mktexlsr|
         format)+>
<!ELEMENT OnTargetCust ANY>

<!ELEMENT mkdir ANY>
<!ELEMENT docfile ANY>
<!ELEMENT binfile ANY>
<!ELEMENT libfile ANY>
<!ELEMENT bstfile ANY>
<!ELEMENT bibfile ANY>
<!ELEMENT styfile ANY>
<!ELEMENT tfmfile ANY>
<!ELEMENT mffile  ANY>
<!ELEMENT mapfile ANY>
<!ELEMENT istfile ANY>
<!ELEMENT mktexlsr EMPTY>
<!ATTLIST mktexlsr
        mandatory (yex|no) "no">
<!ELEMENT format EMPTY>
<!ATTLIST format
        name CDATA #REQUIRED>
<!ELEMENT UnInstallation
        (WhileBuild,
         WhileBuiltCust*,
         OnTarget,
         OnTargetCust*)>

<!ELEMENT FileList
        (docfile|cfgfile|file|dir)*>
<!ELEMENT cfgfile ANY>
<!ELEMENT file ANY>
<!ELEMENT dir ANY>

<!ELEMENT Methodes ANY>
```

```
<!ELEMENT Dependencies
        (BuildDep|
         Dep|
         Conflict)+>
<!ELEMENT BuildDep EMPTY>
<!ATTLIST BuildDep
        name IDREF #REQUIRED>
<!ELEMENT Dep EMPTY>
<!ATTLIST Dep
        name IDREF #REQUIRED
        level CDATA "10">
<!ELEMENT Conflict EMPTY>
<!ATTLIST Conflict
        name IDREF #REQUIRED>

<!ELEMENT Methodes EMPTY>
<!--
  FileList
  Methods
-->

<!ELEMENT Provides EMPTY>
<!ATTLIST Provides
        Name ID #REQUIRED>
```

Acknowledgments

FDNTEX, and thus this paper, is not the pure outcome of my ideas of what should constitute a better distribution of TEX, it's also the result of long talks with friends of mine, like Rémy Card and Olivier Gutknecht; and with people more involved in the TEX world and in writing distributions of TEX like Sebastian Rahtz who suggested the use of XML as a description language, Simon Cozens who gave me the idea of a high level language, Fabrice Popineau who worked with S. Rahtz on the system used in TEXlive, Taco Hoekwater who proposed great ideas for dependencies that have not been matured enough (at least by me) to be discussed in this paper, Robin Fairbairns who gave us good advice during the discussions about TPM, and probably others that I forgot.

Special thanks to Sylvia Pédron who spent a lot of hours helping me to improve my English and proof reading this paper.

Unicode and math, a combination whose time has come — Finally!

Barbara Beeton
American Mathematical Society
bnb@ams.org

Abstract

To technical publishers looking at ways to provide mathematical content in electronic form (Web pages, e-books, etc.), fonts are seen as an "f-word". Without an adequate complement of symbols and alphabetic type styles available for direct presentation of mathematical expressions, the possibilities are limited to such workarounds as `.gif` and `.pdf` files, either of which limits the flexibility of presentation.

The STIX project (Scientific and Technical Information eXchange), representing a consortium of scientific and technical publishers and scientific societies, has been trying to do something about filling this gap. Starting with a comprehensive list of symbols used in technical publishing, drawn from the fonts of consortium members and from other sources like the public entity sets for SGML as listed in the ISO Technical Report 9573-13, a proposal was made to the Unicode Technical Committee to add more math symbols and variant alphabets to Unicode. Negotiations have been underway since mid-1997 (the wheels of standards organizations grind exceedingly slowly), but things are beginning to happen.

This paper will share the latest information on the progress of additional math symbols in Unicode, and the plans for making fonts of these symbols freely available to anyone who needs them.

Introduction

The composition of mathematics has never been straightforward; it has always required special fonts above and beyond the alphabetic complement required for text. Even if an author makes an effort to describe mathematical concepts and relationships in words, there comes a point where symbols become necessary for both clarity and conciseness. In some fields (for example, symbolic logic), the use of notation has expanded to such a degree that it is nearly impossible to express concepts clearly in ordinary words; symbols convey the desired meaning much more directly. The situation might be compared to that of two literate Chinese from different areas meeting, and communicating by writing rather than in their different spoken dialects. There is sometimes just no reasonable substitute for a common writing system.

Although symbols form a large and important part of written mathematics, mainly indicating operations, relations, and other similar concepts, alphabets are also co-opted from their role of representing ordinary language to provide the notation for mathematical constants, variables and functions — the things operated on. The number of different alphabets used in some documents appears to be limited only by what is available or by the capacity of the typesetting system (manual, mechanized or electronic). Only numerals seem to denote more or less the same kinds of concepts in both ordinary prose and mathematical notation. Needless to say, font foundries have never been overly eager to provide an unlimited supply of new symbol shapes of arcane design and often intricate production requirements.

Complicating this situation is the fact that the audience for typeset mathematics is relatively small. If the number of mathematicians clamoring for competently printed material in their subject were anywhere near the number of readers of novels or sports magazines, or if these mathematicians had budgets matching those of major advertising agencies, font foundries could muster much greater interest in doing this sort of work.

Terminology

When one looks at a printed page, one sees that it is constructed from many small elements. There are letters, digits, punctuation, symbols, dingbats, ...

One might think to refer to all of these as *characters*. The dictionary [9] definition of *character* is, in part,

> **character** ... **1**. A sign or token placed upon an object as an indication of some special fact, as ownership or origin; a mark, brand, or stamp. **2**. Hence: **a** a graphic symbol of any sort; esp., a graphic symbol employed in recording language, as a letter. **b** Writing; printing. **c** ...

Clear enough? Well, not quite.

In standardese, a term can have only one meaning. The basic ISO[1] definition [5] is

> **character** A member of a set of elements used for the organisation, control, or representation of data.

Thus the term *character* cannot be used in an ISO standard with any other meaning.

Another relevant term is *code*; from the same dictionary [9]:

> **code** ... **3**. A system of signals for communication by telegraph, flags, etc. (...); also, a system of words or other symbols arbitrarily used to represent words; as, a secret *code*.

This is the term adopted to identify the system by which data is stored in a computer memory, and the individual elements are known as *coded characters*, or *characters* for short. Different computer coding systems use different bit patterns to represent the same character; for example, the letter A would have a different code in ASCII, BCD, EBCDIC, ISO 646, ISO 8859-1, etc., but in each of these systems, A is still considered the same character. If it is in a context that might (in print) be represented in italic or boldface, that makes no difference; the same code is used for all.

But an A in a font or on a printed page is not (by this system) a character, and an italic A is different from a boldface A, and so on. The term adopted in standardese [3] for such an element is *glyph*:

> **glyph** A recognizable abstract graphic symbol which is independent of any specific design.

Thus it is clear that one code may represent many different glyphs. The reverse is also true: while the word "file" is spelled with four letters, and coded as four characters, when printed with a font that has ligatures, only three glyphs are used.

The association between characters and glyphs is referred to as *mapping*. What is important here is that in neither direction is the mapping between a coded character and a glyph one-to-one; it may be one-to-many, or many-to-one. While this is not only adequate, but even admirable, when dealing with text, for mathematics it can introduce serious ambiguity.

[1] International Organization for Standardization

Codes

The alphanumeric soup of standardized codes has already been mentioned. Consider the history of codes used for computer input.

Although quite a few different models of digital computer architecture have been devised, very few have been based on a wider range of possibilities for the smallest element other than zero and one — on and off. (This provides the rationale for naming the bit, "binary digit".) Different combinations of bits, in strings of predefined length, designate characters. The number of bits in such a string is the limiting factor in how many characters can comprise a code.

Very early codes contained six bits — 64 characters, just enough for a single-case (latin) alphabet, ten digits, five arithmetical operators (`+`, `-`, `*`, `/` and `=`), the punctuation required to format real numbers and accounting data, and a number of *control codes* to support interaction with a Teletype machine. The following symbol complement, a variant of the BCD code, was available on a punched paper tape device used in the 1970s at AMS; symbols in the second row had to be preceded by an upshift and followed by a downshift, as they were piggy-backed onto other characters.

```
. , - / * # & $
: ; + = @ ( ) < > ' "
```

This was sufficient to support Fortran, Cobol, and other antique programming languages, but not a direct visual representation of mathematical expressions.

ASCII, in its original form, had seven bits and 128 characters, which could accommodate a lowercase alphabet and more symbols. (This is the code under which TEX was first implemented.) ISO 646 is the "international" version of ASCII. A key principle was — and is — that once a character gets into a code, it is never removed, so the current 8-bit ASCII is backward compatible with the 7-bit version, at least insofar as what can be encoded.

Other codes have been promulgated by manufacturers, national standards bodies, and the ISO. Until the mid-1980s, these codes were used almost exclusively to support processing of programming languages and natural languages. Whatever symbols were included were necessary to specify programming operations, not the symbolic representation of scientific disciplines, and typically, except for

the few symbols and punctuation characters that were already present in six-bit codes, symbolic characters were typically segregated in programming language–specific codes such as the one for APL.

Some language codes already exceeded the typical eight-bit capacity of 256 elements. It is impossible, for example, to fit in all the accented and variant letters of the alphabet needed to represent all the languages based on the latin alphabet. And codes for Japanese and Chinese had to accommodate the nearly 10,000 characters used to publish newspapers, or, preferably, the 50,000 characters or more found in literary works.

The development of a multi-byte code, ISO 10646 (originally a two-byte code), undertook to combine in a single code all existing national and commercial codes. Computer manufacturers and other commercial organizations dependent on computer technology became dissatisfied with the progress of the ISO working group responsible for standardizing codes, and, in 1988, formed the Unicode Consortium for the purpose of creating a unified international code standard on which new multinational computer technology could be based. The ISO old guard was joined or replaced by the Unicode members, and since 1991 Unicode and ISO 10646 have been parallel.

The content and structure of Unicode

In the Unicode 2.0 manual [7], the section *Design goals* identifies some of the gaps in coverage by existing codes.

> When the Unicode project began in 1988, groups most affected by the lack of a consistent international character standard included the publishers of scientific and mathematical software, newspaper and book publishers, bibliographic information services, and academic researchers. ... The explosive growth of the Internet has added to the demand for a character set standard that can be used all over the world.

The first iteration of Unicode included "characters from all major international standards published before December 31, 1990". One of these was the SGML standard [2], which contained a sizeable list of mathematical and technical symbols in its original Annex A (this list was later moved to a technical report [4]). Other sources included "bibliographic standards used in libraries ..., the most prominent national standards, and various industry standards in very common use".

In the Unicode 3.0 manual [8], only one reference can be unambiguously associated with math symbols: ISO 6862, *Information and documentation — Mathematics character set for bibliographic information interchange* (no explicit references are shown in the Unicode 2.0 manual). Many of the symbols listed in the annex to the SGML standard don't appear in Unicode. More about this later.

There are several design principles especially relevant to the designation of math symbols as characters [8]:

- The Unicode Standard encodes characters, not glyphs.
- Characters have well-defined semantics.
- The Unicode Standard encodes plain text.

The implication is that the meaning of each character is distinct, so that the representation when interchanged or typeset will be unambiguous. More about this later as well.

Unicode is organized into segments of 65,536 characters called *planes*. The first of these, plane 0, is the *basic multilingual plane* (BMP). Within this plane, characters with common characteristics are grouped into blocks, usually of 256 characters. The first full block is equivalent to *Latin 1*, with the first half comprising 7-bit ASCII. The code for any character assigned to the BMP can be represented by 16 bits, a two-byte, or *two-octet* code. The formal representation of such a code is "U+*xxxx*", where *xxxx* is a string of four hexadecimal digits.

Within the BMP, these blocks are occupied by symbols:

- U+2000–206F: General punctuation
- U+2070–209F: Subscripts and superscripts
- U+20A0–20CF: Currency symbols
- U+20D0–20FF: Combining diacritical marks for symbols
- U+2100–214F: Letterlike symbols
- U+2150–218F: Number forms
- U+2190–21FF: Arrows
- U+2200–22FF: Mathematical Operators
- U+2300–23FF: Miscellaneous technical (in Unicode 2.0, U+2380–23FF are unassigned, reserved for later additions)
- U+2400–243F: Control pictures
- U+2440–245F: Optical Character Recognition
- U+2460–24FF: Enclosed alphanumerics
- U+2500–257F: Box drawing
- U+2580–259F: Block elements
- U+25A0–25FF: Geometric shapes
- U+2600–267F: Miscellaneous symbols

- U+2700–27BF: Dingbats
- U+27C0–27FF: (unassigned)
- U+2800–28FF: Braille patterns (added in Unicode 3.0)
- U+2900–2DFF: (unassigned)

Symbols that were part of earlier codes are kept with those codes in other blocks; if a code already existed, the character was not duplicated.

A segment of the BMP has been set aside for *private use*, where characters may be assigned which are not formally included in Unicode but for which an agreement exists between sending and receiving users.

Up to now, most character assignments are in the BMP, with the intention that they be easily accessed. However, for less frequently occurring characters, work has begun to populate Plane 1. This is another area with relevance that will become obvious later.

Identifying symbols required for math typesetting

Early in 1997, a group of scientific and technical societies and publishers banded together under the name STIPub — Scientific and Technical Information Publishers — to address matters of common interest. The founding members of this group were

- American Chemical Society
- American Mathematical Society
- American Institute of Physics
- American Physical Society
- Elsevier Science, Inc.
- Institute of Electrical and Electronic Engineers

One topic of growing concern to the STIPub members was how best to move into the Internet age, to make use of the World Wide Web as an adjunct, if not the new centerpiece, of their publishing efforts. A major obstacle facing Web publication was — and is — the pitifully inadequate symbol set available with HTML, and its lack of support for the two-dimensional positioning of mathematical notation. Several attempts at providing some support for this material had been brushed aside as successive releases of the HTML Recommendation[2] added features to improve the visual presentation and control of document layout.

It was understood that a future version of most browsers would include "Unicode support". Although it is unclear exactly what is meant by this, an obvious course of action was to make certain that Unicode coverage for math and technical notation is complete.

A working group for Scientific and Technical Information eXchange (STIX) was formed with the charter to identify the required symbol complement and get the missing elements incorporated into Unicode. A first step was to collect from the STIX participants and other sources lists of symbols currently in use, and to reduce this to two subcollections: symbols already in Unicode and symbols not in Unicode. Information was gathered from the following sources, in addition to the STIX members:

- the entity sets of ISO TR 9573-13 [4]; electronic files were provided by the editor, Anders Berglund.
- fonts designed to be used with TEX: Computer Modern, AMSFonts, Lucida New Math, `lasysym`, St. Mary Road, `wasysym`
- Wolfram Research (Mathematica)
- Justin Ziegler's LATEX 3 project report [10]
- Taco Hoekwater (for Kluwer Academic Publishers)
- Jörg Knappen (for Springer Verlag)
- Paul Topping, Design Science, Inc. (MathType)
- the ISO Z language standard (ISO CD 13568)
- various requests for specific symbols identified through AMS technical support and the newsgroup `comp.text.tex`

More than 2200 distinct symbols were identified.

The next step was to determine which were already included in Unicode. As already mentioned, not all symbols are located in the blocks designated for symbols; some have codes in other blocks, including Latin 1, Greek, and even among the CJK[3] symbols and punctuation. About half of the symbols in the collection were found in the Unicode 2.0 manual [7]; the remainder were assigned provisional identifiers in the Unicode private use area, and a table was constructed, listing the following for each symbol:

- its ID
- a possible cross-reference to an ID for another symbol of similar shape or meaning
- the AFII[4] glyph identifier
- the entity name, TEX code, or other identifying information for each contributor
- a brief description

[2] A Recommendation is the World Wide Web Consortium's (W3C) equivalent of an international standard.

[3] Chinese, Japanese and Korean

[4] Association for Font Information Interchange

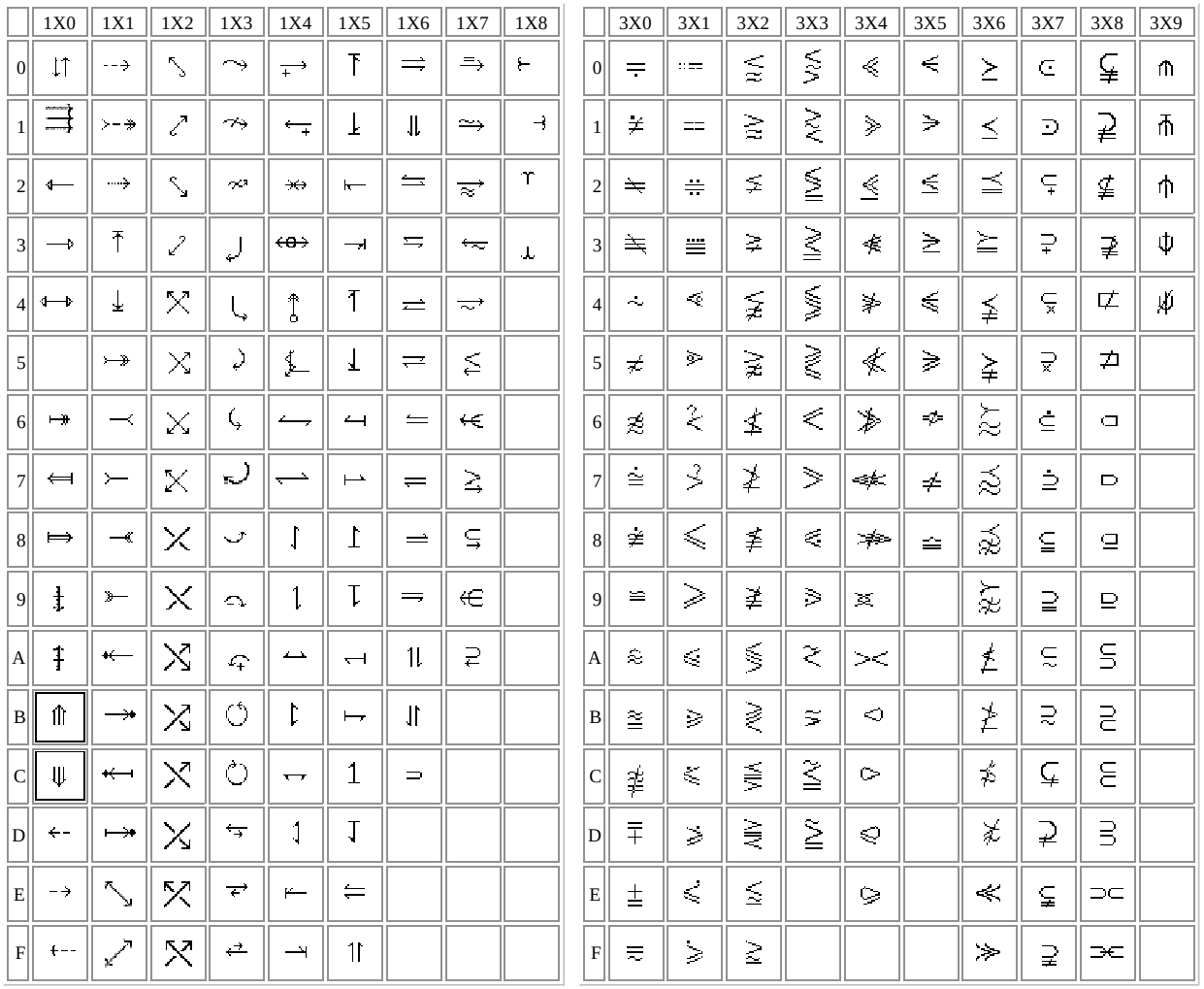


Figure 1: First (arrows) and third (binary relations) of seven symbol tables in the December 1998 version of the Unicode math proposal

This completed the first half of the project. The second, more difficult, half remained — putting the request into a form that would be acceptable to the Unicode Technical Committee (UTC).

Preparing the Unicode proposal

It is a UTC requirement that any request for encoding must be accompanied by a sample image of the requested character. This posed some problems. For many of the symbols to be submitted, no fonts were available. We solved that problem in a rudimentary way by creating GIF images at a very low resolution; for symbols we did have in fonts, this was accomplished using latex2html, and for others, bitmaps were created by hand and packaged as GIFs. The resulting images, packaged in HTML tables, were rough but recognizable. For the initial version of the proposal, the order of symbols in the tables was the same as that in our master list, by reference ID; the full collection of tables and text — about 70 files in all — was sent to the UTC in March 1998.

The arrangement of symbols was semi-random, and individual symbols were hard to find, so at the UTC's request, the proposal was reorganized and the first revision delivered in December 1998. Figure 1 shows two of the seven tables in this first revision.

Once the symbols were rearranged, the presence of similar shapes — considered by the UTC to be possible duplicates — became obvious.

The UTC takes its job seriously [8, p. 17]:

> The Unicode Standard avoids duplicate encoding of characters by unifying them within scripts across languages; characters that are equivalent in form are given a single code.

> Common letters, punctuation marks, symbols, and diacritics are given one code each, regardless of language, ...

With respect to math symbols, this means that if two symbols look very much alike, unless there is very strong documentation to support the contention that they have different meanings, only one will be assigned a code. Thus, for example, $\leq$ and $\leqq$ might be considered equivalent — if they had not already both been accepted into Unicode. Some UTC members feel that the original inclusion of such pairs was a mistake, and they are determined not to repeat it. Knowledge of that fact guided the organization of the math proposal, and helped to determine what kind of documentation would be needed.

The first rearrangement of the symbols was into groups that roughly coincided with existing Unicode blocks: arrows, "traditional" math symbols, geometric shapes, etc. The "traditional" symbols were classified further into groups that corresponded to their functions: large operators, binary operators, binary relations, delimiters, etc. After some preliminary discussions with a member of the UTC, we decided to structure our proposal in blocks corresponding to these functional groups, with the symbols arranged in the same general order as similar ones already present in Unicode.

We were also advised to eliminate any "duplicates" of existing characters, but since slight variations do often have different meanings in mathematical exposition, we decided to keep everything for the first round, and refine on the basis of specific directives from the UTC. However, we did at this stage identify symbols with similar shapes, and possibly equivalent meanings, and flagged them in our master list to indicate the need for additional documentation.

The UTC requested at the outset that symbols used not in math, but in fields such as chemistry, astronomy, engineering and phonetics, be omitted, to be requested separately by representatives of those disciplines. This change was made, eliminating more than a hundred symbols.

In the first round of the proposal, we included three alphabets — blackboard bold, script and fraktur — as well as a list of alphabets required for mathematical exposition. Some additional alphabetic inclusions were letters that occur in an unusual orientation (e.g., $\exists$) and variant forms of several Greek letters (including ϵ, the straight-backed epsilon) which were missing from the Unicode complement. Although we felt that the case for including these alphabets was strong, opposition from members of the UTC was stronger; in the next iteration, the alphabets were removed to a separate proposal.

Refining the proposal

Adjusting the content of the list of symbols to make it acceptable to the UTC took several iterations over the course of two years. Attendance at the meetings where the proposal was discussed proved to be essential, given the scope of the project. Two UTC members were assigned to help refine the proposal, cast it in the form required by the ISO working group (WG2) in charge of character coding standards, and prepare the text that will appear in the published Unicode manual.

The first task was to overcome the reluctance of some members of the UTC to believe that there could actually be more than a thousand math and technical symbols not already in Unicode. The reorganized proposal, generally following the ordering of symbols already present, clarified the situation by making it relatively easy to compare the new material to the existing Unicode.

Quite a few symbols consisted of a base symbol plus a cancellation. This could be a long or short slash or vertical stroke, a backwards slash, or a double vertical stroke. Although two lengths of the forward slash and vertical stroke appear in Unicode among the combining diacritics, it was decided that the longer versions would be designated as the proper cancellation markers for math, leaving the actual shape of the cancelled symbol as a font issue.

Special attention was paid to symbols that look similar, that differ for example in the number

Several decisions were made in order to mimimize the number of codes to be assigned. These were the most important:

- Any symbols cancelled by a vertical or slanted stroke should be constructed from a base symbol and a combining diacritic; this eliminated the entire cancelled alphabet used by physicists.
- A "variant selector" (VS) would be provided to allow for shape variants that ordinarily represent personal preference or house style and not differences in meaning. Except where needed to provide a base character for cancellation, in which case a code would be assigned, only the most common variant of such a symbol would be assigned, and the specified variant represented by the code for the base symbol plus the VS. A list of such variants appears in figure 2.

Documentation for the symbols that were most likely to be controversial was sought in published material. A request for citations was presented on

Symbol variants defined using a Variation Selector (VS)

Barbara Beeton, for STIPUB
7 February 2000

- 2268 ≨ + VS → ≦⃥ less-than and not double equal - with vertical stroke
- 2269 ≩ + VS → ≧⃥ greater-than and not double equal - with vertical stroke

- 22DA ⋚ + VS → ⪋ less-than above slanted equal above greater-than
- 22DB ⋛ + VS → ⪌ greater-than above slanted equal above less-than
- 2272 ≲ + VS → ⩽ less-than or similar - following the slant of the lower leg
- 2273 ≳ + VS → ⩾ greater-than or similar - following the slant of the lower leg
- 2A9D ⪝ + VS → ⪡ similar - following the slant of the upper leg - or less-than
- 2A9E ⪞ + VS → ⪢ similar - following the slant of the upper leg - or greater-than
- 2AAC ⪬ + VS → ⪯ smaller than or slanted equal
- 2AAD ⪭ + VS → ⪰ larger than or slanted equal

- 228A ⊊ + VS → ⫋ subset not equals - variant with stroke through bottom members
- 228B ⊋ + VS → ⫌ superset not equals - variant with stroke through bottom members
- 2ACB ⫋ + VS → ⫋ subset not two-line equals - variant with stroke through bottom members
- 2ACC ⫌ + VS → ⫌ superset not two-line equals - variant with stroke through bottom members

- 2A3B ⨻ + VS → ⨽ interior product - tall variant with narrow foot
- 2A3C ⨼ + VS → ⨾ righthand interior product - tall variant with narrow foot

- 2295 ⊕ + VS → ⊕ circled plus with white rim
- 2297 ⊗ + VS → ⊗ circled times with white rim
- 229C ⊜ + VS → ⊜ equal sign inside and touching a circle

- 2225 ∥ + VS → // slanted parallel
- 2225 ∥ + VS + 20E5 \ → ⫽ slanted parallel with reverse slash

** • 222A ∪ + VS → ∪ union with serifs
** • 2229 ∩ + VS → ∩ intersection with serifs
** • 2293 ⊓ + VS → ⊓ square intersection with serifs
** • 2294 ⊔ + VS → ⊔ square union with serifs

Notes:

** The shape is incorrect, owing to unavailability of a suitable font; the correct shape will be provided as soon as possible. The associated text correctly describes the desired shape.

Figure 2: Symbols constructed using the Variant Selector

the AMS Web site during the summer and early fall of 1998. Although the response was not as great as hoped for, some useful references were obtained. The ideal citation contained several similar-appearing symbols on a single page, in context, preferably with definitions of one or more of the symbols in the text. More than a hundred pages of such examples were copied, annotated, assigned reference IDs, indexed, and provided to the two UTC members responsible for advancing the symbols proposal. This mass of data proved its worth more than once, when it was possible to cite a particular sample in answer to a challenge.

Late in 1999, the content of the symbols proposal was agreed, codes were assigned, and a final version of the proposal was prepared for the spring 2000 meeting of WG2. The proposal forwarded to WG2 places material into the following blocks:

- U+2000–206F: General punctuation (9 new codes)
- U+20D0–20FF: Combining diacritical marks for symbols (4 codes)
- U+2100–214F: Letterlike symbols (16 codes)
- U+2190–21FF: Arrows (12 codes)
- U+2200–22FF: Mathematical operators (14 codes)
- U+2300–23FF: Miscellaneous technical (26 codes)
- U+2400–243F: Control pictures
- U+25A0–25FF: Geometric shapes (8 codes)
- U+2900–297F: Supplemental arrows (new; 128 codes)
- U+2980–29FF: Miscellaneous math symbols (new; 114 codes)
- U+2A00–2AFF: Supplemental math operators (new; 246 codes)

In addition, a few characters were added to other areas; in all, 584 new codes have been assigned.

After much discussion, the proposition was reluctantly accepted that the same letter from different alphabets has different meanings within a single document, and thus these different alphabets deserve to be coded *for use only in mathematical notation*. The example used to clinch the argument was the contrast between these two formulas:

$$\mathcal{H} = \int d\tau(\varepsilon E^2 + \mu H^2)$$

$$H = \int d\tau(\varepsilon E^2 + \mu H^2)$$

The first is the Hamiltonian formula well known in physics; the second is an unremarkable integral equation.

These alphabets are needed for proper composition of mathematics:

- lightface upright Latin, Greek and digits
- boldface upright Latin, Greek and digits
- lightface italic Latin, Greek and digits
- boldface italic Latin, Greek and digits
- script
- fraktur
- bold fraktur
- open-face (blackboard bold) including digits
- lightface upright sans serif Latin and digits
- lightface italic sans serif Latin
- boldface upright sans serif Latin, Greek, and digits
- boldface italic sans serif Latin and Greek
- monospace Latin and digits

Except for the lightface upright letters and digits, which are to be encoded using the base Unicodes (ASCII for the Latin letters and digits), the alphanumerics are to be placed in a tightly packed block (U+D400–D7FF) in plane 1, so that they can be used for math (most likely via entity names in MathML), but will be very difficult to access for other purposes.

The math alphanumerics block has been incorporated into a larger proposal for plane 1, and its schedule is slightly behind that of the symbols proposal. A "final" version is now in preparation, and will be forwarded to WG2 for their fall meeting.

Assuming that the required three ISO ballots are favorable, the new codes should be a formal part of Unicode and ISO 10646 by

Commissioning a font

Late in 1999, even before the fate of the Unicode math proposals was known, STIPub issued a Request for Proposal to a number of font suppliers. This RFP requested bids for creating a set of fonts compatible with Times and incorporating all the symbols and alphabets identified by the STIX project, suitable both for use in Web browsers and in print. The resulting font set is to be "made available for general use under license, but free of charge, with the aim of easing and fostering the uninhibited flow, exchange, and linking of scientific information." [6]

Proposals were received from four potential suppliers, and comments from a fifth, which refrained from proposing because of a prior commitments. As of this writing, a probable supplier has been chosen, and negotiations are proceeding toward a contract.

More details will be available on this topic at the time of the Oxford TUG meeting.

Remaining

The Unicode manual contains extensive text describing the proper use of the character codes, as a guide to programmers. Particular attention is paid to processing of context dependencies, combining codes and the like. Since mathematical notation will be realized in a document as a combination of coding and markup, and not all mathematical symbols are interpreted in the same way, instructions are needed. The creation of a technical report is an open action item on the Unicode docket; the text of this report will ultimately be incorporated into the Unicode manual.

The symbols from chemistry, astronomy, engineering and phonetics that were excluded by request of the UTC have been left for consideration by the organizations that submitted them.

Mathematical notation is not static. Authors continually devise new symbols and ascribe new meanings to existing ones. The complement of symbols requested from Unicode was frozen in mid-1998; a few additions were made only to achieve consistency in the base set of symbols affected by combining codes, namely the VS and negation marker. About 50 additional symbols used in math, physics and theoretical computer science have been identified since then. Documentation must be completed for these, and the formal request presented to the UTC for their addition.

The glyph complement was frozen somewhat later than the Unicode complement, but this too remains to be addressed.

It has not been determined how these, or further additions, are to be handled. Nonetheless, I am still actively collecting citations for new notation, and welcome contributions.

Acknowledgments

Three UTC members were involved heavily in advancing this project: Murray Sargent of Microsoft, formerly a practicing physicist; Ken Whistler of Sybase, the UTC archivist and general editor of Unicode 3.0; and Asmus Freytag, the UTC font specialist, who is responsible for typesetting the Unicode manuals and ISO 10646. Their knowledge and experience of character code standards has proved invaluable, and the project's success owes much to their thoughtful assistance. Ken in particular is able to explain in non-expert terms the UTC's requirements and the historical background affecting specific decisions. From the other direction, he can quickly assess the evidence for support of an item and, if convinced that it has sufficient merit, can convince the rest of the committee that it should be included. Thus, for the math proposal, the strategy was to provide sufficient evidence for a symbol to convince Ken, and then let him persuade the committee using arguments they would find convincing.

The contributors to the symbol collection were always willing to provide additional information. Neil Soiffer of Wolfram Research attended several UTC meetings to explain the uses of several "letter-like symbols" that have special significance in Mathematica®.

Patrick Ion, co-chair of the W3C MathML working group, took my place at several UTC meetings when questions were expected that would best be answered by a practicing mathematician, such as, "Do you know for a fact that [some particular symbol] is used, and are you sure it isn't the same as [some other symbol]?" His presence and his answers successfully conveyed the importance of this project to the mathematical community.

Thanks to all for their efforts.

For more information

The history of the STIX project is recorded at the Web site `http://www.ams.org/STIX/`.

References

[1] Association for Font Information Interchange, *International Glyph Register, Volume 1: Alphabetic scripts and symbols*, Rochester, 1993.

[2] International Organization for Standardization, ISO 8879:1986, *Information Processing — Text and office systems — Standard Generalized Markup Language (SGML)*, Geneva, 1986.

[3] International Organization for Standardization, ISO 9541:1991, *Information Technology — Font Information Interchange — Part 1: Architecture*, Geneva, 1991.

[4] International Organization for Standardization, ISO 9573-13:1991, *Information Technology — SGML Support Facilities — Techniques for using SGML — Part 13: Public entity sets for mathematics and science*, Geneva, 1991.

[5] International Organization for Standardization, ISO 10646-1:1993, *Information Technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*, Geneva, 1992.

[6] "STIPUB Request for Proposals for Scientific and Technical Fonts for the STIX Project", November 16, 1999 (unpublished).

[7] The Unicode Consortium, *The Unicode Standard, Version 2.0*, Addison-Wesley Developers Press, Reading, MA, 1996.

[8] The Unicode Consortium, *The Unicode Standard, Version 3.0*, Addison Wesley Longman, Inc., Reading, MA, 2000.

[9] *Webster's New Collegiate Dictionary*, G. & C. Merriam, Springfield, MA, 1959.

[10] Justin Ziegler, *Technical Report on Math Font Encodings*, available from CTAN (`http://www.tug.org/tex-archive/info/ltx3pub/l3d007.tex` and supporting files in the same directory), 1994.

Barbara Beeton

Some experience in converting LH Fonts from METAFONT to Type1 format

Alexander Berdnikov
Institute of Analytical Instrumentation
Rizsky pr. 26
198103 St. Petersburg, Russia
berd@ianin.spb.su

Yury Yarmola
Institute of Analytical Instrumentation
Rizsky pr. 26
198103 St. Petersburg, Russia
yar@legion.ru

Olga Lapko
Institute of Analytical Instrumentation
Rizsky pr. 26
198103 St. Petersburg, Russia
olga@mir.msk.su

Andrew Janishewsky
Institute of Analytical Instrumentation
Rizsky pr. 26
198103 St. Petersburg, Russia
JAW@p0.f209.n5030.z2.fidonet.org

Abstract

This paper decribes the long-term project of *Cyr*TEX (Association of Cyrillic TEX Users Group) to convert Cyrillic LH fonts from METAFONT format into Type1 format, which is more suitable for modern typography.

Sasha Berdnikov

Typesetting TEX documents containing computer code

Włodek Bzyl
matwb@univ.gda.pl

Abstract

I would like to present a preprocessor driven by grammars able to automatically mark-up pieces of computer code immersed in TEX documents. By computer code I mean any language for which a context free grammar exists, which YACC will accept. These include almost any computer language. But my tool could use any such grammar to automatically mark-up text written in the language defined by it.

Introduction

One of the most difficult tasks in technical typesetting is to get computer programs to look right. [. . .] No automatic system can hope to find the best breaks in programs, since an understanding of the semantics will indicate that certain breaks make the program clearer and reveal its symmetries better.

— D. E. KNUTH, Digital Typography

Computer books and journals do not look as beautiful as they used to. It is not their content that is unsatisfactory, rather the typography is strange. The example below illustrates that. It is a really disgusting piece of typography taken from the Polish translation of an English book.

Program JavaScript przedstawiony w listingu 6.2 stanowi przyklad zastosowania cookie.

```
//=======================================
// Here are our standard Cookie routines
//=======================================
//---------------------------------------
// SetCookieEZ - Quickly sets a cookie
//    which which will last until the user
//    shuts down his browser
//---------------------------------------
function SetCookieEZ(name, value) {
  document.cookie=name+"="+escape(value);
}
//---------------------------------------
// GetCookie - Returns the value of the
//    specified cookie or null if the
//    cookie doesn't exist
```

It seems that typesetters think that a typewriter is the best tool to prepare readable and clear programs. This situation reminds us of an old Polish proverb: "The shoemaker does not wear shoes". Why? Because it is inexplicable that the typesetter does not typeset programs. Why do they use verbatim mode of typesetting — a kind of 'ASCII typography'? Do they have nothing better? Although typography is well developed, that of computer code lags far behind. However, there are rules for typographic formatting of computer code. Developing excellent computer code typography was pioneered by two people: Peter Naur and Myrtle Kellington, who set the standards that were adopted by many computer journals [8, 6]. But it seems that they are no longer used at all.

Editor's note: What does the citation 6 in the above paragraph signify? — it's a citation of Knuth

In the next section I will try to analyze why, and what makes code typesetting so difficult. In the following one, I will present my idea of a prettyprinting tool, which combines Knuth's [3] and Oppen's [9] approaches. One example is worth a thousand words, so the last section presents two longer examples typeset by my tool.

The term 'prettyprinting', which goes back to 1975 book *Programming Proverbs* by Henry Ledgard, is nowadays used instead of 'code typesetting'.

Prettyprinting HOWTO

Obviously ASCII typography is what programmers are familiar with and see most often. Look at something less ugly.

The following code uses the extended features of BC to implement a simple program for calculating checkbook balances.

```
print "Check book program!\n";
print "  Exit by a 0 transaction.\n\n";
print "Initial balance? ";
bal = read();
```

```
bal /= 1;
print "\n";
while (1) {
  "current balance = "; bal
  "transaction? "; trans = read()
  if (trans == 0) break;
  bal -= trans
  bal /= 1
```

Although the structure of the code is clearly laid out, the beginner will have problems with recognizing the elements which are predefined part of the language. Adding some typography to this example solves this problem and makes the program clearer.

The following code uses the extended features of BC to implement a simple program for calculating checkbook balances.

print "Check book program!\n";
print " Exit by a 0 transaction.\n\n";
print "Initial balance? ";
bal = **read**( );
bal /= 1;
print "\n";
while (1) {
  "current balance = "; *bal*
  "transaction? "; *trans* = **read**( )
  if (*trans* ≡ 0) **break**;
  bal −= *trans*
  bal /= 1
}

A simply bit of typography makes the difference. So the question is: why is it not used? Adding typography means adding mark-up to the source code. This makes the source for the example look like

```
The following code uses the extended
features of |\acro{BC}| to implement a
simple program for calculating checkbook
balances.
\startPP[BC]
\PPK{print}\PPbreakspace \PPS{"Check\ book\
\PPK{print}\PPbreakspace \PPS{"\ \ Exit\ by
\PPK{print}\PPbreakspace \PPS{"Initial\ bal
\PPV{bal}\PPequal \PPK{read}\PPbraceleft \P
\PPV{bal}\PPslasheq \PPN{1}\PPsemicolon \PP
\PPK{print}\PPbreakspace \PPS{"\\n"}\PPsemi
\PPK{while}\PPspace \PPbraceleft \PPN{1}\PP
  \PPS{"current\ balance\ =\ "}\PPsemicolon
  \PPS{"transaction?\ "}\PPsemicolon \PPspa
  \PPK{if}\PPspace \PPbraceleft \PPV{trans}
  \PPV{bal}\PPminuseq \PPV{trans}\PPforce
  \PPV{bal}\PPslasheq \PPN{1}\PPforce
  \PPbackspace \PPparenright
\stopPP[BC]
```

We can see that it is not that simple. Markup requires consistency and time which people do not have, whereas computers have both. So, why not use computers? — it seems possible, because computer languages, unlike natural languages, are unambiguously described by grammars. So we can try to use grammars to control the process of marking-up code. For example, in C or PERL, one grammar rule for *statement* says that *statement* is build of the opening brace '`{`' followed by a *statement_list* and the closing brace '`}`', which could be concisely written as:

statement → ´`{`´ *statement_list* ´`}`´

Now, assume that we want to typeset braces on separate lines with *statement_list* typeset indented between them

```
{
  statement_list
}
```

This could be done in the following way: after recognizing the statement components, we typeset the opening brace followed by the newline; next we typeset indented each statement from the statement list; next we typeset the newline followed by the closing brace.

Editor's note: Propose above paragraph be replaced, to save enough space that the paper doesn't run to 6pp; wording would be (note that the operations described also appear later on): This could be done very simply once the syntactic element has been recognized.

However, prettyprinting is not that simple, because programmers use both syntax and semantics to make their programs clearer and more readable. This makes the task of building a wholly automatic prettyprinting tool impossible. Fortunately, the conflicts between syntax and semantics are sufficiently rare that it is acceptable to require the user to override syntax-based decisions when necessary.

Building a tool

Many people have tried to formalize and implement the idea of prettyprinting. William McKeeman [11] was the first to present a prettyprinting algorithm. Oppen [9] came up with the idea of using the original grammars to control the marking-up process.

Knuth's [3, 12] approach is based on his so-called 'format primitives' listed below.

indent	indent the next line one more notch
outdent	indent the next line one less notch
optbreak	optional line break
backspace	backspace one notch
breakspace	optional break or space
force	force line break
bigforce	force line break and a little extra vertical space
noindent	no indentation

Spacing in expressions is inherited from the TEX mathematical mode.

Knuth does not use the original PASCAL or C grammars to derive prettyprinting grammars. For each language, a specially designed and optimized context sensitive grammar is created. These grammars do not describe reference languages, which means that some pieces of correct code would not be prettyprinted and incorrect code would be prettyprinted. Markup is done by manually built parsers. Using parsers for building prettyprinters is a double-edged sword: comments are lost during parsing, and while prettyprinting they have to be put back. Knuth's tools recover comments too.

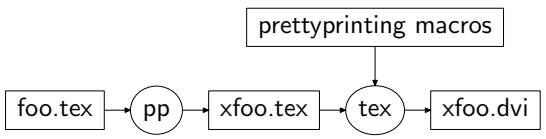
The main difficulty with Knuth's approach lies in creating prettyprinting grammars. So far only two such nontrivial grammars have been created. It took 10 years for the prettyprinting C grammar to evolve. Therefore, I think that the Oppen's idea of enhancing the original grammars is easier to implement.

For example, look at the rule for *statement* above and assume, that we want to typeset statement list indented within braces. The following rule will do:

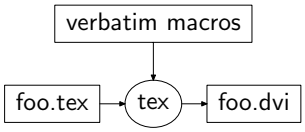
statement → ´{´ *indent force statement_list force backspace* ´}´ *outdent force*

This reads as follows: print the opening brace { and prepare to *indent* following lines; next break the line with *force* and typeset *statement_list*; now, break the line again with *force*, *backspace* and print the closing brace }; finally, remove with *outdent* the indentation for the following lines and *force* the line break again.

Combining the ideas above I designed a prettyprinting tool named pp which works as a preprocessor for TEX.



It is also possible to typeset code verbatim — simply typesetting as it used to be.



Examples

There are many computer languages in use nowadays. Some of them I use on a day to day basis, some I use occasionally, some I want to look at, and still others are used by my colleagues. Therefore I want my tool to be able to learn as many languages as possible. The current version knows how to typeset only two languages: EBNF — extended Backus-Naur formalism and BC — the language of the binary calculator (a tool available on every UNIX system), because it is still being developed.

Someone said "One picture is worth one thousand of words," so I want to end up with two pieces of prettyprinted code. Because there is no general agreement how to typeset computer documents, as opposed to mathematical ones, some readers may like these examples and others not. Moreover, mathematicians, have a well developed notation which is supported by suitable fonts. This may explain why these examples are not as clear and readable as they could be.

Source: Ken Arnold, James Gosling.
"The Java Programming Language."
Addison Wesley Longman, Inc. 1996.

Consider the two groups of productions:

FieldDeclaration → {*FieldModifiers*}∗ *Type VariableDeclarators*
;
FieldModifiers → *FieldModifier*
| *FieldModifiers FieldModifier*
;
FieldModifier → keywords common for field and method
| `transient` | `volatile`
;

and:

MethodHeader → {*MethodModifiers*}∗ *ResultType MethodDeclarator* {*Throws*}∗
;
MethodModifiers → *MethodModifier*
| *MethodModifiers MethodModifier*
;

MethodModifier → keywords common for field and method
| `abstract` | `native` | `synchronized`
;

where common keywords consists of: `public`, `protected`, `private`, `final`, `static`.

Source: Philip A. Nelson.
LIBMATH.B.
The arbitrary precision math library for the BC calculator.

To compute exponential we use the fact that $e^x = (e^{x/2})^2$. When x is small enough, we use the series: $e^x = 1 + x + x^2/2! + x^3/3! + \ldots$.

```
scale = 20;
define e(x) {
  auto a, d, e, f, i, m, n, v, z
  ▸ a — holds x^y of x^y/y!
  ▸ d — holds y!
  ▸ e — is the value x^y/y!
  ▸ v — is the sum of the e's
  ▸ f — number of times x was divided by 2.
  ▸ m — is 1 if x was minus.
  ▸ i — iteration count.
  ▸ n — the scale to compute the sum.
  ▸ z — orignal scale.
  ▸ Check the sign of x.
  if (x < 0) {
    m = 1; x = -x
  }
  ▸ Precondition x.
  z = scale;
  n = 6 + z + .44 * x;
  scale = scale(x) + 1;
  while (x > 1) {
    f += 1; x /= 2; scale += 1;
  }
  ▸ Initialize the variables.
  scale = n;
  v = 1 + x
  a = x
  d = 1
  for (i = 2; 1; i++) {
    e = (a *= x)/(d *= i)
    if (e == 0) {
      if (f > 0) while (f--) v = v * v;
      scale = z
      if (m) return (1/v);
      return (v/1);
    }
    v += e
  }
}
```

Define the logarithm function.

```
define l(x) {
  auto e, f, i, m, n, v, z
  ▸ Return something for the special case.
  if (x <= 0) return ((1 - 10^scale)/1)
  ▸ Precondition x to make .5 < x < 2.0.
  z = scale; scale = 6 + scale;
  f = 2; i = 0;
  ▸ For large numbers.
  while (x >= 2) {
    f *= 2; x = sqrt(x);
  }
  ▸ For small numbers.
  while (x <= .5) {
    f *= 2; x = sqrt(x);
  }
  ▸ Set up the loop.
  v = n = (x - 1)/(x + 1)
  m = n * n
  ▸ Sum the series.
  while (x < 2) {
    e = (n *= m)/i
    if (e == 0) {
      v = f * v
      scale = z
      return (v/1)
    }
    v += e
  }
}
```

The sin function uses the standard series: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$

```
define s(x) {
  auto e, i, m, n, s, v, z
  ▸ Precondition x.
  z = scale
  scale = 1.1 * z + 2;
  v = a(1)
  if (x < 0) {
    m = 1;
    x = -x;
  }
  scale = 0
  n = (x/v + 2)/4
  x = x - 4 * n * v
  if (n % 2) x = -x
  ▸ Do the loop.
  scale = z + 2;
  v = e = x
  s = -x * x
```

```
    for (i = 3; 1; i += 2) {
      e *= s/(i * (i - 1))
      if (e ≡ 0) {
        scale = z
        if (m) return (-v/1);
        return (v/1);
      }
      v += e
    }
  }
```

For arctan we use the formula: $\arctan(x) = \arctan(c) + \arctan((x-c)/(1+xc))$ for small c (0.2 here). For $x \le 0.2$, use the series: $\arctan(x) = x - x^3/3 + x^5/5 - x^7/7 + \ldots$.

```
  define a(x) {
    auto a, e, f, i, m, n, s, v, z
    ▸ a is the value of a (0.2) if it is needed.
    ▸ f is the value to multiply by a in the return.
    ▸ e is the value of the current term in the
      series.
    ▸ v is the accumulated value of the series.
    ▸ m is 1 or -1 depending on x (-x → -1);
    ▸ results are divided by m.
    ▸ i is the denominator value for series element.
    ▸ n is the numerator value for the series
      element.
    ▸ s is -x · x.
    ▸ z is the saved user's scale.
    m = 1;
    ▸ Negative x?
    if (x < 0) {
      m = -1; x = -x;
    }
    ▸ Special case and for fast answers
    if (x ≡ 1) {
      if (scale ≤ 25) return
              (.78539816339744830961566080/m)
    }
    if (x ≡ .2) {
      if (scale ≤ 25) return
              (.19739555984988075837004970/m)
    }
    ▸ Save the scale.
    z = scale;
    ▸ Note: a and f are known to be zero due to
      being auto vars. Calculate arctan of a
      known number.
    if (x > .2) {
      scale = z + 5; a = a(.2);
    }
    ▸ Precondition x.
    scale = z + 3;
    while (x > .2) {
      f += 1; x = (x - .2)/(1 + x * .2);
    }
    ▸ Initialize the series.
    v = n = x;
    s = -x * x;
    ▸ Calculate the series.
    for (i = 3; 1; i += 2) {
      e = (n *= s)/i;
      if (e ≡ 0) {
        scale = z;
        return ((f * a + v)/m);
      }
      v += e
    }
  }
```

References

[1] Blashek, Günter and Johannes Sametinger. "User-Adaptable prettyprinting." *Software — Practice & Experience* **19** (July 1989).

[2] Gärtner, Felix. 1998. *The PretzelBook*. Available online as part of the PRETZEL distribution, in directory `~gaertner/pretzel` at `www.iti.informatik.th-darmstadt.de`

[3] Knuth, Donald E. 1983. "The WEB System of Structured Documentation." Computer Science Report 980. Stanford University. Available online as part of the WEB distribution, in file `~web/doc/webman.tex` at `ftp.dante.de`.

[4] Knuth, Donald E. "Literate Programming", *The Computer Journal* **27** (1984). pp. 97–111.

[5] Knuth, Donald E. 1986. "How to read a WEB." Appeared in *Computers & Typesetting*, Volume B. Addison-Wesley.

[6] Knuth, Donald E. 1992. *Literate Programming*. Center for the Study of Language and Information Leleand Stanford Junior University.

[7] Knuth, Donald E. 1999. *Digital Typography*. Center for the Study of Language and Information Leleand Stanford Junior University.

[8] Naur, Peter [ed.] et al. "Report on the algorithmic language ALGOL 60." *Communications of the ACM* **3** (May 1960). pp. 299–314.

[9] Oppen, Derek C. "Prettyprinting." *ACM Transactions on Programming Languages & Systems* **2** (1980). pp. 465–483.

[10] Ramsey, Norman. 1988. "A SPIDER user's guide." Technical Report, Princeton

University. Available online as part of the SPIDER distribution, in file `~web/spiderweb/doc/spiderwebman.tex` at `ftp.dante.de`.

[11] McKeeman, William. "Algorithm 268." Communications of the ACM **8** (1965). pp. 667–668.

[12] Knuth, Donald E. and Silvio Levy. 1990. "The CWEB System of Structured Documentation." Computer Science Report 1336. Stanford University. Available online as part of the CWEB distribution, in file `~web/c_cpp/cweb/cwebman.tex` at `ftp.dante.de`.

Włodek Bzyl

XMLTEX: A non validating (and not 100% conforming) namespace aware XML parser implemented in TEX

David Carlisle
NAG Ltd
Jordan Hill Road
Oxford
davidc@nag.co.uk

Abstract

XMLTEX implements a non validating parser for documents matching the W3C XML Namespaces Recommendation.

Introduction

XMLTEX may be used simply to parse a file (expanding entity references and normalising namespace declarations) in which case it records a trace of the parse on the terminal. However, in normal use, the information from the parse is used to trigger TEX typesetting code. Declarations (in TEX syntax) are provided as part of XMLTEX to associate TEX code with the start and end of each XML element, attributes, processing instructions, and with unicode character data.

Installation

The XMLTEX parser itself does not require LATEX. It may be loaded into `initex` to produce a format capable of parsing XML files. However such a format would have no convenient commands for typesetting, and so normally XMLTEX will be used on top of an existing format, normally LATEX. In this section we assume that the document to be processed is called `document.xml`.

Using XMLTEX as an input to the LATEX command LATEX requires a document in TEX syntax, not XML. To process `document.xml`, first produce a two line file called `document.tex` of the following form:

```
\def\xmlfile{document.xml}
\input xmltex.tex
```

No other commands should appear in this file!

The document may then be processed with the command: `latex document`, or some equivalent procedure in the user's TEX environment.

Using XMLTEX as a TEX format built on LATEX Some users may prefer to set up XMLTEX as a format in its own right. This may speed things up slightly (as `xmltex.tex` need not be read each time) but more importantly perhaps it allows the XML file to be processed directly without needing to make the `.tex` wrapper.

To make a format, some command such as the following is used, depending on the user's TEX system.

```
initex &latex xmltex
initex \&latex xmltex
tex -ini &latex xmltex
tex -ini \&latex xmltex
```

This will produce a format file `xmltex.fmt`. It is then possible to make an `xmltex` command by copying the way the `latex` command is defined in terms of `latex.fmt`. Depending on the TEX system, this might be a symbolic link, or a shell script, or batch file, or a configuration option in a setup menu.

Making an XMLTEX format 'from scratch' It may be convenient, for some, to build an XMLTEX format as above, starting from the LATEX format. However, other users may prefer to work with an `initex` with no existing format file. Even for those who wish to use standard LATEX it may be preferable to make a TEX input file that first inputs `latex.ltx` then `xmltex.tex`. In particular this permits different hyphenation and language customisation for XMLTEX than for LATEX. Many of the features of the language support in LATEX are related to modifying the input syntax to be more convenient. Such changes are not needed in XMLTEX as the input syntax is always XML. Some language files may change the meaning of such characters as `<` which would break the XMLTEX parser. Also, rather than using `latex.ltx` one can in principle use a modified `docstrip` install file and produce a 'cut down' LATEX that omits features that are not going to be used in XMLTEX.

Unfortunately the support for this method of building XMLTEX (and access to non-English hyphenation generally) is not fully designed and totally undocumented.

Using XMLTEX

XMLTEX by default 'knows' nothing about any particular type of XML file, and so needs to load external files containing specific information. This section describes how the information in the XML file determines which files will be loaded.

1. If the file begins with a Byte Order Mark, the default encoding is set to UTF-16. Otherwise the default encoding is UTF-8.
2. If (after an optional BOM) the document begins with an XML declaration that specifies an encoding, this encoding will be used, otherwise the default encoding will be used. A file with name of the form *encoding*`.xmt` will be loaded that maps the requested encoding to Unicode positions. (It is an error if this file does not exist for the requested encoding.)
3. If the document has a DOCTYPE declaration that includes a local subset then this will be parsed. If any external DTD entity is referenced (by declaring and then referencing a parameter entity) then the SYSTEM and PUBLIC identifiers of this entity will be looked up in a catalogue (to be described below). If either identifier is known in the catalogue the corresponding XMLTEX package (often with `.xmt` extension) will be loaded.
4. After any local subset has been processed, if the DOCTYPE specifies an external entity, the PUBLIC and/or SYSTEM identifiers of the external DTD file will be similarly looked up, and a corresponding XMLTEX file loaded if known.
5. As each element is processed, it may be 'known' to XMLTEX by virtue of one of the packages loaded, or it may be unknown. If it is unknown then if it is in a declared namespace, the namespace URI (not the prefix) is looked up in the XMLTEX catalogue. If the catalogue specifies an XMLTEX package for this namespace it will be loaded. If the element is not in a namespace, then the element name will be looked up in the catalogue.
6. If after all these steps the element is still unknown then depending on the configuration setting either a warning or an error will be displayed. (Currently only warning implemented.)

The XMLTEX Catalogue As has already been explained, XMLTEX requires a mapping between PUBLIC and SYSTEM identifiers, namespace URI, and element names, to files of TEX code. This mapping is implemented by the following commands:
`\NAMESPACE{`*URI*`}{`*xmt-file*`}`
`\PUBLIC{`*FPI*`}{`*file*`}`
`\SYSTEM{`*URI*`}{`*file*`}`
`\NAME{`*element-name*`}{`*xmt-file*`}`
`\XMLNS{`*element-name*`}{`*URI*`}`

As described above, if the first argument of one of these commands matches the string specified in the XML source file, the corresponding TEX commands in the file specified in the second argument are loaded. The PUBLIC and SYSTEM catalogue entries may also be used to control which XML files should be input in response to external entity references. The `\XMLNS` command is rather different; if an element in the null namespace does not have any definition attatched to it, this declaration forces the default namespace to the given URI. The catalogue lookup is then repeated. This allows for example documents beginning `<html>` to be coerced into the XHTML namespace.

These commands may be placed in a configuration file, either `xmltex.cfg`, in which case they apply to all documents, or in a configuration file '`\jobname.cfg`' (e.g., `document.cfg` in the example in section 'Using XMLTEX', above) in which case the commands just apply to the specified document.

Configuring XMLTEX In addition to the 'catalogue' commands described earlier there are other commands that may be placed in the configuration files.

- `\xmltraceonly`
  This stops XMLTEX from trying to typeset the document. The external files specified in the catalogue are still loaded, so that the trace may report any elements for which no code is defined, but no actual typesetting takes place. In the event of unknown errors it is always worth using XMLTEX in this mode to isolate any problems.

  It may be noted that if an XMLTEX format is built just using `initex` without any typesetting commands, the resulting format should still be able to parse any XML file if `xmltex.cfg` just specifies `\xmltraceonly` and `\jobname.cfg` is empty.
- `\xmltraceoff`
  By default XMLTEX provides a trace of its XML parse, displaying each element begin and end.

This command in `xmltex.cfg` or `\jobname.cfg` will stop the trace being produced.

- `\inputonce{`*xmt-file*`}`
  The catalogue entries specify that certain files should be loaded if XML constructs are met. Alternatively the files may just always be loaded. The system will ignore any later requests to load. This is especially useful if an XMLTEX format is being made.

- `\UnicodeCharacter{`*hex-or-dec*`}{`*tex-code*`}`
  The first argument specifies a unicode character number, in the same format as used for XML character entities, namely either a decimal number, or an upper case Hex number preceded by a lower case '`x`'.

  The second argument specifies arbitrary TEX code to be used when typesetting this character. Any code in the XML range may be specified (i.e., up to x10FFFF). Although codes in the 'ASCII' range, below 128, may be specified, the definitions supplied for such characters will not by default be used. The definition will however be stored and used if the character is activated using the command described below.

- `\ActivateASCII{`*hex-or-dec*`}`
  The argument to this command should be a number less than 128. If a character is activated by this command in a configuration file then any special typesetting instructions specified for the character will be executed whenever the character appears as character data.

  Some ASCII characters are activated by default. The list is essentially those characters with special meanings to either TEX or XML.

If a format is being made, there are essentially two copies of `xmltex.cfg` that may play a role. The configuration file input when the format is made will control catalogue entries and packages built into the format. A possibly different `xmltex.cfg` may be used in the input path of 'normal' TEX, this will then be used for additional information loaded each run.

In either case, a separate configuration file specific to the given XML document may also be used (which is loaded immediately after `xmltex.cfg`).

Stopping xmltex

XMLTEX should stop after the end of the document element has been processed. If something goes wrong one may be offered TEX's `*` prompt from which one might choose to exit with `<?xmltex stop?>`.

XMLTEX package files

XMLTEX package files are the link between the XML markup and TEX typesetting code. They are written in TEX (rather than XML) syntax and may load directly or indirectly other files, including LATEX class and package files. For example a file loaded for a particular document type may directly execute `\LoadClass{article}`, or alternatively it may cause some XML element in the document to execute `\documentclass{article}`. In either case the document will suffer the dubious benefit of being formatted according to the style implemented in the standard article class. Beware though that the package files may be loaded at strange times, the first time a given namespace is declared in a document, and so the code should be written to work if loaded inside a local group.

Characters in XMLTEX package files have their normal LATEX meanings except that line endings are ignored so that there is no need to add a `%` to the end of lines in macro code. Unlike LATEX `.fd` file conventions, other white space is *not* ignored.

The available commands are:

- `\FileEncoding{`*encoding*`}`
  This is the analogue for TEX syntax files of the encoding specification in the XML or text declaration of XML files. If it is not specified the file will be assumed to be in UTF-8.

- `\DeclareNamespace{`*prefix*`}{`*URI*`}`
  This declares a prefix to be used *in this file* for referring to elements in the specified namespace. If the prefix is empty then this declares the default namespace (otherwise, unprefixed element names refer to elements that are not in a namespace).

  Note that the elements in the XML document instance may use a different prefix, or no prefix at all to access this namespace. In order to resolve these different prefixes for the same namespace, each time a namespace is encountered for the first time (either by `\DeclareNamespace` in a preloaded package, or in a namespace declaration in the XML instance) then it is allocated a new number and any further namespace declaration for the same URI just locally associates a prefix with this number. It is these numbers that are displayed when the XML trace of the parse of the document is shown, and also if any element is written out to an external file it will have a normalised numerical prefix whichever prefix it originally had. (Numeric prefixes are not legal XML, but this is an advantage, as it

ensures these internal forms can not clash with any prefix actually used in the document.)

Three namespaces are predeclared. The null namespace (0), the XML namespace `http://www.w3.org/1998/xml` (1) which is predeclared with prefix `xml` as specified in the Namespace Recommendation, and the XMLTEX namespace `http://www.dcarlisle.demon.co.uk/xmltex` (2) which is not given a default prefix, but may be used to have XML syntax for some internal commands (eg to have .aux files fully in XML, currently they are a hybrid mixture of some TEX and some XML syntax).

- `\XMLelement{`*element-qname*`}{`*attribute-spec*`}{`*begin-code*`}{`*end-code*`}`
  This is similar to a LATEX `\newenvironment` command. It declares the code to execute at the start and end of each instance of this element type. This code will be executed in a local group (like a LATEX environment). The second argument declares a list of attributes and their default values using the `\XMLattribute` command whose description follows.
- `\XMLelement{`*element-qname*`}{`*attribute-spec*`}{\xmlgrab}{`*end-code*`}`
  A special case of the above command (which may be better made into a separate declaration) is to make the *start-code* just be the command `\xmlgrab`. In this case the *end-code* has access to the element content (in XML syntax) as `#1`. This content isn't literally the same as the original document, namespaces, white space and attribute quote symbols will all have been normalised.
- `\XMLattribute{`*attribute-qname*`}{`*command-name*`}{`*default*`}`
  This command may only be used in the argument to `\XMLelement`. The first argument specifies the name of an attribute (using any namespace prefixes current for this package file, which need not be the same as the prefixes used in the document). The second argument gives a TEX command name that will be used to access the value of this attribute in the begin and end code for the element. (Note using TEX syntax here provides a name independednt of the namespace declarations that are in scope when this code is executed). The third argument provides a default value that wil be used if the attribute is not used on an instance of this element.

  The special token `\inherit` will cause the command to have a value set in an ancestor element if this element does not specify any value.

  If a TEX token such as `\relax` is used as the default the element code may distinguish the case that the attribute is not used in the document.
- `\XMLnamespaceattribute{`*prefix*`}{`*attribute-qname*`}{`*command-name*`}{`*default*`}`
  This command is similar to `\XMLattribute` but is used at the top level of the package file, not in the argument to `\XMLelement`. It is equivalent to specifying the attribute in *every* element in the namespace specified by the first argument. As usual the prefix (which may be to denote the default namespace) refers to the namespace declarations in the XMLTEX package: the prefixes used in the document may be different.
- `\XMLentity{`*name*`}{`*code*`}`
  Declare an (internal parsed) entity, this is equivalent to a `<!ENTITY]]>` declaration, except that the replacement text is specified in TEX syntax.
- `\XMLname{`*name*`}{`*command-name*`}`
  Declare the TEX command to hold the (normalised, internal form) of the XML name given in the first argument. This allows the code specified in `\XMLelement` to refer to XML element names without knowing the encodings or namespace prefixes used in the document. Of particular use might be to compare such a name with `\ifx\XML@parent` which will allow element code to take different actions depending on the parent of the current element.
- `\XMLstring{`*command-name*`}<>`*XML Data*`</>`
  This saves the XML fragment as the TEX command given in the first argument. It may be particularly useful for redefining 'fixed strings' that are generated by LATEX document classes to use any special typesetting rules specified for individual characters.

XML processing

XMLTEX tries as far as possible to be a fully conforming non validating parser. It fails in the following respects.

- Error reporting is virtually non existent. Names are not checked against the list of allowed characters, and various other constraints are not enforced.
- A non validating parser is not forced to read external DTD entities (and this one does not). It is obliged to read the local subset and process entity definitions and attribute declarations. Entity declarations are reasonably well handled: External parameter entities are handled as above, loading a corresponding XMLTEX

file if known. External entities are similarly processed, inputting the XML file, a difference in this case is that if the entity is not found in the catalogue, the SYSTEM identifier will be used directly to `\input` as often this is a local file reference. Internal parsed entities and parameter entities are essentially treated as TEX macros, and nonparsed entities are saved along with their NDATA type, for use presumably by `\includegraphics`.

Any default attributes specified in the local subset are saved and added to the corresponding element befor any processing is triggered. Note that this defaulting, unlike the defaults specified with `\XMLattribute` are 'namespace unaware' and only apply to elements using the same expanded name. The element from th esame namespace but represented with a different prefix will not have these defaults applied.

- Support for encodings depends on having an encoding mapping file. Any 8-bit encoding that matches Unicode for the first 127 positions may be used by making a trivial mapping file. (The one for latin1 looks over complicated as it programs a loop rather than having 127 declarations saying that latin1 and Unicode are identical in this range).

  UTF-8 is supported, but support for UTF-16 is minimal. Currently only latin-1 values work: (In this range UTF-16 is just latin-1 with a null byte inserted after (or before, depending on endedness) each latin-1 byte. The UTF-16 implementation just ignores this null byte then processes as for latin-1. Probably the first few 8-bit pages could be similarly supported by making the low ASCII control characters activate UTF-16 processing but this will never be satisfactory using a standard TEX. Hopefully a setup for a 16bit TEX such as Omega will correct this.

Accessing TEX

In theory one should be able to control the document simply by suitable code specified by `\XMLelement` and friends, but sometimes it may be necessary to 'tweak' the output by placing commands directly in the source.

Two mechanisms are availalable to do this.

- Using the XMLTEX namespace. The XMLTEX namespace conatins a small (currently empty) set of useful TEX constructs that are accessed by XML syntax. For example if XMLTEX provides a mechanism for having XML (rather than LATEX) syntax toc files, it will need an analogue of `\contentsline` which might be an element accessed by `<xmltex:contentsline>`... where the XMLTEX prefix is declared on this or a parent element to be `xmlns:xmltex="http://www.dcarlisle.demon.co.uk/xmltex"`.

  As the XMLTEX namespace is declared but currently empty, a more useful variant of this might be:

- Declare a personal namespace for TEX tweaks, and load a suitable package file that attatches TEX code to the elements in this namespace (or at least specify the correspondence between the namespace and the package using `\NAMESPACE`). For instance, `<clearpage xmlns="/my/tex/tweak"/>` will force a page break if, at suitable points, the document contains:

  ```
  \NAMESPACE{/my/tex/tweak}{tweak.xmt}
  ```

  and

  ```
  \DeclareNamespace{twk}{/my/tex/tweak}
  \XMLelement{twk:clearpage}{\clearpage}
  ```

- A second different mechanism is available, to use XML processing instructions. A Processing Instruction of the form: `?xmltex` *TEX commands* `?>` will execute the TEX commands.

Bugs

None, of course.

Don't Read Past This Point

Thus section discusses some of the more experimental features of XMLTEX that may get a cleaner syntax (or be removed, as a bad idea) in later releases, and also describes some of the internal interfaces (which are also subject to change)

Input Encodings and States At any point while processing a document, XMLTEX is in one of two *states*: *tex* or *xml*.

States In the *xml state*, `<` and `&` are the only two characters that trigger special markup codes. Other characters, such as `!`, `>`, `=`, ...; may be used in certain XML constructs as markup but unless some code has been triggered by `<` they are treated simply as character data. All characters above 127 are 'active' to TEX and are used to translate the input encoding to UTF-8. All internal character handling is based on UTF-8, as described below. Some characters in the ASCII range, below 127 are also active by default (mainly punctuation characters used in XML constructs, such as the ones listed above). Some or all of the others may be activated using the `\ActivateASCII` command, which allows special

typesetting rules to be activated for the characters, at some cost in processing speed.

In the *tex state*, characters in the ASCII range have their usual TEX meanings, so letters are 'catcode 11' and may be used in TEX control sequences, \ is the escape character, & the table cell separator, etc. Characters above 127 have the meanings current for the current encoding just as for the *xml state*, probably this means that they are unusable in TEX code, except for the special case of referring to XML element names in the first argument to `\XMLelement` and releated commands.

Encodings Whenever a new (XML or TEX) file is input by the XMLTEX system the *encoding* is first switched to UTF-8. At the end of the input the encoding is returned to whatever was the current encoding. The encoding current while the file is read is determined by the encoding pseudo-attribute on the XML or text declaration in the case of XML files, or by the `\FileEncoding` command for TEX files. Note that the encoding mechanism *only* is triggered by XMLTEX file includes. Once an XMLTEX package file is loaded it may include other TEX files by `\input` or `\includepackage` these input command swill be transparent to the XMLTEX encoding system. The vast majority of TEX macro packages only use ASCII characters so this should not be a problem.

Note that if the `\includepackage` occurs directly in the XMLTEX package file, the TEX code will be included with a known encoding, the one specified in the XMLTEX package, or UTF-8. If however the `\includepackage` is included in code specified by `\XMLelement`, then it will be executed with whatever encoding is current in the document at the point that element is reached. Before XMLTEX executes the code for that element it will switch to the *tex state*, thus normalising the ASCII characters but characters above 127 will not have predefined definitions in this case.

Internally eveything is stored as UTF-8. So `.aux` and `.toc` files will be in UTF-8 even if the document (or parts of the document) used different encodings.

To specify a new encoding, if it is an 8 bit encoding that matches ASCII in the printable ASCII range, then one just needs to produce a file with name *encoding*`.xmt` (in lowercase, on case sensitive systems) this should consist of a series of `\InputCharacter` commands, giving the input character slot and the equivalent Unicode. If an encoding is specified in this manner character data will be converted to UTF-8 by *expansion* and so ligatures and inter letter kerns will be preserved. (Conversely if characters are accessed by character references, `Ӓ` then TEX arithmetic is used to decode the information and ligature information will be lost. For some large character sets, especially for Asian languages, these mechanisms will probably not prove to be sufficient. Alternative mechanisms are being investigated, but in the short term it may be necessary to always use UTF-8 if the input encoding is not strictly a ine byte extension of the ASCII code page.

XMLTEX Package Commands And TEX command may be used in an XMLTEX package, although the user should be aware that the file may be input into a local group, at the point in a document that a particular namespace is first used, for example. There are however some specific commands designed to be used in the begin or end code of `\XMLElement`.

- `\ignorespaces`
  This is similar to the TEX primitive of the same name, but redefined to work more naturally in this context.
- `\obeyspaces`
  Obey consecutive space characters, rather than treating consecutive runs as a single space. (A command of this name, but not this definition is in plain TEX.)
- `\obeylines`
  Obey end of line characters, rather than treating then as a space, force a line break. (A command of this name, but not this definition exists in plain TEX.)
- `\xmltexfirstchild#1\@`
  If the *start-code* for an element is specified as `\xmlgrab` then the *end-code* may use `#1` in order to execute the element content. However, the entire content is not always needed, and the construction `\xmltexfirstchild\#1\@` (with currently unpleasant syntax) will just evaluate the first child element of the content, discarding the remaining elements.
- `\xmltextwochildren\csa\csb#1`
  If it is known that the content will be exactly two child elements (e.g., a MathML `frac` or `sub` element) then this command may be used. The command executes the TEX code `\csa{`*child-1*`}\csb{`*child-2*`}` So either two TEX commands may be supplied, one will be applied to each child, or the second argument may be `{}` in which case the first argument may be a TEX command that takes two arguments. For example the code for MathML frac might be

```
\XMLelement{m:mfrac}
  {}
```

```
{\xmlgrab}
{\xmltextwochildren\frac{}#1}
```

- `\xmltexthreechildren\csa\csb\csc#1`
  As above, but more so.
- `\xmltexforall\csa{#1}`
  The TEX command `\csa` is executed repeatedly, taking as argument each time one child from the content '#1' of the current element. The command name `\xml@name` is set to the (normalised, internal) name of each child element before `\csa` is executed.
- `\NDATAEntity\csa\csb\attvalue`
  If the XML parser encounters an internal or external entity reference it expands it without executing any special hook that may be defined in an XMLTEX package. However NDATA entites are never directly encountered in an entity reference. They may only be used as an attribute value. If `\attvalue` is a TEX command holding the value of an attribute, as declared in `\XMLattribute` then `\NDATAEntity\csa\csb` `\attvalue` applies the two TEX commands `\csa` and `\csb` to the notation type and the value, in a way analogous to `\xmltextwochildren`, so for example the XML version of manual document, from which this paper is derived, specifies:

```
<!NOTATION URL SYSTEM "" >
<!ENTITY lppl SYSTEM
 "http://www.latex-project.org/lppl.txt"
  NDATA URL>
```

and this is handled by the following XMLTEX code

```
\XMLelement{xptr}
 {\XMLattribute{doc}{\xptrdoc}{}}
 {\NDATAEntity\xptrdoc\@gobble\url}
 {}
```

which saves the attribute value in `\xptrdoc` and then discards the notation name (URL) and applies the command `\url` to typeset the supplied URL.

David Carlisle

Abstract: Developing Interactive, Web-based Courseware

Donald W. DeLand and Greg Faron
Integre Technical Publishing Co.
4015-B Carlisle N.E.
Albuquerque, NM 87107 U.S.A.
don@integretechpub.com

Abstract

This paper describes our company's ongoing efforts to adapt mathematics textbooks for online delivery. We show how the textbooks, written primarily in LaTeX, have been adapted for web delivery using IBM's techexplorer browser plug-in, and enhanced using an extensive set of interactive Java applets. To complement these textbooks, we are developing an online study, homework, and testing system for mathematics that combines techexplorer, PHP (a free, open-source hypertext preprocessor), mySQL (an open-source database application), and application links to computer algebra systems. Our ultimate goal is to provide a comprehensive environment for delivering math courseware online.

Don DeLand

The amsrefs LaTeX package and the amsxport BibTeX style

Michael Downes
American Mathematical Society
mjd@ams.org

Introduction

When the bibliography entries in a LaTeX document are written in amsrefs form, they have rich internal structure and high-level markup close to what is traditionally found in BibTeX database files. Among other things, this raises the information quality of the LaTeX document if it is intended to serve as (or to yield via automatic translation) a self-contained archival version. Using amsrefs markup also means that the style of the bibliography can be specified completely in a LaTeX documentclass file, instead of being handled partly by LaTeX and partly by the BibTeX style file, in two different style languages, each of them more idiosyncratic than the other.

It has always been possible to write short bibliographies by hand without going through BibTeX, but those who do so usually put in as many ad hoc formatting commands as BibTeX would. An author who uses the amsrefs package when writing a bibliography by hand will be better off if it becomes necessary at some later time to change the style of the bibliography: the interior structural markup means that the same information can be reformatted in different ways by simple changes in the LaTeX setup rather than by explicitly changing formatting commands in each item.

Some of the sources that I consulted during the design of the amsrefs package are given in the bibliography. The bibliographies of [1] and [7] are recommended. The chief goal of my design efforts was to find a way of representing and citing bibliography entries at a level of markup abstract enough to support automatic formatting for a wide range of known bibliography styles without any change in the data; the secondary goal was to find a syntax for this representation that was natural in a LaTeX context and as easy as possible for authors to use. What I ended up with was a design that would leave any experienced TeXnician aghast at the implementation difficulties. Fortunately, however, many of the hardest bits could be dealt with using known methods; for example, Donald Arseneau's cite package established long ago that sorting and compressing lists of cite numbers is not horrendously impossible to do in TeX, and the technique used in amsrefs for combining multiple author names is adapted from some code of mine written a while back for the amsart documentclass.

Contents of a bibliography entry

Bibliography entries are done with a \bib command, not \bibitem, and look like this:

```
\bib{BW}{article}{
 author={Bertram, A.},
 author={Wentworth, R.},
  title={Gromov invariants for holomorphic
         maps on Riemann surfaces},
   date={1996},
journal={jams},
 volume={9},
 number={2},
  pages={529\ndash 571},
}
```

This will be recognizable as very similar to the data format used in BibTeX database files. There are, however, some key differences. If you use the amsxport BibTeX style and export from a BibTeX database, the differences will be automatically attended to, but if you write a short bibliography by hand, you should bear in mind the following points:

braces Always use braces to enclose the value of each field, never quotes (BibTeX allows both quotes and braces).

repeated fields Certain fields can be repeated (and should be, where applicable). As shown in this example, when there is more than one author, each author name is given separately. The task of combining the names as needed for the current publication is handled by LaTeX. The fields that are defined to be repeatable by the amsrefs package are author, editor, isbn, review, and translator.

inverted names It is recommended to give author and editor names uniformly in *Last, First* order. This is the form that provides the most flexibility with the least extra markup. When printed, the names will be automatically uninverted in whatever manner is specified by the bibliography style in use. (Some styles have the first author's

name inverted and the remaining names not inverted.) For suffixes such as III or Jr., write `Smith, John Q., III` or `Smith, John Q., Jr.` as the inverted form. For Chinese names and others where the surname comes first, if you write *Last First* without a comma the parts will never be transposed.

abbreviations In certain fields abbreviations may be used. In the example above the journal name `jams` will be expanded by LaTeX to J. Amer. Math. Soc. Abbreviations for journals and publishers can be defined with two commands provided for this purpose:

`\DefineJournal` takes four arguments: the abbreviation, the ISSN, a short form of the journal name, and the full journal name. For example,

```
\DefineJournal{mpcps}
  {0305-0041}
  {Math. Proc. Cambridge Philos. Soc.}
  {Mathematical Proceedings of the
   Cambridge Philosophical Society}
```

For `\DefinePublisher` the four arguments are: abbreviation, short form, full publisher name, and location, e.g.,

```
\DefinePublisher{ucp}
  {Univ. Chicago Press}
  {University of Chicago Press}
  {Chicago}
```

Beware of variation among the books of a single publisher in the cities of publication.

If the `amsrefs` package is invoked with the `jpa` option, it will automatically load an auxiliary package `amsjpa` that contains `\DefineJournal` and `\DefinePublisher` statements for more than a hundred of the journals and publishers mentioned most often in AMS bibliographies. To get a different set of abbreviations, you can put your own definitions in a `.sty` file and load them with `\usepackage`.

capitalization Proper nouns do not need to be written with extra braces: it is sufficient to write `Riemann` instead of `{R}iemann` or `{Riemann}` (as required by BibTeX). Do not capitalize words that are not proper nouns (unless you're writing in German, of course). Capitalization for English-language titles will be applied automatically where specified by the bibliography style. See **Capitalization of English titles**, below.

ndash Using `\ndash` instead of `--` for en-dashes is recommended. (And `\mdash` instead of `---`, for that matter.) See the remarks on `textcmds` in the **Auxiliary Packages** section.

date It is recommended to use a `date` field to give the year of publication; although `year` is also accepted, `date` is more general. If the date includes a month or month and day, using month numbers in ISO 8601 form is recommended, e.g., 1987-12 (or 1987-12-30 if a day is present). This allows month names to be printed in full or abbreviated (or left as numeric), at the behest of the current bibliography style, without changing the contents of the bibliography. For "Winter", "Spring", "Summer", "Fall", either use month numbers of 13, 14, 15, 16 (respectively), or just put in the text before the year:

```
date={Summer 1987},
```

The first mandatory argument of `\bib` is the citation key to be used with `\cite`. Like `\bibitem`, `\bib` also takes an optional argument to be used as the item label in the bibliography and as the printed output of the `\cite` command.

The second mandatory argument of `\bib` is the entry type. The recognized entry types are just about the same as the ones commonly recognized in BibTeX style files, except that `phdthesis` and `masterthesis` are subsumed under a single `thesis` category. By default an entry of this type is treated as a master's thesis; to indicate some other kind, one can write, e.g., `type={Ph.D. thesis}` or `type={Diplomarbeit}`. As a special case the abbreviations `type={phd}` and `type={masters}` are also recognized.

Restricted key-value scanning

Although the fields within a `\bib` command are given in standard LaTeX key-value notation, the parser used to scan the keys and their values is not the standard one from LaTeX's `keyval` package but one written especially for the `amsrefs` package in order to provide some refinements in the error checking. It is embodied as a separate package, `rkeyval`.

Missing commas If a comma is missing in a `\bib` command, you get an error message that tells you exactly what the problem is and where. With the standard key-value parsing provided by LaTeX this would not happen — a missing comma would lead to one erroneous value and one lost value without any warning to the user. For example, consider what happens with

```
\rotatebox[x=9pt y=9pt]{180}{UPSIDE DOWN}
```

This results in a value for `x` of "`9pt y`", while the second `9pt` is discarded and `y` retains its default value. Consequently the rotated box is positioned incorrectly and a spurious letter y is printed on the

page when LaTeX attempts to use the value of x. And there is no warning message to alert the user that something went wrong.

To be sure, when key-value notation is used for options that specify *how* some material should be printed, the values tend to be short and simple, they are normally written without braces, and missing commas are not all that frequent. But when the values contain textual material, and each key-value pair is given in braces on a separate line, missing comma errors are very easy to make and occur quite often in practice. It was the sinking feeling of this realization that drove me in the end to write an alternative parser, with great reluctance, after using the standard parser for nearly the whole development period of the amsrefs package.

Mandatory use of braces for values Requiring braces all the time is better for key-value pairs if the values are printable text. Because, of course, braces are the only completely reliable way to avoid the problems that afflict delimited arguments if any kind of nesting is involved or if the argument may legitimately contain the delimiter string.

If the braces are inadvertently left out, the error message looks like this (more or less):

```
! Package rkeyval Error:
         Missing open brace for key value.
 ...
l.10    year=1985,

?
```

In most cases LaTeX will be able to carry on after such an error and produce something close to the intended output.

Capitalization of English titles

The capitalization recommended for English titles is *sentence case*: all lowercase except for proper nouns (i.e., the same as is normally used anyway for all other languages with a Latin-1 character set). Initial caps can be applied on demand to a title written in this form, as explained in the discussion of \bibspec below.

The capitalization done by amsrefs succeeds rather well in following the rules given in the *Chicago Manual of Style*, as paraphrased here:

> Capitalize each word except for articles, coordinate conjunctions, and prepositions, or the word *to* in infinitives. Do capitalize pronouns and subordinate conjunctions. Always capitalize the first and last word of the title and the first and last word of any subtitles that it may contain. In a hyphenated compound, capitalize the second (or any later) word only if it is a noun or proper adjective, or it has equal force with the first word.

If some word is capitalized that should not be capitalized, putting braces around the word will prevent that. But I think it will be very seldom necessary in practice; in a test of some two hundred titles taken at random from AMS journal articles, all of the titles were capitalized correctly, even the ones containing more difficult fragments such as hyphenated compounds, math formulas, or \␣ and ~ for inter-word spaces.

Citations

The amsrefs package offers three primary citing commands: \cite, \citelist, \cites. Features include:

- sorting and range compression for numeric cite keys (like the excellent cite package)
- support for author-year citation schemes, with some additional commands \ycite, \ocite, \citeauthor, etc.
- back-reference capabilities (similar to those of the backref package that comes with hyperref)

The \cite command works just about the way the LaTeX book says it should, except that it supports an additional optional argument mechanism that avoids a pitfall associated with the standard optional argument syntax. Instead of \cite[Chapter 2]{xyz}, one writes

```
\cite{xyz}*{Chapter 2}
```

Using this * notation instead of the usual square brackets prevents the unexpected error messages that novice users meet if they incautiously attempt to use a \cite command with the square brackets inside another pair of optional argument brackets, for example,

```
\begin{thm}[\cite[Theorem 4.9]{xyz}]
```

The \citelist command takes one argument, which is simply a list of \cite commands, optionally separated by spaces.[1] Each \cite command may have its own optional argument.

```
\citelist{\cite{key1}
  \cite{key2}*{Chapter 2} \cite{key3}}
```

The \cites command is a straightforward variant of the \citelist command that can be used when none of the individual \cite commands have

[1] No commas or other inter-cite punctuation should be written between the \cite commands because that will all be supplied automatically and attempts to write it in by hand will only interfere.

an optional argument. Then one can give the list of cite keys without including a `\cite` command for each one:

```
\cites{key1,key2,...}
```

Use of multiple cite keys with `\cite` is deprecated, even though it must be supported for backward compatibility. Using `\cites` or `\citelist` is better because it clears up the semantically dubious old treatment of the optional argument.

Author-year citation schemes

In author-year citation schemes three main citing forms are required to cover the cases that in other schemes require only one form. The first form is used when the citation serves as a parenthetical annotation — i.e., it could be omitted without harming the grammatical structure of the sentence containing it. For example:

> The question first arose in systems theory (Rupp and Young, 1977).

The second form is like the first but is used when the author name is already present as a natural part of the sentence, and the cite therefore ought to supply only the year:

> Rupp and Young (1977) have investigated ...

The third form is used when the citation serves as a direct object or other inomissible noun-like object within its sentence. For instance, dropping the citation from the following sentence leaves a grammatically incomplete remainder:

> ... for further details, see Rupp and Young (1977).

The logic of requiring three forms is perhaps most clearly seen if we envision replacing the author-year citations by numerical citations. The type of text replaced by [14] is different in each case:

> ... arose in systems theory [14].
>
> Rupp and Young [14] have investigated ...
>
> ... for further details, see [14].

We delegate `\cite` to produce the primary parenthetical form (Author, Year) and provide `\ycite` ("year cite") and `\ocite` ("object cite") as the other forms. Plural forms `\ycites` and `\ocites` are also provided in parallel with `\cites`. And `\citeauthor` can be used to produce the list of author names without the year.

These correspond with command names from some other commonly used author-year packages as follows:

amsrefs	harvard	natbib
`\cite`, `\cites`	`\cite`	`\citep`
`\ycite`, `\ycites`	`\citeyear`	`\cite`
`\ocite`, `\ocites`	`\citeasnoun`	`\citet`
`\citeauthor`	(none)	`\citeauthor`
`\citeauthory`	(none)	`\citet`

Some people like to use `\citet` or `\citeasnoun` when the author name serves as the subject of a sentence. This seems to me a questionable blurring of the boundary between the essential text of the sentence and the bibliography pointer. But just in case I am mistaken (though you may gasp in disbelief at the thought), I have provided `\citeauthory{xyz}` as an abbreviation for

```
\citeauthor{xyz} \ycite{xyz}
```

I.e., the command name is `\citeauthor`+`\ycite` with the redundant second `cite` dropped.[2]

When an author-year scheme is in use, parens are normally added by `\cite`, `\citelist` and their variants — unless the character immediately following the command's argument is a closing paren. This simple rule of thumb suffices for almost all cases.

Starred forms `\cite*` and `\ocite*` print the full list of author names instead of an abbreviated list, when an (Author, Year) style of citation is in use. For other citation schemes they produce the same output as the unstarred forms. Some citation styles require the first cite to use the full list and subsequent ones to use the abbreviated version. With proper setup this can be done automatically, so that the starred forms of these commands should seldom be necessary in practice.

Bibliography style setup

With the `amsrefs` package all style changes can be done with LaTeX. You don't need to understand BibTeX's unnamed `bst` language. And because everything is handled from the LaTeX side, the bibliography style for a given document class can be specified completely in the class file instead of partly there and partly in a `.bst` file.

The overall style of the bibliography list is dictated by a `biblist` environment; it takes an optional argument which may contain overrides of list parameters and other style specs. This makes it easy to modify the style slightly for a particular document. Documentclasses that load the `amsrefs` package should supply their own definition of `biblist`, and doing so normally means that they can leave `thebibliography` environment unchanged, because

[2] Why not simply call it `\aycite`, you may ask? Well, I could hardly pass up the opportunity to get all six vowels in a single command name, could I?

amsrefs makes it a simple wrapper function that calls biblist:

```
\renewenvironment{thebibliography}[1]{%
  \bibsection
  \biblist[\resetbiblist{#1}]%
}{%
  \endbiblist
}
```

where \bibsection is normally defined as

```
\newcommand{\bibsection}{%
  \section*{\bibname}}%
```

When defining \bibsection, the amsrefs package uses \chapter if it is defined, otherwise \section. Of course you can always redefine \bibsection yourself if need be.

The interior formatting within entries is specified by \bibspec commands, one for each entry type. To illustrate, let's look at an example style spec for entries of type article:

```
\bibspec{article}{%
  +{}{\PrintAuthors}  {author}
  +{,}{ \textit}      {title}
  +{,}{ }             {journal}
  +{}{ \textbf}       {volume}
  +{}{ \parenthesize} {date}
  +{,}{ }             {pages}
  +{,}{ }             {note}
  +{.}{}              {transition}
  +{}{ }              {review}
}
```

It should be pretty obvious that each line specifies the formatting for a particular field. The fundamental model here is that after reading the data for a particular \bib command, LaTeX steps through the style spec and for each field listed, prints the field with the given formatting *if and only if the field has a nonempty value*. The + character at the beginning of each field spec must be followed by three arguments: the punctuation to be added if the field is nonempty; space and/or other material to be added after the punctuation; and the field name. It is permissible for the second part to end with a command that takes an argument, such as \textbf, in which case it will receive the field's value as its argument. By defining a suitable command and using it here you can place material after the field contents as well as before; \parenthesize is an example of this.

The reason that the punctuation and the following space are specified separately is that between them there is a crucial boundary for line breaks. If you put a \linebreak command at the end of a field value, the break point will actually be carried onward to a suitable point after the next bit of punctuation (whose actual value may vary depending on which of the following fields is the first to turn up with a nonempty value).

The meaning of the \parenthesize command, supplied by amsrefs, should be obvious. The meaning of the \PrintAuthors command is a different story. But I don't think it is all that hard to understand. If we have two or three author names which were given separately, and we need to combine them into a conventional name list using commas and the word "and", then it would be nice if we had a command which could take a list of names and Do The Right Thing. And that is just what \PrintAuthors is.

The rkeyval package allows keys to be defined as additive: if the key occurs more than once, each successive value will be concatenated to the previous value, along with a prefix. The setup done by amsrefs for the author field is

```
\DefineAdditiveKey{bib}{author}{\name}
```

This means that if two names are given, as in

```
author={Bertram, A.},
author={Wentworth, R.},
```

then the final value of the author field seen when LaTeX processes the style spec will be

```
\name{Bertram, A.}\name{Wentworth, R.}
```

The transition field in our bibspec example is a dummy field to be used when punctuation or other material must be added at a certain point in the bibliography without regard to the emptiness or non-emptiness of the fields after it. The transition field always tests as non-empty but has no printed content. So when you use it you always get the indicated punctuation and space at the indicated point in the list of fields. If it were the last thing in this bibspec example, it could serve just to put in the final period that is always wanted. But in AMS bibliographies, if a Mathematical Reviews reference is given, it is conventionally printed *after* the final period. Using the transition field as shown here ensures that the final period will be always printed, even when the review field is empty.

Miscellaneous commands provided by the amsrefs package Most of the following commands are helper commands for use in \bibspec statements. The others are intended for use in bibliography data.

\parenthesize This command adds parentheses around its argument. It is useful in \bibspec statements because there is no special provision for adding material after the field value.

`\bibquotes` This command is much like `\parenthesize` but it adds quotes around its argument and it has one other important difference: there are special arrangements to print the closing quote *after* a following comma or similar punctuation (unless the `amsrefs` package is invoked with the `logical-quotes` option, in which case `\bibquotes` puts the closing quote immediately after the quoted material).

`\voltext` The normal definition of this command is `vol.~` to supply the text that precedes a volume number.

`\editiontext` This command produces `ed.` following an edition number. See `\PrintEdition` for more information.

`\pptext` This command is similar in spirit to `\voltext` but more complicated in its implementation. It takes one argument which is expected to contain one or more page numbers or a range of page numbers. The argument is printed with a prefix of "p." if it seems to be a single page number, otherwise with a prefix of "pp.".

`\tsup, \tsub, \tprime` These are for text subscripts and superscripts, with `\tprime` producing a superscript prime symbol. Unlike the standard `\textsuperscript` and `\textsubscript` functions provided by LaTeX, these do not use math mode at all.[3]

`\nopunct` This command causes following punctuation to be omitted if it is added with the internal function `\@addpunct` (which is used throughout the `\bibspec` handling when appending a non-empty field).

`\PrintAuthors` This is a relatively complicated function that tests a list of author names in order to decide whether `\sameauthors` or `\aulist` should be called.

`\aulist` This takes a list of author names in the form

```
\name{Jones, Sam}\name{Smith, John}...
```

and prints them in standard series form with the names uninverted, e.g., "Sam Jones and John Smith", or "Sam Jones, John Smith, and James Taylor".

`\UninvertedNames` This is a lower-level function called by `\aulist`. For documentation see `amsrefs.dtx`.

`\sameauthors` This is a function of one argument. If you use the default set of `\bibspecs` from `amsrefs`, `\sameauthors` is applied to the author name for a given `\bib` command if it matches exactly the author name of the preceding `\bib` command. Change the definition of `\sameauthors` if you don't want to get a bysame dash.

`\bysame` This is a horizontal rule of length 3 em. The default definition of `\sameauthors` prints `\bysame` instead of the author names.

`\PrintEditorsA` This is similar to `\aulist` but adds `(ed.)` following the editor name (or `(eds.)` if applicable).

`\PrintEditorsB` This is like `\PrintEditorsA` but puts parentheses around the entire list of editor names.

`\Plural, \SingularPlural` These are helper functions for use with `\UninvertedNames` that allow you to conditionally print singular or plural forms such as `(ed.)` or `(eds.)` depending on the number of names in the current name list. The definition of `\PrintEditorsA` reads, in part,

```
... (ed\Plural{s}.) ...
```

`\ReviewList` This is similar to `\aulist` but is used for printing (possibly multiple) MR numbers given in the `review` field.

`\inicap` This command applies initial capitalization to its argument.

`\EnglishInitialCaps` This command will call `\inicap` if and only if the language of the current reference is English. If English is the default language, you need to specify

```
language={German},
```

(for example) for non-English references to ensure that nothing will be initial-capped.

`\BibField` This is for more complicated programming tasks such as may be necessary for some bib-specs. It takes one argument, a field name, and yields the contents of that field for the current `\bib` entry.

`\IfEmptyBibField` If one writes

```
\IfEmptyBibField{isbn}{A}{B}
```

then the commands in A will be executed if the `isbn` field is empty, otherwise the commands in B.

`\PrintEdition` If a bibliography entry has

```
edition={2}
```

and the bib-spec used `\PrintEdition` to handle this field, then the edition information will be printed as "2nd ed." — that is, the number is converted to cardinal form and "ed." is added (taken from `\editiontext`).

[3] There is one drawback: If you don't want to get the prime symbol for `\tprime` from the cmsy font, you will need to redefine `\tprime` in some suitable way.

`\CardinalNumeric` This provides the conversion to cardinal number form used by `\PrintEdition`.

`\PrintDate`, `\PrintYear` These functions convert a date in canonical form (ISO 8601) to the form required by the current bibliography style. You can get your preferred date form by redefining these functions or by changing your `\bibspec` statements to use another function of your own devising. The original definition of `\PrintDate` adds parens (as for the year of a journal article in normal AMS style), whereas the `\PrintYear` function simply prints the year without any additional material (as for a book's year of publication in normal AMS style).

`\SerialName` When a journal abbreviation is used as the content of the `journal` field, `\SerialName` will be called with three arguments from the results of the abbreviation lookup: ISSN number, short form, and long form. The default definition of `\SerialName` prints the short form; by changing the definition suitably you can arrange to get the long form instead.

`\PublisherName` This is like `\SerialName` but gets short form, long form, and city as its arguments; the default again is to print the short form.

`\mdash`, `\ndash` These are short forms for `\textemdash` and `\textendash`, recommended instead of the more usual `---` and `--` notation. From the `textcmds` package.

`\cite`, `\cites`, `\citelist`, `\ycite`, `\ocite`, etc. See the section on **Citations**.

Fields recognized by the `\bib` command

The following fields are supported out of the box by the `amsrefs` package. It should be emphasized that this list is not cast in stone. Additional fields can be supported by defining them with commands from the `rkeyval` package and modifying your `\bibspec` statements to take them into account.

address	issn	series
archive	journal	setup
author	label	status
booktitle	language	subtitle
conference	note	title
date	number	translator
doi	organization	type
edition	pages	volume
editor	part	xid
eprint	place	xref
ios	publisher	year
isbn	review	

For some of these a few explanatory remarks are in order.

archive The archive that holds the eprint listed in the `eprint` field.

author This field can be repeated. Multiple author names will be concatenated into a single field value. Names should be given in *Last, First* order.

date This is a generalization of the `year` and `month` fields. Its value should be written in ISO 8601 format, e.g., 1987-06-05; but the day and month are omissible, so this can be used instead of the `year` field.

doi Digital Object Identifier

edition For books. If the value of this field is a simple number, `\bib` will convert it to cardinal form and add "ed.".

editor Like `author`; can be repeated.

eprint Electronic preprint information such as for www.arXiv.org.

ios Institution, organization, or school. Replaces three different BibTeX field names.

isbn International Standard Book Number. Can be repeated.

issn International Standard Serial Number.

language Language of the work. The default language is English; however, this can be changed for an entire bibliography by redefining the following variable:

```
\renewcommand{\biblanguagedefault}{French}
```

The language name should be the printed form, not Babel-style language names, since in principle this field could contain more complicated remarks such as "Russian, with French abstract".

organization The BibTeX documentation says that `institution` should be used for technical reports and `organization` for other entry types, whereas `school` should be used for theses. Having three different field names for these strikes me as overkill, so I introduced `ios` as a substitute. But `organization` is retained as a synonym, for the sake of those who don't like overly cryptic short names.

part This is for a long journal article that is published in separate parts.

place A synonym for address. Or to put it another way, `address` is supported as a BibTeX-compatible synonym for `place`.

review A review number or similar pointer, e.g., for Mathematical Reviews or Zentralblatt. Can be repeated.

setup This is a special field that can be used to give arbitrary commands to be executed at the

beginning of the current `\bib` entry, after all the fields have been read. The idea is that one can alter the formatting of an individual entry through this field, to handle special cases.

status Typically used for notes such as "to appear" or "in preparation" with journal articles.

subtitle Typically used with a multipart journal article to give a subtitle for each part.

translator Like `author`; can be repeated.

url Universal Resource Locator.

xid This is used by a cross-referenced item to pass its identity to child entries that refer to it.

Miscellaneous features

Here are some miscellaneous features that might be of interest.

- Duplicate `\bib` keys are identified on the first LaTeX run and the line number is given. (With `\bibitem` you don't get a warning until the processing of the `.aux` file at the beginning of the second LaTeX run, nor any line number.)
- When a cite key is undefined, the cite command prints the key, not question marks.

Package options

The `amsrefs` package supports the following options. The options that are listed together are mutually exclusive. The options whose name begins with a star are relevant only for BibTeX use and when any such option is changed, the effects will not take hold until after the next BibTeX run.

`?` Information about the `amsrefs` package. If you use the `jpa` option as well, the most obvious effect of this option is that all the available journal and publisher abbreviations are shown on screen (and in the LaTeX log).

`traditional-quotes`, `logical-quotes` This option changes the action of the `\bibquotes` command. When a field is appended after a quoted field, the closing quote is moved if necessary to fall after a comma or similar punctuation instead of before. If the `logical-quotes` option is chosen, ending quotes are not moved.

`sorted-cites`, `non-sorted-cites` Relevant only when numeric cites are in use. Lists of two or more cites are sorted into numerical order.

`compressed-cites`, `non-compressed-cites` Relevant only when numeric cites are in use. Three or more consecutive cite numbers will be converted to range notation (using `\ndash`).

`short-journal-names`, `full-journal-names` These options only work for journals that are specified via abbreviations. Otherwise, of course, you have either the full journal name or the short-form journal name in your data and that's all you've got.

`short-month-names`, `full-month-names` This should be fairly easy to guess.

`initials` Convert authors' first and middle names to initials.

`jpa` Load the standard AMS journal and publisher abbreviations package.

`backrefs` Print back-reference page numbers in the bibliography.

`numeric`, `alphabetic`, `author-year` This option specifies the printed format to be used for cites.

`*sorted`, `*citation-order` This option is passed on to BibTeX (the `amsxport` style) and indicates that when producing the `.bbl` file the entries should be sorted or left in order of first citation, respectively. (The default is sorted.)

Auxiliary packages

The following components of the `amsrefs` package are written in package form for reasons of modularity or to facilitate using them elsewhere.

textcmds This package provides `\mdash`, `\ndash` and other commands to replace the TeX notations for ligatures that are "ligatures of convenience" rather than of esthetics — in effect, all the standard ligature combinations that consist of punctuation characters rather than letters. The fact that the ligatures of convenience lead to quite a bit of trouble in font substitutions and document conversion suggests that they are fundamentally flawed as a markup device. If you still like the convenience of typing two or three hyphens to get a dash instead of some longer sequence, my suggestion is to use the capabilities of your text editor to automatically convert `--` to `\ndash` as you write. You will find a copy of the Emacs setup that I use for this purpose in the `dtx` file for the `textcmds` package.

inicap This package provides the basic `\inicap` function which is called by `\EnglishInitialCaps`.

rkeyval This package provides the more restrictive key-value parser used in processing the contents of a `\bib` command.

ifoption This package provides a way of testing the presence or absence of particular options:

```
\IfOption{jpa}{
  \RequirePackage{amsjpa}[2000/02/02]
}{}
```

The \@ifpackagewith test that is already provided by LaTeX returns false for options that were only switched on with \ExecuteOptions, but not mentioned explicitly in the \usepackage call.

In order for \IfOption to work properly the options must be declared with \DeclareExclusiveOptions or \DeclareBooleanOption and processed with an additional statement \ProcessExclusiveOptions; see the package documentation for further information.

amsjpa This package simply contains a number of abbreviation definitions for the journals and publishers that are most frequently mentioned in AMS bibliographies.

BibTeX exporting — amsxport style

There are four chief functions of BibTeX: selecting items required by a particular document from a database, sorting them, discarding unwanted fields, and applying LaTeX formatting. If the formatting is handled instead entirely on the LaTeX side, and if the author uses editor facilities such as the fine Emacs RefTeX package to handle selecting and sorting, it is a short step to wonder if BibTeX might be approaching superfluity. Since my work on the amsxport BibTeX style involved a certain amount of beating my head against various limitations of BibTeX, I am indeed inclined to advocate abandoning it. (For example, there are only global variables (easy to clobber something inadvertently), and an extraordinarily low ceiling on the number of them; there's no way to get the actual ASCII length of a string (text.length$ counts {\foo} as one character, not six)[4]; and there is no way to call up a given cite individually, e.g., to examine a crossref'd item.) If you convert your existing .bib files to amsrefs format (using the amsxport style) you will never again have to worry about adding braces for proper nouns or extra braces around accented letters when adding material to your database. And if you let RefTeX build your bibliography incrementally while you write, you can get all your cites resolved on the first LaTeX run, not the fourth.

For most users of the amsrefs package it will remain necessary, however, to use BibTeX at least on occasion to convert .bib file data to amsrefs form with the amsxport filter. Therefore it will be of interest to mention a few of its features.

- The supported fields are the standard BibTeX fields plus the additional fields that amsrefs supports (see the list above).
- When sorting names, the amsxport style sorts "von Something" under "Something", not under "von".
- Options can be passed to the bst file; separate bst files are not needed to handle such variations as sorting or not sorting. For example, the *citation-order option of the amsrefs package works by writing \bibcite{[citation-order]} to the .aux file: cite keys that begin with a [character are specially interpreted as options by the amsxport style.
- In preamble strings, ^^M is recognized as an escape sequence to put in newlines at good places. With a BibTeX style that doesn't provide this feature (meaning every one except amsxport, at the present time, to the best of my knowledge), all of the text strings that you supply with @preamble{...} get mushed into a single "paragraph" by BibTeX; there is no good way that I know of to ensure that a real newline gets written to the .bbl file at any given point. (The bad way: put in percent characters or other junk to fill up 72-character lines.)

[4] Well, you can write a loop to chop off one character at a time until the remainder is empty, and I did so, but the fact that I had to do so is the limitation.

Availability

A beta version of the package is currently available at ftp://ftp.ams.org/pub/tex/.

References

[1] Pedro J. Aphalo, *A proposal for citation commands in LaTeX3*, *TUGboat* 18/4 (December 1997), 297–302.

[2] Nelson Beebe, xbtxbst.doc, http://www.math.utah.edu/pub/tex/bibtex/

[3] *Chicago Manual of Style*, 13th edition, 1982, University of Chicago Press.

[4] Dana Jacobsen, bp, *a Perl Bibliography Package* (version 0.2.97 beta, 19 December 1996), http://www.ecst.csuchico.edu/~jacobsd/bib/bp/index.html

[5] Michael Piotrowski, Jens Klöcker, and Jörg Knappen, *Is LaTeX 2ε markup sufficient for scientific articles?* EuroTeX'99 Proceedings (Giessen, Augsburg, 1999), ISBN 1438-9959.

[6] David Rhead, *The "operational requirement" for support of bibliographies*, *TUGboat* 14/4 (December 1993), 425–433.

[7] Reinhard Wonneberger and Frank Mittelbach, *BibTeX reconsidered*, *TUGboat* 12/1 (March 1991), 111–124.

Line breaking and page breaking

Jonathan Fine
203 Coldhams Lane
Cambridge, CB1 3HY
United Kingdom
fine@active-tex.demon.co.uk
http://www.active-tex.demon.co.uk

Abstract

In their seminal paper of 1981, Knuth and Plass described how to apply the method of discrete dynamic programming to the problem of breaking a paragraph into lines. This paper outlines how the same method can be applied to the problem of page make up, or in other words breaking paragraphs into pages. One of the key ideas is that there must be interaction between the line breaking and page breaking routines. It is shown that TEX can, with one important limitation, fully support such interaction.

This article also shows how TEX can, by using a custom paragraph shape and a special horizontal list, suppress hyphenation of the last word on a page.

Introduction

For many years the conventional wisdom has been that TEX is good at breaking lines into paragraphs (and setting mathematics and tables), but that it is not at all good at page make-up. There is some measure of truth in this statement. However, it is the author's view that almost all of the deficiencies arise not from TEX the program, but from the macro packages and other tools used with it.

The main problem considered in this article is that of suppressing hyphenation on the last word of a page. Traditionally, this has been avoided wherever possible, for it breaks the reader's concentration, to have to go to the next page (rather than the next line) to complete a hyphenated word. The solution proposed involves constructing a special horizontal list, and an unusual paragraph shape.

Don Knuth's view of TEX's line-breaking algorithm is well expressed by this passage from *The TEXbook* (page 94):

> The remainder of this chapter explains the details precisely, for people who want to apply TEX in nonstandard ways. TEX's line-breaking algorithm has proved to be general enough to handle a surprising variety of different applications; this, in fact, is probably the most interesting aspect of the whole TEX system. However, every paragraph from now on until the end of the chapter is prefaced by at least one dangerous bend sign, so you may want to learn the following material in easy stages instead of all at once.

and twelve pages of technical details follow. Not all of it used here. The article by Knuth and Plass [4] and Plass' thesis [10] are also well worth consulting.

From ASCII to `dvi`

It will help to have an overview of the process by which TEX converts its input file into typeset pages. In general terms the process is the same for all macro packages, but at each stage each package can use in different ways the capabilities offered by TEX the program.

Here we divide the process into seven stages, namely ASCII, tokens, macros, horizontal list, lines, vertical list and pages.

This section concludes with a discussion of the look-ahead problem, whose solution is an important part of the line-breaking algorithm. The same problem arises in page make-up, and it is the present lack of a solution that has given rise to the view that TEX is not good at page make-up.

ASCII This is the input stream, a text file marked up in some syntax, formal or informal. The input file might contain macro definitions and parameter settings, as well as the text to be typeset. For example, with LATEX the body size is a parameter to the `\documentclass` command.

Strictly speaking, the input stream need not be ASCII. TEX is capable of reading 8-bit input files.

We use ASCII as a convenient shorthand for the input text file.

Tokens Internally, TEX deals with tokens. Category codes control the translation of input characters into tokens.

Traditionally, the ASCII character 'a' is given category code letter, which means that when read it become the token 'the letter a'. The same goes for the other letters. Digits and punctuation are given category other, and so the digit 7 become 'the character 7' when read.

Certain other symbols, such as `{`, `}` and `\`, have special category codes. It is this that gives TEX its familiar 'backslash and braces' input syntax. However, this syntax is not built into TEX the program.

Internally the only tokens that TEX has are character tokens (of various categories), and control sequences. (A control symbol is a control sequence whose name is a single character, usually a non-letter.) The traditional category codes cause the 'eyes of TEX' to convert the sequence of characters `\wibble` to the control sequence whose name is, well, `wibble`. This is done by TEX the program, as part of the process that turns ASCII characters into tokens.

In Active TEX, every input ASCII character is an active character. An active character is rather like a control sequence, in that it has a meaning, and this meaning can be changed at any time. However, its 'name' is the active character 'x', or whatever it is. In plain TEX, the '`~`' character is active.

Active TEX does not use the 'eyes' of TEX the program to form control sequences. Instead, it uses macros and the `\csname` primitive to form control sequences out of the active characters that it receives from the eyes of TEX. This means that it never has to change category codes, in order to achieve special effects, such as verbatim typesetting.

Macros The internal tokens of TEX (or more exactly their meanings) can be divided into two classes, namely the expandable and the unexpandable. Most expandable tokens are macros, and most of the primitive commands of TEX are unexpandable. However, some primitive commands, such as `\ifx`, the other conditional commands and `\csname`, are expandable.

Unexpandable commands do something (in the stomach of TEX the program), while expandable commands and macros control what it is that is done. In plain TEX the '`~`' character, which is active, is defined to be a macro that places a penalty and some glue on the horizontal list. This produces an unbreakable interword space.

Horizontal list TEX would not be able to typeset without commands that placed items on the horizontal list. The internal token 'the letter a' (obtained say by reading an 'a' from the input ASCII file) will place a 'character box' that is the character 'a' in the current font onto the current horizontal list. (This is only in horizontal mode. In math mode it does something else.) The internal token 'the character 9' behaves in the same way.

There are other items that can go on the horizontal list. For this article, we need to know only about glue, penalties and discretionary penalties. Glue is potentially stretchable and shrinkable interword space, while penalties record the undesirability of making a line break at this point.

Discretionary hyphens are hyphens that are optional. The line breaking algorithm can break lines at discretionary hyphens. If the break is taken at a discretionary hyphen, the hyphen appears, and otherwise nothing appears. Discretionary hyphens can be placed onto the horizontal list either explicitly, via the execution of a primitive command, or implicitly, as a result of the hyphenation algorithm.

Lines TEX's line breaking algorithm turns a horizontal list into a sequence of lines. It does this by choosing a sequence of break points in the horizontal list. Most of the time, any glue and penalty items after a chosen break point are discarded. This allows the interword glue to disappear at line breaks.

Normally, TEX breaks the paragraph into lines using the current value of the `\hsize`. However, the `\parshape` parameter allows the width (and offset) of each line to be specified individually.

Vertical list After the paragraph has been broken into lines, TEX places the lines onto the current vertical list. Often, this vertical list is the main vertical list, also known as 'the current page'. Each line of the paragraph is a box (in fact a horizontal box). As well as boxes, a vertical list can contain (vertical) glue and (vertical) penalties. A vertical list can also contain other items, such as insertions, that do not concern us here.

The line breaking algorithm places (vertical) glue between the lines, so that the baseline to baseline distance between the lines is uniform (unless the lines contain exceptionally tall or deep set matter). It also inserts (vertical) penalties between the lines, to aid in the page breaking process. The `\clubpenalty` is the extra penalty for a page break immediately after the first line of a paragraph. The

`\widowpenalty` is the extra penalty for a page break immediately before the last line of a paragraph.

Pages Whenever something is placed on the main vertical list, TEX the program checks to see if it has accumulated enough to break off from it the current page. If it has, then TEX chooses the best of the available break points on the main vertical list. It then calls a special token list, the `\output` routine, to add page numbers and the like to the broken off portion, and to ship it out to the `dvi` file. The `\output` routine is part of the macro package.

The `dvi` file This is the output file produced by TEX the program. As well as recording the placement of every character and every rule on the page, it can contain what are known as `\special` commands. Programs that process `dvi` files can read the specials, and use them as parameters to their actions. For example, a special might request the placement of a graphic.

The look-ahead problem TEX's line breaking algorithm 'looks-ahead' to the end of the paragraph before it makes any decisions as to where the first (or any other) line break occurs. Each line break is, so to speak, considered not by itself but in the context of the other line breaks.

The page breaking algorithm does not perform such a look-ahead. Each page break is considered in isolation, without regard for its consequences later in the document.

At the end of a paragraph, the line-breaking algorithm is called, and it produces lines of text. These lines are then placed on, say, the main vertical list. If enough material has accumulated, the page-breaking algorithm cuts off enough material for one page, and the output routine is called.

Thus, from the end of the paragraph to the calling of the output routine, everything is under the control of TEX the program. During this time neither the user nor the macro programmer has any opportunity to influence TEX's behaviour, other than through the values of parameters and the contents of the horizontal and vertical lists.

TEX's page-breaking algorithm clearly is deficient for complex work. One needs to be able look ahead, when there is floating matter to be placed. Multiple column layout is particularly complicated. There are two aspects to the problem. The first is that an improved algorithm requires more than the information local to the current page. The second is what it does with this information.

This article concentrates on making information available to an improved page-breaking algorithm, but has little to say on the internals of such an algorithm. As in the line-breaking algorithm, the page-breaking algorithm selects one sequence of possibilities from the many presented to it

The line-breaking algorithm has look-ahead. Its context is the current paragraph. To avoid hyphenating the last word on the last line of a page, the algorithm needs to know where that last line will fall (unless it suppresses all hyphenation, and so is done with the problem). Therefore, the page-breaking algorithm will have to feed information back to the line-breaking algorithm.

Once the location of the page break is known, this information can be fed to the line-breaking algorithm (in the form of a custom paragraph shape). Provided a suitable horizontal list is constructed, the algorithm will suppress hyphenation at the required point. How this is done will be shown later in this article.

Discrete dynamic programming

The purpose of this section is to describe those parts of TEX's line-breaking algorithm that are specially relevant to this article. This has two aspects. The first is those features that are relevant to suppression of hyphenation of the last word on some specified line of a paragraph. The second is those features that help us to understand what can be done for global optimisation of page breaks, and for establishing communication between the line-breaking and page-breaking algorithms.

In our simplified model, a horizontal list contains character boxes, glue, penalties and discretionary hyphens. Glue and penalties are what are known as discardable items. They can disappear at a line break. The other items are non-discardable. They will never disappear.

A legal breakpoint is any (finite) penalty, any discretionary hyphen, and any glue item, provided the glue is immediately preceded by something that is non-discardable. For any sequence of breakpoints, there is quantity called the total demerits, that depends on both the chosen breakpoints and on parameters that can be set by the macro programmer.

For example, when breaking at a penalty, the amount of the penalty is part of the sum that is the total demerits. Similarly, the `\hyphenpenalty` and `\exhyphenpenalty` parameters are the contributions made by discretionary and explicit hyphens respectively. If the line had been set loose or tight (shorter or longer than its optimum width) then a badness for the line contributes to the total demerits.

Of all the possible sequences of breakpoints for a given paragraph, TEX chooses one that has the smallest possible value for the total demerits. It does not choose the breakpoints line by line, or in other words locally. The breakpoints are chosen with a view to the whole paragraph, or in other words globally.

The way in which it does this is interesting, because in general there are so many possible sequences of breakpoints, that it is impossible for them to be considered individually. The method used is known as discrete dynamic programming. This method allows the last line of a paragraph to 'communicate' with the first. (It is communication not in the sense of sending a message, but in the sense of being part a common larger whole.)

To save time, TEX tries first to make the paragraph without using any hyphenation. The parameters `\pretolerance` and `\tolerance` are limits on how bad a line can be respectively before and after hyphenation. For simplicity, we will assume that hyphenation is alway tried, say because the pretolerance is zero.

A sequence of breakpoints is said to be feasible if no line has badness exceeding the tolerance. The line-breaking algorithm considers only feasible sequences of break points. For formal reasons, the end of the paragraph is considered to be a breakpoint. It is, after all, the end of a line.

The formula the algorithm uses to compute the total demerits has the following useful property. Suppose an optimal sequence of breakpoints is selected, and say lines 5 to 9 are of the horizontal list are considered in isolation from the remainder of the horizontal list. The optimal sequence of breakpoints for the whole paragraph, when restricted to the isolated lines, is also optimal for the line-breaking problem represented by the isolated problem. This is called the property of locality. It is a property of the formula for total demerits.

Discrete dynamic programming, as applied to line-breaking, consists of the following. Start at the beginning of the paragraph. Calculate the feasible breakpoints for the end of the first line. From these breakpoints calculate the feasible breakpoints for the end of the second line. We now prune the list of feasible sequences of breakpoints. If two or more sequences end the second line at the same point, keep only the best one. (If several are joint first, keep only one.) For each of the remaining two-line breakpoint sequences, compute all the feasible extensions to three-line sequences, and prune as before.

As this process continues, so the number of both feasible and locally optimal sequences will in general grow. However, the growth will not be too rapid. Consider the spread in the location of the breakpoint that is the end of, say, the nth line. If the first n lines contain as little set matter as is possible, then we get one location in the horizontal list. If they contain as much as is possible, we get another. This is the spread. It is roughly linear in n. The number of breakpoints in this spread is the number of locally optimal breakpoints that the algorithm must carry along to the $n + 1$ stage.

This analysis limits the running time to of the order of n^2. However, we can do better. When the spread gets large, it will cover the the length of a whole line, and so some of the calculations for $n + 2$ will have been done as part of $n + 1$. This also shows why using a custom paragraph shape is computationally expensive. There is no longer such a sharing of computations between lines.

The line-breaking and the page-breaking algorithms have a certain amount in common. This is how Don Knuth puts it in *The TEXbook* (page 100):

> TEX breaks lists of lines into pages by computing badness ratings and penalties, more or less as it does when breaking paragraphs into lines. But pages are made up one at a time and removed from TEX's memory; there is no looking ahead to see how one page break will affect the next one. In other words, TEX uses a special method to find the optimum breakpoints for the lines in an entire paragraph, but it doesn't attempt to find the optimum breakpoints for the pages in an entire document. The computer doesn't have enough high-speed memory capacity to remember the contents of several pages, so TEX simply chooses each page break as best it can, by a process of "local" rather than "global" optimisation.

The situation is not impossible though. In Appendix D (page 400) Don Knuth writes:

> An output routine can also write notes on a file, based on what occurs in a manuscript. A two-pass system can be devised where TEX simply gathers information during the first pass; the actual typesetting can be done during the second pass, using `\read` to recover information that was written during the first.

Provided sufficient information can be gathered in the first pass, it can then be presented to TEX's line-breaking algorithm, or some other program, so that an optimal choice can be made from amongst

those which are feasible. The second pass can then do the actual typesetting.

Avoiding 'last-word' hyphenation

In this section we explain how a suitable horizontal list and paragraph shape specification taken together will cause the line-breaking algorithm to suppress hyphenation of the word at the end of some specified line.

The basic idea is quite simple. Hyphenation places matter on the next line. Indeed, this is the very purpose of hyphenation. However, if the next line is not long enough to hold even the smallest fragment of a word, then the word at the end of the previous line will not be hyphenated. (It is possible for a very long word to be hyphenated two or more times. Each hyphenation point is a legitimate breakpoint.)

> The sixth line has zero width. This prevents hyphenation of the word at the end of the fifth line. This is because when a word is hyphenated, part of the word is placed on the next line. (This is the
>
> very purpose of hyphenation.) A special sequence of items of glue and penalties is placed between words. This allows the interword glue to span the zero-width line.

Figure 1: Example of suppressed hyphenation

We can achieve this effect on say the fifth line by making the width of the sixth line equal to zero. This however creates a problem. If we use ordinary glue between words, then between any two words there will be only one breakpoint, namely the glue that was between the words. For some word to be allowed to occur at the end of the fifth line, it must be followed by a special piece of 'glue', that is capable of spanning the zero width sixth line.

Recall that in our simplified model (which is all we need), breaks can occur at penalties, at discretionary hyphens, and at glue that is preceded by something that is not discardable. To go further, we need to understand exactly what happens at a line break.

According to *The TEXbook* (page 97):

> When a line break actually does occur, TEX removes all discardable items that follow the break, until coming to something non-discardable, or until coming to another chosen breakpoint. For example, a sequence of glue and penalty items will vanish as a unit, if no boxes intervene, unless the optimum breakpoint sequence includes one or more of the penalties.

In other words, most of the time discardable items are discarded, but any (finite) penalties are allowed to be part of the breakpoint sequence, if that is what the algorithm decided to do. In other words, when moving on to the next feasible breakpoint, it has something of a free choice in the discarding of discardables.

Therefore, each piece of 'glue' between words will have to contain two legitimate break points, as well as an ordinary piece of interword glue. The way to get this is to place two penalties of zero, followed by the ordinary interword glue. (The penalty for breaking at glue preceded by a non-discardable, such as a word, is zero. Thus, in ordinary cases we get the same behaviour as before.)

Something similar arises in ordinary practice. Sometimes a line is deliberately left short, say because the next word is too long to fit on the line, and it cannot be hyphenated. The standard way to achieve this is to insert `\hfil \break` in the line. The `\break` is just a shorthand for a penalty of zero, and the `\hfil` is glue that stretches to fill the line. When the line has zero width, no glue is required to fill it.

In August 1999, the author posted to the newsgroup `comp.text.tex` example code that suppressed hyphenation. A lively debate followed, but not until the author came to write this article did he discover, to his shock and horror, that the code he posted last summer did not work in many cases. In the first version of this paper, his solution had an unnecessary but harmless zero-width piece of glue between the two penalties. This was not noticed until after the paper had been refereed. Clearly, some of us have something to learn about penalties and glue.

The `echowords` environment

Figure 1 shows the result of applying the methods of the previous section. So that there are many hyphens, a discretionary hyphen has been placed between adjacent letters of a word. The spaces between words contribute, as described in the previous section, two penalties of zero and an ordinary interword space. Although it is clearly possible to construct such a horizontal list by hand, doing so is laborious and prone to error.

Instead, the author has used Active TEX to simplify the form of the example's input. In fact the author wrote

```
\begin{hyphdemo}{5}
The sixth line has zero width.
[...]
zero-width line.
\end{hyphdemo}
```

and it is the purpose of this section and the next to describe the macros that were used.

By way of an example, this section contains the complete source for a LaTeX environment that echoes its content to the console, word by word. Here a word is a maximal sequence of visible characters. White space separates words. The next section gives the listing for the `hyphdemo` environment.

In the header of the source for this article the author wrote:

```
\RequirePackage{atcode}[2000/07/22]
\RequirePackage{atlatex}[2000/07/22]
```

and this loads two Active TeX macro packages.

The first of these packages defines a programming environment in which it much easier to write macros that make extensive use of active characters. It has other advantages, which make it useful even when writing macros that have no special features. One uses the command `\@code` to enter the environment, and `]]` to exit it. Later in this section there will be examples of the input syntax and programming style for the package.

The first section also define macros for making all ASCII characters active, and giving them standard meanings. More exactly, the active character 'a' is a macro whose expansion is the control sequence `\active:lcletter` followed be the active character 'a'. This apparent recursion is very useful, for it allows each active character to know its own identity. By letting the prefix control sequence `\active:lcletter` be `\string`, for example, a character can be made to typeset itself.

The second of these packages defines a new command, `\active:latex`, that makes it possible for LaTeX macro programmers to access the facilities of Active TeX. The `hyphdemo` environment above is coded using this command, within an `\@code` programming environment.

First we enter the atcode environment. What follows is the most general way, for it does not assume that the ASCII character `@` has category code letter. There must be a space before the `@`, and no space after the `code`.

```
\csname @code\endcsname
```

Now all ASCII characters are active, and we are in the atcode environment. Here, control sequences do not need to be prefixed by a backslash. You can use a backslash as a prefix, but it is neater to omit the backslash whenever possible. Strings, however, have to be enclosed in double quote marks. Here is an example. (This semi-colon is not syntactic sugar. It tells atcode that it is safe to release the tokens it has been accumulating. Semi-colons within braces, however, are syntactic sugar. The same goes for commas.)

```
message { "Hello world" } ;
```

Next we set up a shorthand feature. This allows us to type `.digit` instead of `active:digit`, and so on. From now on we will drop the backslash before control sequence names, in both text and in atcode source. We will also drop the `active:` prefix. Thus, `active:latex` is `.latex`.

```
def active:prefix { "active" } ;
```

The macro `get.word` parses `the.word` from the input stream, and calls `do.word` to process it. It depends on `init.get.word`, whose value will be set by the calling context. It also relies on some system macros that have not yet been described. In the atcode environment white space is ignored, unless it is part of a string or the like.

```
def get.word
{
  begingroup ; aftergroup do.word ;
    init.get.word ;
    .suspend.white.space ;
    let .suspend .end.xdef ;
    xdef the.word { iffalse } fi ;
}
```

Here is the definition of the `echowords` environment. It provides an example of the `.latex` command. The `]]` closes the atcode environment. All is as it was before except that the macro `get.word` and the environment `echowords` have been defined.

```
newenvironment { "echowords" }
{
  begingroup ;
    let .lcletter get.word ;
    let .ucletter get.word ;
    let .digit get.word ;
    (default) ; let .symbol get.word ;
    let .rs relax ; let .re relax ;
    let .re-sp relax ;
    let !  relax ; let |D09 relax ;
    let init.get.word .string.visible ;
    def do.word
      { message { the.word } } ;
    .latex ;  //  must come last
}
{ endgroup } ;
```

```
]] %% now back to the usual catcodes
```

We will now explain what is going on. First, the `.latex` command. This command opens a group, in which all ASCII characters are active. It looks for a line that begins with the (active) characters `\end{`. When it finds this, it closes the all-active group, and pretends that it had read the `\end{` with LaTeX's normal category codes. Thus, the first and last of the input lines

```
\begin{echowords}
 These words are echoed,
 one by one.
\end{echowords}
```

are processed by the begin and end commands of the `echowords` environment. The two lines in the middle are read and processed with all characters active, and with the values set by the environment.

We simplify slightly. To avoid needlessly filling TeX's macro processor (the mouth) with a long list of tokens, this looking for the `\end{` is done on a line by line basis. However, this makes no difference in practice.

The first three assignment commands tell Active TeX how to process letters (upper- and lowercase) and digits. Symbols are rather different. Most if not all of the time, all lowercase letters are dealt with by the same rules. The same goes for uppercase letters, and for digits. It often happens, however, that each symbol has a specific meaning. For example, in the atcode environment, each symbol has a distinct meaning of its own.

For this reason, Active TeX uses the concept of symbol sets. Within its realm, it 'owns' all the active symbols. (This is done in a way that does not interfere with their use outside of its realm.) Instead of directly assigning a value to a symbol, one selects a symbol set, perhaps of one's own creation. The owner of a symbol set is free to change the meaning of symbols in that set. For as long as that symbol set is selected, for almost all practical purposes changing the meaning of a symbol in a set is the same as changing the meaning of the active symbol itself. However, when a different symbol set is selected, the meaning of all the symbols changes to those of the newly selected set.

Parentheses, as above, are used to select a symbol set. The `(default)` symbol set is part of the atcode package, and in it every symbol expands to the control sequence `.symbol`, followed the active symbol itself. Thus the line of code:

```
(default) ; let .symbol get.word ;
```

causes all symbols to call `get.word`. In short, all visible characters are to call the `get.word` command we just defined.

The `.latex` command 'owns' the active end-of-line character. Only when one knows for sure that it is safe to do so, should one change its meaning. It is used by `.latex` to inspect the next line for the `\end{` characters.

At the start and end of each non-blank input line, `.latex` generates `.rs` and `.re` events. Blank input lines generate the `.rs-re` event. These events are control sequences, whose values can be set by the macro programmer. Here we are setting them to do nothing. (One can think of the visible characters as similarly being events, but this time with parameters.)

We have now initialised all the ASCII characters except for space and tab. The next line sets them both to `relax`. (The construction `|ABC` generates a character whose category code is hexadecimal `A`, and whose character code is hexadecimal `BC`. Thus, `|D09` is active tab.)

The low-level events (reading a character from the input stream) have now been dealt with. They create higher level events, namely the initialisation of the parsing of a word, and the processing of the word once parsed. The `.string.visible` macro is a low-level system macro that causes all visible characters to behave as if they were characters of category code other. This system macro by-passes the symbol set mechanism. It runs quicker, but must be used with care.

We are almost done. There are some commands in `get.word` that need explanation. The command `.suspend.white.space` cause the active form of the white space characters (space, tab and end-of-line) to expand to `.suspend` followed by the active white space character. This should only be done within a group, which is closed by white space. The parsing of a word is exactly such a context.

Finally, the `atlatex` package contains a helper macro that is very useful for closing a 'flying `xdef`'. Here is its definition.

```
def .end.xdef
  { iffalse { fi ; } ; endgroup } ;
```

To conclude, we reconsider the `get.word` macro.

```
def get.word
{
  begingroup ; aftergroup do.word ;
    init.get.word ;
    .suspend.white.space ;
    let .suspend .end.xdef ;
    xdef the.word { iffalse } fi ;
```

```
}
```

It opens a group. After the group, we call `do.word`. The variable part of the initialisation routine, namely `init.get.word` is defined to make all visible characters 'other'. Thus, when the `xdef` which closes the macro executes, it simply accumulates visible characters in `the.word`. (The `iffalse` is a hack that allows a macro to 'contain' unbalanced braces.) The two suspend commands cause white space to close the `xdef`, and thereby trigger the processing of the word.

The reader may find it instructive to run these macros with `tracingall` on, and examine the resulting log file.

The `hyphdemo` environment

Our next example is more substantial. The suppression of hyphenation on the last line of a page requires the construction of a fairly special horizontal list. Here is the sequence of penalties and glue that is to be placed between words. (We use `hd` as a two letter prefix for 'hyphenation demonstration'.) But first we enter the atcode environment.

```
\csname @code\endcsname
def hd:iwspace
{
  unskip ;
  penalty "0 " ; penalty "0 " ;
  ~ ;  // ordinary interword space
}
```

The `unskip` is in case we get two spaces in a row. This is not rigorous, but in the context it is good enough. Then we put down two penalties, which allows `hd:iwspace` to span a blank line. Finally, we put down an ordinary piece of interword glue. In Active TEX, `~` produces an ordinary space character.

So that we get lots of hyphens, we will place a discretionary hyphen between adjacent letters in a word. To do this, we use a variant of the `get.word` command. This macro applies `string` to the first character in the word. Each subsequent character is then responsible for putting down a discretionary hyphen before `string`ing itself. To avoid hyphenating just before punctuation at the end of a word, symbols do not insert a discretionary hyphen.

```
def hd:get.word { get.word ; string } ;
def hd:init.get.word
{
  def .lcletter { \- ; string } ;
  let .ucletter .lcletter ;
  let .digit .lcletter ;
  let .symbol string ;
}
```

The `hyphdemo` environment takes a single parameter, namely the number of the line, at the end of which hyphenation is to be suppressed. This parameter controls the construction of a custom `parshape`, which will be coded later. If the parameter is zero, no suppression is offered.

The parameters encourage hyphenation. The large value of the line penalty is to stop the linebreaking from making the lines very loose, just so it can get the reward (negative penalty) for the additional hyphen.

```
newenvironment { "hyphdemo" } [1]
{
  par ;
  begingroup ;
    hyphenpenalty "-100" ;
    doublehyphendemerits "0 " ;
    linepenalty "200 " ;
    leftskip "2pc " ;
    rightskip leftskip ;
    hd:set.parshape { #1 } ;

    let .lcletter hd:get.word ;
    let .ucletter .lcletter ;
    let .digit .lcletter ;
    (default) ; let .symbol .lcletter ;
    let !  hd:iwspace ;
    let |D09 !  ; let .re !  ;
    let init.get.word hd:init.get.word ;
    def .re-sp { par } ;
    def do.word { the.word } ;
    .latex ; //  don't forget this
}
{ par ; endgroup }
```

The remainder of the definition of this environment sets up the conditions for the parsing and processing of words, in much the same way as in the previous section. Note that `let do.word the.word` would be very wrong. This would cause the macro to continually process the value of `the.word` that was current at the start of the environment.

The difficult part of setting the parameters is to feed the parameters to TEX's `parshape` primitive. It takes $2n + 1$ parameters, where n is the number of lines, whose width we are specifying. These are TEX number and dimension parameters, and not macro or token parameters. We use aftergroup accumulation to build up this list. Scratch counters are used to hold the values of parameters whose values have to be calculated.

```
def hd:set.parshape #1
{
```

```
    count@ #1~ ;
    ifcase count@ ,
    else
      advance count@ tw@ ;
      dimen z@ leftskip ;
      advance dimen z@ rightskip ;
      def temp { z@ ; hsize } ;
      begingroup ;
        aftergroup parshape ;
        aftergroup count@ ;
        count@ #1~ ;
        loop ; ifnum count@ > z@ ,
         aftergroup temp ;
         advance count@ m@ne ;
        repeat ;
      endgroup ;
      //  accumulated tokens released here
      z@ ; dimen z@ ;  //  zero-width line
      temp ;  //  remaining lines normal
    fi
  }
```

Some words of explanation. If the parameter is zero, we do nothing, otherwise we store in scratch registers the number of lines in the parshape, and the sum of the `leftskip` and the `rightskip` (the actual width of a 'zero-width' line). We then store in `temp` the specification for a normal line. A process of 'aftergroup accumulation' is then used to build up the `parshape` along with its parameters. (The `@code` environment uses the same method to gain its power. The asterisks problem in Appendix D of *The TEXbook* (page 373) is a simpler example of this.)

Outside the group, `count@` is $n+2$, the number of lines in the paragraph shape. Inside the group it is set to n, which is the number of lines before the zero width line. (Grouping ensures that the two values do not interfere with each other.) The loop accumulates n `temp` tokens. At the end of the group, the accumulated aftergroup tokens re-appear. The zero-width line and the final `temp` complete the paragraph specification.

All that remains now is to close the atcode environment.

```
  ]]
```

Flexible paragraphs

Sometimes it is helpful, for purposes of page make-up, to set a paragraph slightly longer or shorter than is optimal. For this purpose TEX provides the `looseness` parameter. Negative values of looseness can be thought of as tightness. If the looseness is 1, then TEX will try to make the paragraph one line longer than it would otherwise. Traditionally, in the TEX world, looseness is applied by hand, when fine-tuning the document for publication.

Let us consider now how it might be done automatically. Ahead of time, we will not know how long we will want the paragraph to be. Nor will we know where the paragraph appears on the page, and thus which custom paragraph shape to use. Therefore, we shall consider all possibilities.

We might find, for example, that a given paragraph can be set using 9, 10 or 11 lines. We might also find that when 9 lines are used, we can suppress hyphenation at the line breaks 4, 6, 7 and 8. (We are lucky if we can suppress hyphenation early on in a paragraph.)

The following table represents this data about 9-line versions of the paragraph. Each line gives a way of breaking the paragraph, and the number pointed to by the arrow is the total demerits for the optimal way of so breaking the paragraph. Similar tables can be constructed for the 10 and 11 line versions of the paragraph. Such a report, on the flexibility of all paragraphs in the document, will be the input for the global optimisation algorithm considered in the next section.

```
4+5->3489
6+3->2748
7+2->2956
9->2413
```

The reader may object that to prepare such a report, even by computer, will take a long time. This may be true, but the situation is not hopeless. First, if the document is in its final form, this report need be prepared only once. The page-breaking algorithm, by design, requires no knowledge of the document, other than this report.

Second, even if the document is not in its final form, changes are likely to be confined to a small proportion of its paragraphs. Matters can be configured, provided macros have been written with this in mind, so that fresh report data need only be generated for the paragraphs that have changed. This is probably something that could be done in real time on the entry-level hardware available today.

What is true is that much more time will be spent on trial paragraph breaking, to generate the report, as is spent on breaking the paragraphs for the final triumphant globally optimised version. The same is true, however, of the line-breaking algorithm.

Indeed, the two are yet more similar than this. Discrete dynamic programming depends on the principle of local optimality, which is a property of the

formula for total demerits. One consequence of this property is the following. If an optimal sequence of breakpoints takes in the feasible breakpoints A and B, then over the range A to B this sequence is also optimal for this local form of the problem.

Now suppose A is 'sufficiently distant' from the start of the paragraph, and that B is 'sufficiently distant' from A. Here, 'sufficiently distant' means that there is a feasible (but not necessarily optimal) sequence of breakpoints linking the two points. If enough set matter of random width intervenes between the two points, the concept has its intuitive meaning. In these circumstances the line-breaking algorithm will find an optimal sequence of breakpoints between A and B. It will do this whether or not this is part of the finally chosen optimal sequence.

Thus, the line-breaking algorithm finds not only the best breaking for the whole paragraph, but also for a great many portions of the paragraph. In the same way, any worthwhile discrete dynamic programming solution to the problem of global optimisation will consider most or all possible feasible ways of breaking the paragraphs that constitute the document. The strength of Knuth and Plass's algorithm is not that it runs quickly in abstract, but that the running time is roughly linear, rather than quadratic, in the size of the problem. Because of linearity, in time hardware will be able to catch up, even if the problems are large.

Global optimisation

This section describes briefly how a report on paragraph, as in the previous section, can be used as the input for a global optimisation process. For simplicity, we assume that we are setting straight text on a grid, and that hyphenation is to be suppressed on the last word of each page. We also assume that no paragraph is longer than a page, or in other words, that it cannot span two page breaks.

First, it is convenient to recast each paragraph's report into the following form. We give the possibilities in order first of the number of lines before the potential page boundary, and then in order of the number after. Thus, a fragment of paragraph's report might look like the following. (The values for total demerits are fictional, and are chosen to make the rest of the exposition clearer. The right hand column will be explained later.)

```
4+5->4050 ; wibble 4050, 0 ;
4+6->4060 ; wibble 4060, 1 ;
4+7->4070 ; wibble 4070, 2 ;
          ; wobble ;
5+4->5040 ; wibble 5040, 0 ;
5+5->5050 ; wibble 5050, 1 ;
```

Given such a sequence of paragraph reports, and the requirement that there be, say, exactly 12 lines on each page, there is an associated optimisation problem. First, for each paragraph report choose one of its entries. Call this a selection (of paragraphs). Write the selection in the form

$$(5) + (3) + (4 + 7) + (5 + 2) + 10 + \ldots$$

and say that the selection is feasible if, when summing from left to right, successive exact multiples of 12 are reached during the progress of the sum. The above selection is feasible (as far as it goes). For every feasible selection, define the grand total demerits to be the sum of demerits associated with the terms of the form $(a+b)$. Thus, the $(4+7)$ terms contributes 4070 to the grand total demerits.

The optimisation problem is to find a feasible selection that minimises the grand-total demerits. This is one way (there are many others) of defining a global optimisation for the line and page breaks of a document. If such a problem is to be solved using discrete dynamic programming, the global optimisation data might take a more elaborate form, but the general structure will be the same. (The interested reader might wish to look at how TEX's line-breaking algorithm supplies demerits for adjacent lines whose looseness is visually incompatible. It is done by providing each partial problem with a context.)

It is both interesting and fortunate that the global problem, as described above, can be solved using TEX's line-breaking algorithm. It is a matter of 'putting the book on its side', and thinking of each line as a 'word' in a paragraph. The problem is to construct a suitable list of boxes, glue and penalties. So that we can get nice diagrams, we will let one pica represent one line.

Discardable items can vanish at line breaks, and with trickery this allows the problem to be solved. Consider for example the sequence of horizontal list items,

```
penalty "4070 " ;
kern "-2pc " ;
noalign {} ;
kern "2pc " ;
```

Kerns are discardable items. If the line break is taken at the penalty, the first kern will be discarded. The `noalign` is non-discardable, and it prevents the second kern from being discarded. Thus, if the penalty is not a break-point, the kerns cancel, but if the penalty is a break point, it effectively inserts a kern of two pica.

Denote such a sequence of horizontal list items by `wibble 4070, 2;`, and let `wobble` represent a `kern` by one pica. This procedure translates the sequence of paragraph reports into a horizontal list. When the line-breaking algorithm is applied to this list, with a `hsize` of twelve pica, the result is a global optimisation of the line and page breaks. If we are not typesetting on a grid, then some 'interword glue' (representing interline glue) should be added to the above construction.

As mentioned earlier, the line-breaking algorithm introduce penalties between the lines, in order to help TEX's page-breaking algorithm. These prevent, or discourage, page breaks just after the first line of a paragraph, and just before the last line. In the algorithm described here, the potential page break is part of the paragraph's specification. The club penalty thus becomes an extra demerit charged for paragraph specifications of the form $(1+n)$, and similarly the widow penalty applies to $(n+1)$.

Although there are some difficulties of a technical nature in implementing such a solution, there is a more fundamental problem. In 1989 [6], when Don Knuth released version 3 of TEX, he introduced several new primitives. One of them, the `\badness`, records the badness of the box that was most recently constructed. Thus, this quantity is made available to the macro programmer. Sadly, he did not at the same time introduce `\totaldemerits`, and so there is no ready access to this quantity.

Summary and conclusions

When Don Knuth announced [8] in 1990 that his work on developing TEX had come to an end, he pointed out that improved macro packages could be added on the input side, and improved device drivers added on the output side. This article shows that ten years after the event, there is still plenty of room for improvement on the macro package side. (However, the lack of a `\totaldemerits` command is unfortunate.)

The problem of suppressing hyphenation at the end of a page is relatively simple, particularly if a macro package such as Active TEX is used to construct the horizontal list. What has not been discussed is how to rearrange the resulting sequence of lines, so that the blank amongst them can be discarded. In abstract this is not difficult, but in the context of an existing macro package one may find assumptions being made that are inconsistent with this goal.

The problem of page make-up is much harder, particularly where there are multiple columns and floating material. TEX was not designed to do such work, although it can readily typeset the paragraphs that will go into the pages. As in shown in the previous section, it is possible to use the line-breaking algorithm to solve simple page make-up problems. For more complicated problems, an external program might be more suitable.

TEX is not good at complicated page makeup, but that is no reason to 'improve' it. Complicated page make-up was never a design goal of TEX. Instead, TEX can be used to feed paragraphs and paragraph reports to an external make-up program. Such can be thought of as an improved device driver, in the same way as Active TEX is intended to be an improved macro package.

Postscript

Prior to the TUG 2000 meeting I sent an earlier version of this paper to Don Knuth, and invited his comments. He told me that I should cite and read Michael Plass's thesis [10]. The citing is done, and I hope soon to read this work. He also says that he cannot add `\totaldemerits`, as that would mean changing TEX and suggests instead that I approach the authors of extensions to TEX.

In his essay on the errors of TEX, [7] Don Knuth wrote:

> Of course I don't mean to imply that all problems of computational typography have been solved. Far from it! There are still countless important issues to be studied, relating especially to the many classes of documents that go far beyond what I ever intended TEX to handle.

I hope that this article shows that a few judicious extensions to TEX will produce a new system that can handle well many new classes of documents, and that even TEX can make a fair attempt at doing the job. What seems to be required, above all, is an understanding of the problem, and the development of suitable algorithms. From then on, the programming of the extensions should be straightforward.

The article by Frank Mittelbach in these proceedings addresses a different aspect of the page makeup problem. His concern is with placement of floats. Combining his work with mine, even at the level of algorithms, is already a challenge. When it comes to implementation, the widespread use of active space characters is likely to present LATEX with many problems. Assumptions about category codes are built into its input syntax.

So much for output. On the input side the papers by David Carlisle and by Pedro Palao Gostanza in these proceedings have significant overlap with

this paper. I am delighted that others are taking steps in the direction of making all characters active. However, we now have three incompatible systems of values for the meaning of active letters and digits.

Active TEX provides a powerful and effective programming environment, especially for defining active characters. Without such a device, the programmer has to resort to ad hoc tricks, time and time again. For example, of the 1,936 lines of xmltex (v0.07), exactly 194 contain the string `catcode`. By contrast, of the 6,361 lines of my `sgmlbase` package (v0.00), exactly 23 contain the string `catcode`. Perhaps if Carlisle had used Active TEX, his work would have been easier.

For this area to flourish, standards are required. Without standards, incompatible versions of the basic macros will be re-invented. Application programmers will then have to work harder, to cope with this unhelpful diversity. There is also be the danger of schisms within the community.

To understand this, imagine what life would be like if there we used incompatible mechanisms for register allocation (`\newcount` and the like). In *The TEXbook* (page 346), Don Knuth addressed precisely this problem:

> *Allocation of registers.* The second major part of the `plain.tex` file provides a foundation on which systems of independently developed macros can coexist peacefully without interfering in their usage of registers.

We need the same for active characters. The packages `atcode.sty` and `atlatex.sty` have been written to be a fixed point that opens this area to the plain and LATEX macro programmer. They differ only in a small but significant detail (colon instead of prefix is used to segment the name space) from the version announced at TUG 1999.

I offer these packages to the community, and hope for the rapid and widespread adoption of a standard for the use of active characters. I would of course prefer that my own macros were the standard, but more important both to me and to the community as a whole, I believe, is that a standard acceptable to all is adopted.

References

[1] David P. Carlisle, *xmltex: A non validating (and not 100% conforming) namespace aware XML parser implemented in TEX*, these proceedings

[2] Jonathan Fine, Active TEX and the DOT input syntax, *TUGboat*, **20**, (1999), 248–254

[3] Pedro Palao Gostanza, Fast scanners and self-parsing in TEX, these proceedings

[4] Donald E. Knuth, Michael F. Plass, Breaking paragraphs into lines, *Software — Practice and Experience*, **11** (1981), 1119–1184.

[5] Donald E. Knuth, *The TEXbook*, Addison-Wesley (1984).

[6] ———, The new versions of TEX and METAFONT, *TUGboat*, **10** (3) (1989), 325–328

[7] ———, The Errors of TEX, *Software — Practice and Experience*, **19** (1989), 605–685; reprinted with additions and corrections as Chapter 10 of *Literate Programming*.

[8] ———, The future of TEX and METAFONT, *TUGboat*, **11** (4) (1990), 489.

[9] Frank Mittelbach, *Formatting documents with floats*, these proceedings

[10] Michael F. Plass, *Optimal Pagination Techniques for Automatic Typesetting Systems*, Ph.D. thesis, Stanford University (1981). Published also as Xerox Palo Alto Research Center report ISL-81-1

PassiveTEX: from XML to PDF

Michel Goossens
CERN, IT Division, CH-1211 Genève 23, Switzerland
michel.goossens@cern.ch

Sebastian Rahtz
Oxford University Computing Services, Oxford, United Kingdom
sebastian.rahtz@oucs.ox.ac.uk

Abstract

This article introduces PassiveTEX, a library of TEX macros based on `xmltex`, that processes XML documents containing XSL formatting objects and generates PDF or DVI output. We show examples of typesetting XML sources marked up using the TEI, DocBook, and MathML DTDs.

Introduction

New information becomes available on the Internet every day, and increasingly material is becoming available in XML. However, when one wants to print the information on the screen one is faced with several shortcomings, the more important being the low typographic quality of the result or the problem of how to serialize a 'tree' of pages onto a linear output medium. In this article we explain how a source document marked up in XML can be typeset nicely by a two-step procedure that combines a transformation of the XML source into XSL formatting objects and the direct interpretation of these XML objects with Sebastian Rahtz' PassiveTEX, a variant of TEX based on David Carlisle's `xmltex` [4].

The choice of TEX as the typesetting engine is justified by the fact that TEX handles mathematics in a natural way, can manage several languages simultaneously (hyphenation, typographic rules, multiple encodings and alphabets, right-left typesetting, etc.), and has a good model for marking up complex tabular information. Moreover, TEX can easily handle very long documents, such as large software manuals, or bills for hundreds of thousands of telephone subscribers.

In the first part of this article we use as an example some literary texts, including poems by the French poet Paul Verlaine and the Russian poet Alexander Blok, marked up according to the TEI DTD (*Text Encoding Initiative* [3], Lite version). In the second part we show an example of a more technical document, including a couple of simple math formulae, using the DocBook [14] and MathML [17] DTDs.

The present article was prepared in XML using TEI markup. It was not actually typeset with PassiveTEX, but transformed into LATEX with an XSLT transformation stylesheet and typeset using the `ltugproc` LATEX class file.

XSL formatting objects

The *Extensible Stylesheet Language* (XSL) is a language for describing page designs. For any given class of arbitrarily structured XML documents or data files, an XSL stylesheet allows you to express how the content contained in the XML should be presented. The stylesheet indicates how source elements should be styled, laid out, and paginated in some presentation form. In this article we only address issues related to high quality *typesetting* (using TEX). Our XML examples can be transformed into other representations (see [7] for more discussion), in particular HTML, and XSL stylesheets targeting HTML are available for both the TEI [12] and DocBook [15] markup schemes.

An XSL processor reads an XML source document together with an XSL stylesheet, and produces the presentation of the given XML content according to the instructions in the stylesheet. The preparation of the presentation form proceeds in two steps. First, a *result tree* is constructed from the XML input source tree (*tree transformation* step). Second, the result tree is interpreted to produce the formatted results (*formatting* step), that can be presented on output media such as a computer screen, a WAP display, on paper, in speech, etc.

The result tree can be very different from the structure of the source tree, since parts from the

input can be deleted from the result tree, while supplementary information (e.g., table of contents) can be generated. The tree transformation step also has to add the information necessary to format the result tree.

The nodes of the result tree are *formatting objects*, whose semantics are expressed in terms of a catalog of formatting object classes defined in the XSL Specification [24]. They denote typographic abstractions such as page, paragraph, table, list, etc. Fine control over the presentation of these abstractions is provided by a set of *formatting properties*, such as those controlling alignments, color, fonts, spacing, writing-mode, hyphenation etc. XSL offers a rich range of formatting objects as one of its aims is to cover the semantic functionality of both the *Document Style Semantics and Specification Language* (DSSSL) [10] and *Cascading Stylesheets* (CSS) [16] models as completely as possible.

PassiveTEX: implement XSL using TEX

In the previous section we explained how an input XML document can be transformed into a new XML document containing a result tree of XSL formatting objects. How do we go about rendering those objects with a real typesetting engine, and getting high-quality printout? One way is to use TEX itself. PassiveTEX is a library of TEX macros which can typeset the XSL formatting objects. In particular, PassiveTEX combined with the pdfTEX variant of TEX generates high-quality PDF files in a single operation. PassiveTEX allows one to choose TEX as the formatter of choice for XML, while hiding the details of its operation from the user.

PassiveTEX derives from and builds on `xmltex` [4], a TEX package by David Carlisle, providing the core XML parser and UTF8 handler. Ideas and TEX code are also inherited from earlier work by Sebastian Rahtz on his JadeTEX package, which implemented the output of the Jade DSSSL processor's TEX backend, and already used a catalogue of Unicode/TEX mappings.

Issues in using TEX Since PassiveTEX is based on TEX one can rapidly develop and test new high level code, knowing that TEX will take care of the typesetting details. Moreover, TEX is a well-understood, robust, and free page formatter, where good support for fonts, graphics inclusion, hyperlinks, etc., is already available. One can also profit from TEX's mature handling of language issues, including hyphenation, and its high-quality math rendering. Moreover, pdfTEX allows direct generation of high-quality PDF.

There are, however, some drawbacks associated with using TEX as typesetting engine. Firstly, we are constrained to use TEX's page makeup model, and have to force the XSL formatting objects to fit that model, which is not always straightforward. Moreover, since PassiveTEX is actually layered over LATEX it is too easy to allow things to fall through and take LATEX defaults.

On a more practical level, TEX macro writing is obscure and difficult and thus the system is not transparent for most programmers. And, last but not least, TEX is large and monolithic, and unsuited to embedding in other applications.

Users of the system with a TEX background can find it confusing to understand that TEX no longer does all the work, which is now split betwen the stylesheet and the formatter; the four important points to remember are:

- No explicit use is made of LATEX's high-level constructs, in particular there are no sections, lists, cross-references, bibliographies, etc.
- all vertical and horizontal space is explicit in the specification;
- page and line breaking is left to TEX: the rest is up to the stylesheet;
- XSL formatting objects use XML syntax, so that the underlying character set is Unicode. By default, entities are mapped to their Unicode position.

PassiveTEX will switch to using the Unicode-based TEX variant soon (Omega [8]), to handle non-Latin material more naturally.

Two extensions are needed for practical use. The first is support of MathML; this is largely in place (it is simply passed through as-is by an XSL stylesheet), but needs some tuning. In particular, the intricacies of equation numbers remain to be dealt with properly. Second, we need to support *Scalable Vector Graphics* (SVG) [21] XML code somehow (e.g., by direct interpretation, translation into MetaPost [9], or pre-processing). Moreover, SVG fragments need to be recognized directly to perform in-line graphical functions (e.g., setting text at an angle). No work at all has been done on this.

Support for XSL formatting objects Not all parts of the XSL specification are fully supported. The XSL formatting objects which are implemented fairly well cover the page specifications, blocks, inline sequences, lists, graphics inclusion, floats, font properties, and links. Not so well implemented at present are all the table properties, and object margins, borders, and padding. In order to properly implement these, we have to put every paragraph into

an explicit TEX box 'just in case', and this becomes slow and clumsy. Tables, too, are treated rather differently in XSL, with many properties assigned at the cell level.

Parts of the specification that are not yet implemented at all include bidi handling, backgrounds, aural properties, and a large number of assorted properties.

A small example The set of XSL stylesheets for the TEI DTD (`tei-fo` [12]) specify how the various XML elements can be transformed into XSL formatting objects (see Section 7.6.6 of [7] for a more detailed discussion). Without further comments, the following code example shows the transformations for a paragraph (`p` element, lines 1 to 6) and emphasized material (`emph` element, lines 7 to 11).

```
1  <xsl:template match="p">
2    <fo:block indent-start="10pt"
3              space-before="12pt">
4      <xsl:apply-templates/>
5    </fo:block>
6  </xsl:template>
7  <xsl:template match="emph">
8    <fo:inline-sequence font-style="italic">
9      <xsl:apply-templates/>
10   </fo:inline-sequence>
11 </xsl:template>
```

A few words about `xmltex` As PassiveTEX uses `xmltex` to read and interpret the XML source it is appropriate to say a few words about it. `xmltex` is a non validating (and not 100% conforming) namespace-aware XML parser implemented in TEX. `xmltex` reads the encoding declaration and tries to implement it; in particular it handles UTF8. It reads the DTD subset and expands entities properly, it uses namespaces to load modules of TEX code to process elements (for instance for MathML, see the following example). `xmltex` offers limited access to child and parent elements to guide the interpretation process.

The workhorse of an `xmltex` package file is the `\XMLelement` command, whose four parameters indicate how TEX will handle a given element, its attributes and its content. The following example shows how a few simple elements of the MathML namespace are handled.

```
1  \DeclareNamespace{m}
2    {http://www.w3.org/1998/Math/MathML}
3
4  \XMLelement{m:math}
5    {}
6    {\begin{equation}}
7    {\end{equation}}
8
9  \XMLelement{m:mn}
10   {}
11   {\xmlgrab}
12   {\mathrm{#1}}
13
14 \XMLelement{m:msqrt}
15   {}
16   {\xmlgrab}
17   {\sqrt{#1}}
```

Lines 1 and 2 declare the prefix `m` to be used for referring to elements in the given namespace. Lines 4 to 7 declare that the start and end tag of a `math` element are transformed into the opening and closing of an `equation` environment. Lines 9 to 12 define how a MathML `mn` (number) element is typeset. Its contents are stored (grabbed, see line 11) and then injected as parameter of a `\mathrm` to be typeset in roman. Similarly, lines 14 to 17 show how a square root (`msqrt`) element and its content are transformed into LATEX's `\sqrt` and its argument.

To fine-tune the output, `xmltex` supports processing instructions to manipulate TEX formatting directly.

The biggest problem at present in using `xmltex` to write PassiveTEX is that it uses TEX grouping to make sure each element is handled in the right namespace. Because TEX does not allow for nested grouping, each element is caught in a group; while `\aftergroup` can help a bit, this is the first hurdle for anyone trying to extend the package.

An example of TEI markup typeset with PassiveTEX

In this section we use an XML source file that contains a collection of literature of the French poet Paul Verlaine, the Russian poet Alexander Blok, the English novelist Thomas Hardy, and the Finnish author Aleksis Kivi. It is marked up according to the TEI Lite DTD. The master file `poemstei-utf8` follows:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE TEI.2 SYSTEM "teixlite.dtd" [
3  <!ENTITY dash "&#x2010;">
4  <!ENTITY mdash "&#x2014;">
5  <!ENTITY oelig "&#x0153;">
6  <!ENTITY Verlaine SYSTEM "Verlainetei-utf8.xml">
7  <!ENTITY Blok     SYSTEM "Bloktei-utf8.xml">
8  <!ENTITY Hardy    SYSTEM "beyes.xml">
9  <!ENTITY Kihlaus  SYSTEM "kihlaus.xml">
10 ]>
11 <TEI.2>
12 <teiHeader type="text" status="new">
13 <fileDesc>
14 <titleStmt>
15 <title>Various languages and scripts</title>
16 <author>Collected by Michel Goossens</author>
17 <respStmt>
18 <resp>Transcription in TEI.2 markup</resp>
19 <name>Michel Goossens</name></respStmt>
20 </titleStmt>
21 <publicationStmt>
22 <distributor>Distributed by mg</distributor>
```

```
23 </publicationStmt>
24 </fileDesc>
25 <profileDesc>
26 <langUsage default="NO">
27 <language id="EN">English</language>
28 <language id="FI">Finnish</language>
29 <language id="FR">French</language>
30 <language id="RU">Russian</language>
31 </langUsage>
32 </profileDesc>
33 </teiHeader>
34   <text>
35 <front>
36 <titlePage>
37 <docTitle>
38 <titlePart>Languages and scripts</titlePart>
39 </docTitle>
40 <docAuthor>Collected by Michel Goossens</docAuthor>
41 <docDate>July 2000</docDate>
42 </titlePage>
43 </front>
44 <body>
45 &Verlaine;
46 &Blok;
47 &Hardy;
48 &Kihlaus;
49 </body>
50 </text>
51 </TEI.2>
```

This XML source document uses the UTF8 encoding, since we want to mix various alphabets (in this case Latin and Cyrillic). UTF8 is a Unicode-based encoding [13] that can encode each character in Unicode's base plane and its sixteen extended planes (allowing for more than one million characters) with at most three bytes. Documents that only contain ASCII can be coded using one byte per character. Parts of the XML source of the foreign language documents (`Verlainetei-utf8.xml`, defined on line 6 and input on line 45; and also `Bloktei-utf8.xml` defined on line 7 and input on line 46) are shown in Figure 1.

To typeset this document we first transform the XML into XSL formatting objects using an XSLT stylesheet, as explained previously. Then, this intermediate XML file is handled by `xmltex` and PassiveTEX. In what follows we use James Clark's `xt` XSLT processor ([5], see also Section 7.6.4 of [7]), but any conforming XSLT processor should be able to handle these transformations.

```
xt infile.xml style.xsl fotex.xml
latex fotex.tex
```

The file `style.xsl` is an XSL stylesheet that contains the XSLT transformations needed to transform the input XML file `infile.xml` into XSL formatting objects. The resulting intermediate XML document `fotex.xml` can be interpreted by applications that can render XSL formatting objects. In our case we use PassiveTEX.

Below we show part of the beginning of the XML file `fotex.xml` that corresponds to the first page of the collection of poems of Verlaine transformed using Sebastian Rahtz' TEI XSL stylesheets [12].

```
1    ...
2  <fo:flow flow-name="xsl-region-body"
3           font-family="Times Roman"
4           font-size="10pt">
5
6    <fo:block text-align="center" space-after="8pt">
7      <fo:block font-size="16pt">
8        <fo:inline font-weight="bold">
9          Poems in various languages and scripts
10       </fo:inline>
11     </fo:block>
12     <fo:block font-size="14pt">
13       <fo:inline font-style="italic">
14         Collected by Michel Goossens</fo:inline>
15     </fo:block>
16     <fo:block font-size="14pt">
17       July 2000
18     </fo:block>
19   </fo:block>
20
21   <fo:block keep-with-next.within-page="always"
22             id="N115"
23             text-align="start"
24             font-size="14pt"
25             text-indent="-3em"
26             font-weight="bold"
27             space-after="3pt"
28             space-before.optimum="9pt">
29     1. Paul Verlaine, F\^etes galantes (1869)
30   </fo:block>
31
32   <fo:block keep-with-next.within-page="always"
33             id="N128"
34             text-align="start"
35             font-size="12pt"
36             font-weight="bold"
37             space-after="2pt"
38             space-before.optimum="4pt"
39             text-indent="-3em">
40   1.1. <fo:inline font-style="italic">
41          Clair de lune</fo:inline>
42   </fo:block>
43
44   <fo:block text-align="start"
45             space-before.optimum="4pt"
46             space-after.optimum="4pt">
47     <fo:block space-before.optimum="0pt"
48               space-after.optimum="0pt">
49       Votre \^ame est un paysage choisi
50     </fo:block>
51       ...
52   </fo:block>
53   ...
54 </fo flow>
```

At the beginning of the file (not shown) the various page masters are defined (margins, height, width, etc.), and the static page information (running headers, footers) is initialized. Then, the construction of formatted output pages starts. We show what happens at the start of page 1 (compare with

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <div1 lang="FR">
 3 <head>Paul Verlaine, Fêtes galantes (1869)</head>
 4 <!- ++++++++++++++++++++++++++++++++++++++++++++++++++++++ ->
 5 <div2>
 6 <head><hi rend="ital">Clair de lune</hi></head>
 7 <!- ++++++++++++++++++++++++++++++++++++++++++++++++++++++ ->
 8 <lg type="stanza" org="uniform" sample="complete" part="N">
 9 <l>Votre âme est un paysage choisi</l>
10 <l>Que vont charmant masques et bergamasques,</l>
11 <l>Jouant du luth, et dansant, et quasi</l>
12 <l>Tristes sous leurs déguisements fantasques. </l>
13 </lg>
14   ...
15 </div2>
16 <!- ++++++++++++++++++++++++++++++++++++++++++++++++++++++ ->
17 <div2>
18 <head><hi rend="ital">Les indolents</hi></head>
19 <lg type="stanza" org="uniform" sample="complete" part="N">
20 <l>Bah ! malgré les destins jaloux,</l>
21 <l>Mourons ensemble, voulez-vous ?</l>
22 <l>&dash; La proposition est rare.</l>
23 </lg>
24   ...
25 </div2>
26 <!- ++++++++++++++++++++++++++++++++++++++++++++++++++++++ ->
27 <div2>
28 <head><hi rend="ital">Colloque sentimental</hi></head>
29 <lg type="stanza" org="uniform" sample="complete" part="N">
30 <l>Dans le vieux parc solitaire et glacé,</l>
31 <l>Deux formes ont tout à l'heure passé.</l>
32 </lg>
33 <lg>
34 <l>Leurs yeux sont morts et leurs lèvres sont molles,</l>
35 <l>Et l'on entend à peine leurs paroles.</l>
36 </lg>
37   ...
38 </div2>
39 </div1>
```

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <div1 lang="RU">
 3 <head>Александр Блок, Стихи о прекрасной даме (1901-1902)</head>
 4 <!- ++++++++++++++++++++++++++++++++++++++++++++++++++++++ ->
 5 <div2>
 6 <head>«Отдых напрасен. Дорога крута...» (28 декабря 1901)</head>
 7 <lg type="stanza" org="uniform" sample="complete" part="M">
 8 <l>Отдых напрасен. Дорога крута.</l>
 9 <l>Вечер прекрасен. Стучу в ворота.</l>
10 </lg><lg>
11 <l> Дольнему стуку чужда и строга,</l>
12 <l>Ты рассыпаешь кругом жемчуга.</l>
13 </lg>
14   ...
15 </div2>
16 <!- ++++++++++++++++++++++++++++++++++++++++++++++++++++++ ->
17 <div2>
18 <head>«Я вышел. Медленно сходили...»
19   (С.-Петербург, 25 января 1901)</head>
20 <lg type="stanza" org="uniform" sample="complete" part="M">
21 <l>Я вышел. Медленно сходили</l>
22 <l>На землю сумерки зимы.</l>
23 <l>Минувших дней младые были</l>
24 <l>Пришли доверчиво из тьмы...</l>
25 </lg>
26   ...
27 </div2>
28 <!- ++++++++++++++++++++++++++++++++++++++++++++++++++++++ ->
29 <div2>
30 <head>«Ветер принес издалка...» (29 января 1901)</head>
31 <lg type="stanza" org="uniform" sample="complete" part="M">
32 <l>Ветер принес издалка</l>
33 <l>Песни весенней намек,</l>
34 <l>Где-то светло и глубоко</l>
35 <l>Неба открылся клочок.</l>
36 </lg>
37   ...
38 </div2>
39 </div1>
```

Figure 1: Partial input source of French (left) and Russian (right) documents

the output shown in the upper left corner of Figure 3). The `fo:flow` element contains the material that has to be typeset and cut into individual pages by the typesetting engine. We see that the default document font is Times Roman at 10 pt.

Lines 6 to 19 typeset a block of centered material. In particular, lines 7–11 take care of the document title that is typeset in 16 point bold, lines 12–15 correspond to the author's name typeset in 14 point italic, while lines 16–18 typeset the date in 14 point roman. All this information is obtained from the content of the `titlepage` element of the XML master source file `poemstei-utf8`, shown previously (lines 36–42).

The remaining text is in the external files `Verlainetei-utf8.xml` (entity reference on line 46 of `poemstei-utf8`, which contains the poems of Paul Verlaine in French, a fragment of which is seen in the left half of Figure 1), and `Bloktei-utf8.xml` (entity reference on line 47 of `poemstei-utf8`, which contains the poems of Alexander Blok in Russian, a fragment of which is seen in the right half of Figure 1).

The author and the title of the collection of poems (specified as the content of the `head` element of a `div1` element on line 3 in Figure 1) are handled on lines 21 to 30 in the formatting objects output, where a block for the first level title is created. Note that the section number (1. on line 29) is generated by the XSLT application. Similarly, the title of an individual poem (specified as the content of the `head` element of a `div2` element in Figure 1, e.g., on line 6) is handled on lines 32 to 42. Once more, the number (1.1 on line 40) is generated automatically. Then, the stanza and the lines inside the stanza of the poem (the `lg` and `l` elements in Figure 1) are treated. For instance, the stanza on lines 8–13 in Figure 1 corresponds to the block on lines 44–52 in the formatting objects output, in particular the input line 8 becomes the block 47–50. More details about the various XSL formatting objects and their attributes can be found in the XSLT Recommendation [24].

The second stage of the process (typesetting the XSL formatting objects with LaTeX) uses `xmltex` that is called by running LaTeX on the intermediate file `fotex.tex` containing the following three lines:

```
1 \def\xmlfile{fotex.xml}
2   \input xmltex.tex
3   \end{document}
```

Line 1 specifies the file that has to be interpreted by `xmltex`, before it takes control on line 2. The

result of treating the file `poemstei-utf8.xml` with the two-step procedure outlined is shown in Figure 2. Figure 3 shows, for comparison, the result of typesetting the same XML source file directly with `xmltex`, where we had to define for each XML element and its attributes how it should be rendered by LaTeX.

Besides PassiveTEX, FOP [2], originally developped by James Tauber, and presently supported by the Apache Project, is another application, written in Java, that transforms XSL formatting objects into PDF. Because it uses Java, it is very portable, but it does not yet handle many aspects of fine typography as well as TEX. Commercial products targeting the same task of producing excellent typographic output from XSL formatting objects have also been announced (*RenderX*, ArborText's *Epic*), but are not yet available.

DocBook markup and some words about mathematics

Up to now we have discussed a document marked up according to the Text Encoding Initiative (TEI) schema. In this section we show that other XML markup schemes can also be used, in particular, DocBook [14] which we will combine with MathML [17], W3C's XML DTD for mathematics.

DocBook is an XML (originally SGML) DTD which is especially well-suited for marking up technical documents. It is used extensively for preparing software reference guides and computer equipment manuals.

The DocBook DTD or schema contains hundreds of elements to mark up clearly and explicitly the different components of an electronic document (book, article, reference guide, etc.), not only displaying its hierarchical structure but also indicating the semantic meaning of the various elements. The structure of the DTD is optimized to allow for customization, thus making it relatively straightforward to add or eliminate certain elements or attributes, to change the content model for certain structural groups, or to restrict the value that given attributes can take.

An example of DocBook markup We are not going to cover the DocBook vocabulary in any detail (see Walsh's book [14] or his DocBook Web page). We shall only consider an example where we used the DocBook markup schema. A file `dbxmml.xml`, whose first part is reproduced below, contains various elements of the DocBook vocabulary. It will allow us to test the typesetting paradigm based on PassiveTEX outlined for the TEI schema previously.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <!DOCTYPE article SYSTEM
3  "/usr/local/share/docbookxml/4.11/docbookx.dtd"
4  [
5   <!ENTITY LaTeX "LaTeX">
6   <!ENTITY TeX   "TeX">
7   <!ENTITY PTeX  "PassiveTeX">
8   <!ENTITY xmltex
9             "<application>xmltex</application>">
10  <!ENTITY % equation.content "(math+)">
11  <!ENTITY % inlineequation.content "(math+)">
12  <!ENTITY % mathml
13            SYSTEM "mathmldtd/mathml2.dtd">
14  <!ATTLIST math xmlns CDATA #FIXED
15            "http://www.w3.org/1998/Math/MathML">
16 <!-- load MathML -->
17 %mathml;
18 ]>
19 <article>
20 <articleinfo>
21 <title>A Docbook document featuring a
22        few formulae</title>
23 <author><forename>Michel</forename>
24         <surname>Goossens</surname>
25 </author>
26 <pubdate>Wednesday, 18 July 2000</pubdate>
27 <abstract>
28 <para>
29 This XML document is marked up according the
30 DocBook schema It shows a few elements of the
31 DocBook vocabulary, as well as a couple of
32 examples of mathematical expressions where
33 we used MathML markup.
34 </para>
35 </abstract>
36 </articleinfo>
37 <section>
38 <title>The Docbook model</title>
39 <para>
40 DocBook <xref role="bib" linkend="docbook"
41 endterm="docbookab"/> is an XML model
42 <xref role="bib" linkend="rec-xml"
43 endterm="rec-xmlab"/> for marking up technical
44 documents.  It is particularly well-suited for
45 software reference guides and computer
46 equipment manuals.
47 </para>
```

The DTD internal subset (lines 4–18) contain the general entity definitions of `LaTeX`, `PassiveTeX`, `TeX`, and `xmltex` (lines 5 to 9), and on lines 10 and 11 parameter entity definitions specifying the content models for the `equation.content` and `inlineequation.content` elements, that can contain one or more `math` elements. Lines 12–13 define the `mathml` parameter entity that defines the system file containing the MathML DTD, which is loaded on line 17. Line 14 specifies the namespace of the `math` element. The remaining lines have markup using the DocBook vocabulary, and their meaning should be rather straightforward to understand.

An example of MathML presentation markup [17] follows.

Literature in various languages and scripts

Collected by Michel Goossens
July 2000

1. Paul Verlaine, Fêtes galantes (1869)

1.1. *Clair de lune*

Votre âme est un paysage choisi
Que vont charmant masques et bergamasques,
Jouant du luth, et dansant, et quasi
Tristes sous leurs déguisements fantasques.

Tout en chantant sur le mode mineur
L'amour vainqueur et la vie opportune,
Ils n'ont pas l'air de croire à leur bonheur
Et leur chanson se mêle au clair de lune,

Au calme clair de lune triste et beau,
Qui fait rêver les oiseaux dans les arbres
Et sangloter d'extase les jets d'eau,
Les grands jets d'eau sveltes parmi les marbres.

1.2. *Les indolents*

Bah ! malgré les destins jaloux,
Mourons ensemble, voulez-vous ?
- La proposition est rare.

- Le rare est bon. Donc mourons
Comme dans les Décamérons.
- Hi ! Hi ! Hi ! quel amant bizarre !

- Bizarre, je ne sais. Amant
Irréprochable, assurément.
Si vous voulez, mourons ensemble ?

- Monsieur, vous raillez mieux encor
Que vous n'aimez, et parlez d'or ; :
Mais taisons-nous, si bon vous semble ! »

Si bien que ce soir-là Tircis
Et Dorimène, à deux assis
Non loin de deux sylvains hilares,

Eurent l'inexpiable tort
D'ajourner une exquise mort.
Hi ! hi ! hi ! les amants bizarres !

2. Александр Блок, Стихи о прекрасной даме (1901-1902)

2.1. «Отдых наш прекрасен. Дорога круга...» (28 декабря 1901)

Отдых наш прекрасен. Дорога круга.
Вечер прекрасен. Стучу в ворота.

Дольнему стуку чужда и строга,
Ты рассыпаешь кругом жемчуга.

Терем высок, и заря замерла.
Красная тайна у входа легла.

Кто поджигал на заре терема,
Что воздвигала Царевна Сама?

Каждый конек на узорной резьбе
Красное пламя бросает к тебе.

Купол стремится в лазурную высь.
Синие окна румянцем зажглись.

Все колокольные звоны гудят.
Залит весной беззакатный наряд.

Ты ли меня на закатах ждала?
Терем зажгла? Ворота отперла?

2.2. «Я вышел. Медленно сходили...» (С.-Петербург, 25 января 1901)

Я вышел. Медленно сходили
На землю сумерки зимы.
Минувших дней младые были
Пришли доверчиво из тьмы...

Пришли и встали за плечами,
И пели с ветром о весне...
И тихими я шел шагами,
Провидя вечность в глубине.

О, лучших дней живые были!
Под вашу песнь из глубины
На землю сумерки сходили
И вечности вставали сны!..

2.3. «Ветер принес издалёка...» (29 января 1901)

Ветер принес издалёка
Песни весенней намек,
Где-то светло и глубоко
Неба открылся клочок.

В этой бездонной лазури,
В сумерках близкой весны
Плакали зимние бури,

1

Collected by Michel Goossens

Реяли звездные сны.
Робко, темно и глубоко
Плакали струны мои.
Ветер принес издалёка
Звучные песни твои.

3. From *A Pair Of Blue Eyes*, by Thomas Hardy

Elfride Swancourt was a girl whose emotions lay very near the surface. Their nature more precisely, and as modified by the creeping hours of time, was known only to those who watched the circumstances of her history.

Personally, she was the combination of very interesting particulars, whose rarity, however, lay in the combination itself rather than in the individual elements combined. As a matter of fact, you did not see the form and substance of her features when conversing with her; and this charming power of preventing a material study of her lineaments by an interlocutor, originated not in the cloaking effect of a well-formed manner (for her manner was childish and scarcely formed), but in the attractive crudeness of the remarks themselves. She had lived all her life in retirement—the *monstrari gigito* of idle men had not flattered her, and at the age of nineteen or twenty she was no further on in social consciousness than an urban young lady of fifteen.

One point in her, however, you did notice: that was her eyes. In them was seen a sublimation of all of her; it was not necessary to look further: there she lived.

These eyes were blue; blue as autumn distance—blue as the blue we see between the retreating mouldings of hills and woody slopes on a sunny September morning.

A misty and shady blue, that had no beginning or surface, and was looked *into* rather than *at*.

4. From Aleksis Kivi's play *Kihlaus*

EENOKKI Vai Herrojen-Eevan. Tuliko mies juonikkaaksi, hurjapäiseksi, villityksi, tai mistä kieppasi hän tämän rohkeuden?

JOOSEPPI Preivin kirjoitti Herrojen-Eeva tällä tavalla mestarilleni eilen illalla: "Kraatari Aapeli! Lyhyesti tahdon tietä antaa, että, jos Jumala niin on sallinut, niin valmis olen kohta tulemaan aviopuolisoksenne. Tulkaat minua kyydittämään täältä luoksenne; sillä herrat jätän minä tämän ijankaikkisen pilkun päällä. Sen olen nyt vahvasti tykänäni päättänyt." Niin seisoi preivissä.

EENOKKI Ja mestaris tämän luettuansa rupesi tuumailemaan ankarasti?

JOOSEPPI Pasteeraili, pasteeraili edestakaisin permanolla, raappien niskatukkaansa.

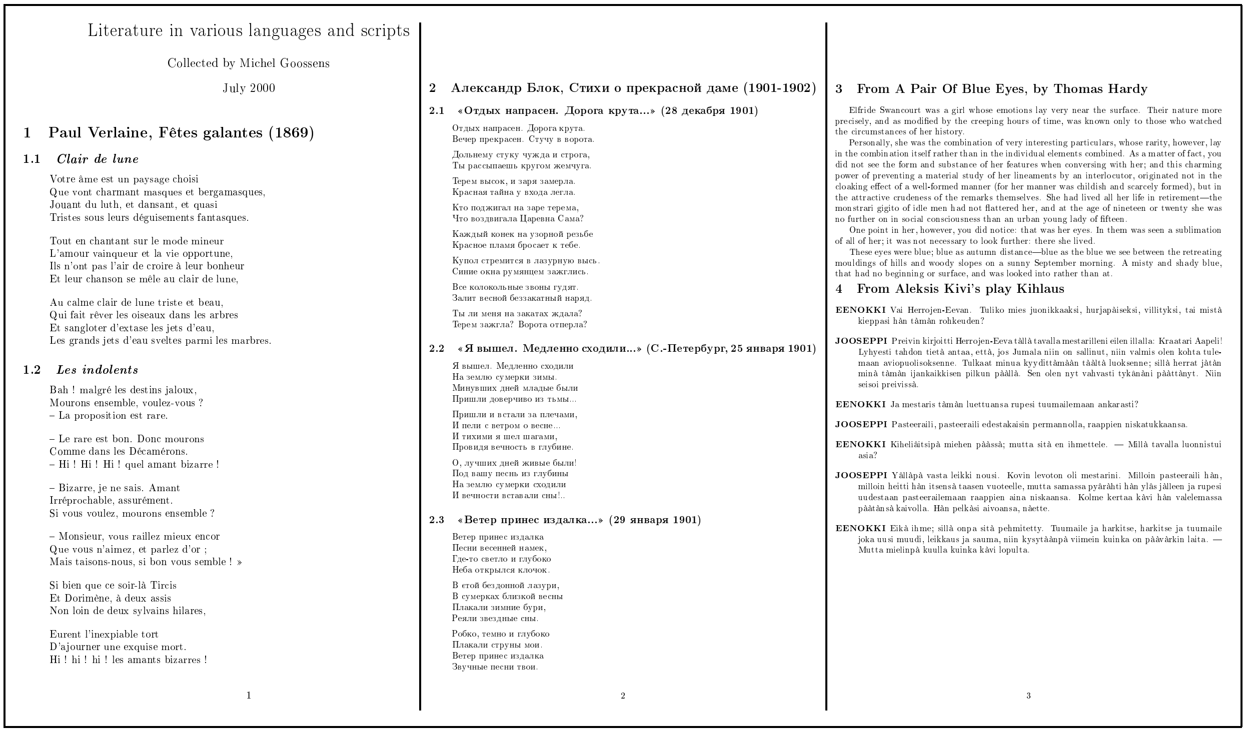
EENOKKI Kiheliäitsipä miehen päässä; mutta sitä en ihmettele. — Millä tavalla luonnistui asia?

JOOSEPPI Yälläpä vasta leikki nousi. Kovin levoton oli mestarini. Milloin pasteeraili hän, milloin heitti hän itsensä taasen vuoteelle, mutta samassa pyärähti hän ylös jälleen ja rupesi uudestaan pasteerailemaan raappien ainа niskaansa. Kolme kertaa kävi hän valelemassa päätänsä kaivolla. Hän pelkäsi aivoansa, näette.

EENOKKI Eikä ihme; sillä onpa sitä pehmietty. Tuumaile ja harkitse, harkitse ja tuumaile joka uusi muudi, leikkaus ja sauma, niin kysytäänpä viimein kuinka on pääväkin laita. — Mutta mielinpä kuulla kuinka kävi lopulta.

2

Figure 2: Typesetting a collection of literature marked up according to the TEI, using XSL formatting objects


Literature in various languages and scripts

Collected by Michel Goossens

July 2000

1 Paul Verlaine, Fêtes galantes (1869)

1.1 *Clair de lune*

Votre âme est un paysage choisi
Que vont charmant masques et bergamasques,
Jouant du luth, et dansant, et quasi
Tristes sous leurs déguisements fantasques.

Tout en chantant sur le mode mineur
L'amour vainqueur et la vie opportune,
Ils n'ont pas l'air de croire à leur bonheur
Et leur chanson se mêle au clair de lune,

Au calme clair de lune triste et beau,
Qui fait rêver les oiseaux dans les arbres
Et sangloter d'extase les jets d'eau,
Les grands jets d'eau sveltes parmi les marbres.

1.2 *Les indolents*

Bah ! malgré les destins jaloux,
Mourons ensemble, voulez-vous ?
– La proposition est rare.

– Le rare est bon. Donc mourons
Comme dans les Décamérons.
– Hi ! Hi ! Hi ! quel amant bizarre !

– Bizarre, je ne sais. Amant
Irréprochable, assurément.
Si vous voulez, mourons ensemble ?

– Monsieur, vous raillez mieux encor
Que vous n'aimez, et parlez d'or ;
Mais taisons-nous, si bon vous semble ! »

Si bien que ce soir-là Tircis
Et Dorimène, à deux assis
Non loin de deux sylvains hilares,

Eurent l'inexpiable tort
D'ajourner une exquise mort.
Hi ! hi ! hi ! les amants bizarres !

1

2 Александр Блок, Стихи о прекрасной даме (1901-1902)

2.1 «Отдых напрасен. Дорога крута...» (28 декабря 1901)

Отдых напрасен. Дорога крута.
Вечер прекрасен. Стучу в ворота.

Дольнему стуку чужда и строга,
Ты рассыпаешь кругом жемчуга.

Терем высок, и заря замерла.
Красная тайна у входа легла.

Кто поджигал на заре терема,
Что воздвигала Царевна Сама?

Каждый конек на узорной резьбе
Красное пламя бросает к тебе.

Купол стремится в лазурную высь.
Синие окна румянцем зажглись.

Все колокольные звоны гудят.
Залит весной беззакатный наряд.

Ты ли меня на закатах ждала?
Терем зажгла? Ворота отперла?

2.2 «Я вышел. Медленно сходили...» (С.-Петербург, 25 января 1901)

Я вышел. Медленно сходили
На землю сумерки зимы.
Минувших дней младые были
Пришли доверчиво из тьмы...

Пришли и встали за плечами,
И пели с ветром о весне...
И тихими я шел шагами,
Провидя вечность в глубине.

О, лучших дней живые были!
Под вашу песнь из глубины
На землю сумерки сходили
И вечности вставали сны!..

2.3 «Ветер принес издалка...» (29 января 1901)

Ветер принес издалка
Песни весенней намек,
Где-то светло и глубоко
Неба открылся клочок.

В єтой бездонной лазури,
В сумерках близкой весны
Плакали зимние бури,
Реяли звездные сны.

Робко, темно и глубоко
Плакали струны мои.
Ветер принес издалка
Звучные песни твои.

2

3 From A Pair Of Blue Eyes, by Thomas Hardy

Elfride Swancourt was a girl whose emotions lay very near the surface. Their nature more precisely, and as modified by the creeping hours of time, was known only to those who watched the circumstances of her history.

Personally, she was the combination of very interesting particulars, whose rarity, however, lay in the combination itself rather than in the individual elements combined. As a matter of fact, you did not see the form and substance of her features when conversing with her; and this charming power of preventing a material study of her lineaments by an interlocutor, originated not in the cloaking effect of a well-formed manner (for her manner was childish and scarcely formed), but in the attractive crudeness of the remarks themselves. She had lived all her life in retirement—the monstrari gigito of idle men had not flattered her, and at the age of nineteen or twenty she was no further on in social consciousness than an urban young lady of fifteen.

One point in her, however, you did notice: that was her eyes. In them was seen a sublimation of all of her; it was not necessary to look further: there she lived.

These eyes were blue; blue as autumn distance—blue as the blue we see between the retreating mouldings of hills and woody slopes on a sunny September morning. A misty and shady blue, that had no beginning or surface, and was looked into rather than at.

4 From Aleksis Kivi's play Kihlaus

EENOKKI Vai Herrojen-Eevan. Tuliko mies juonikkaaksi, hurjapàiseksi, villityksi, tai mistà kieppasi hàn tàmàn rohkeuden?

JOOSEPPI Preivin kirjoitti Herrojen-Eeva tàllà tavalla mestarilleni eilen illalla: Kraatari Aapeli! Lyhyesti tahdon tietà antaa, ettà, jos Jumala niin on sallinut, niin valmis olen kohta tulemaan aviopuolisoksenne. Tulkaat minua kyydittàmààn tààltà luoksenne; sillà herrat jàtàn minà tàmàn ijankaikkisen pilkun pààllà. Sen olen nyt vahvasti tykânàni pààttànyt. Niin seisoi preivissà.

EENOKKI Ja mestaris tàmàn luettuansa rupesi tuumailemaan ankarasti?

JOOSEPPI Pasteeraili, pasteeraili edestakaisin permannolla, raappien niskatukkaansa.

EENOKKI Kiheliâitsipà miehen pààssà; mutta sità en ihmettele. — Millà tavalla luonnistui asia?

JOOSEPPI Yâllàpà vasta leikki nousi. Kovin levoton oli mestarini. Milloin pasteeraili hàn, milloin heitti hàn itsensà taasen vuoteelle, mutta samassa pyâràhti hàn ylâs jàlleen ja rupesi uudestaan pasteerailemaan raappien aina niskaansa. Kolme kertaa kàvi hàn valelemassa pààtànsà kaivolla. Hàn pelkàsi aivoansa, nàette.

EENOKKI Eikà ihme; sillà onpa sità pehmitetty. Tuumaile ja harkitse, harkitse ja tuumaile joka uusi muudi, leikkaus ja sauma, niin kysytààpnpà viimein kuinka on pààvàrkin laita. — Mutta mielinpà kuulla kuinka kàvi lopulta.

3

Figure 3: Typesetting a collection of literature marked up according to the TEI, using xmltex

```
1  <section>
2  <title>A MathML example</title>
3  <para>
4  A MathML formula can be typeset inline, as here
5  <inlineequation>
6  <math><mi>E</mi><mo>=</mo><mi>m</mi>
7  <msup><mi>c</mi><mn>2</mn></msup></math>
8  </inlineequation>, Einstein's famous equation.
9  </para>
10
11 <para>
12 A mathematical equation can also be typeset in
13 display mode using DocBook's
14 <sgmltag class="element">informalequation</sgmltag>
15 element, as is shown in the following example
16 containing a matrix:
17 </para>
18
19 <informalequation>
20 <math>
21 <mrow>
22   <mi>A</mi>
23   <mo>=</mo>
24   <mfenced open="[" close="]">
25     <mtable><!-- table or matrix -->
26       <mtr> <!-- row in a table  -->
27          <mtd><mi>x</mi></mtd><!-- table -->
28          <mtd><mi>y</mi></mtd><!-- entry -->
29       </mtr>
30       <mtr>
31          <mtd><mi>z</mi></mtd>
32          <mtd><mi>w</mi></mtd>
33       </mtr>
34     </mtable>
35   </mfenced>
36 </mrow>
37 <mtext>.</mtext>
38 </math>
39 </informalequation>
```

On lines 5–8 we use the inlineequation element, which contains a mathematical equation to be displayed inline, whereas line 19 to 39 show an informalequation element, that contains mathematical material (a matrix equation) to be typeset in display mode. The typeset result (with PassiveTEX, see below) is shown following the heading *A MathML example* in the lower left hand image of Figure 4.

The DocBook example and PassiveTEX We use the same two-step procedure (transform the XML file into XSL formatting objects and then typeset with PassiveTEX as outlined previously in the context of the TEI) here with DocBook. For the transformation of the XML DocBook markup into XSL formatting objects we use XSL stylesheets developed by Norman Walsh [15]. We customized the stylesheet somewhat to handle the mathematics and add templates for a few elements. The customized stylesheet foplus.xsl follows.

```
<?xml version='1.0'?>
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
4  version="1.0"
5  xmlns:fo="http://www.w3.org/1999/XSL/Format"
6  xmlns:m="http://www.w3.org/1998/Math/MathML">
7
8 <xsl:import
9  href="/usr/local/share/xsl/1.15/docbook.xsl"/>
10
11 <!-- ****************************************
12   customisations of Norm's XSL FO stylesheets
13   **************************************** -->
14
15 <xsl:template match="m:math">
16  <xsl:copy>
17   <xsl:apply-templates mode="math"/>
18  </xsl:copy>
19 </xsl:template>
20
21 <xsl:template  mode="math"
22     match="* | @* | comment() |
23     processing-instruction()|text()">
24   <xsl:copy>
25     <xsl:apply-templates    mode="math"
26        select="* | @* |
27          processing-instruction() | text()"/>
28  </xsl:copy>
29 </xsl:template>
30
31 <xsl:template
32      match="informalequation|inlineequation">
33   <xsl:element name="{name(.)}">
34     <xsl:apply-templates/>
35   </xsl:element>
36 </xsl:template>
37
38 <xsl:template match="programlisting" priority="4">
39   <fo:block wrap-option="no-wrap"
40             whitespace-treatment="preserve"
41             font-family="monospace"
42             font-size="9pt"
43             space-before.optimum="4pt"
44             space-after.optimum="4pt"
45             text-align="start"
46   >
47     <xsl:apply-templates/>
48   </fo:block>
49 </xsl:template>
50
51 </xsl:stylesheet>
```

Lines 3, 5 and 6 define the prefixes of the various namespaces referenced: XSLT instructions (`xsl`), XSL formatting objects (`fo`), and MathML elements (`m`), respectively. On lines 8–9 we import Norman Walsh's XSL stylesheet `docbook.xsl`, which is part of his DocBook distribution. It transforms elements in the DocBook namespace into XSL formatting objects. The `xsl:import` XSL element must be placed at the beginning of the stylesheet if template definitions further down in the stylesheet are to take precedence over those defined in the imported stylesheet `docbook.xsl`. Lines 31 to 36 specify that the elements `informalequation` and `inlineequation` themselves and their contents are to be copied through. Similarly (lines 21–29), the MathML element `m:math` and the whole tree structure it supports have to be copied unchanged to the result tree. This means that the MathML elements must be dealt with by the application that interprets the XSL formatting objects (in our case PassiveTEX and `xmltex`). Finally, lines 38 to 49 show how we define the characteristics of the `programlisting` element that is used to represent verbatim material and is to be typeset *as-is* (similar to LATEX's `verbatim` environnement).

The two-step procedure to typeset the XML source file `dbxmml.xml` uses the following commands:

```
xt dbxmml.xml foplus.xsl fotex.xml
latex fotex.tex
```

The first line transforms the DocBook markup into XSL formatting objects, that are then handled (as well as passed through MathML markup) by PassiveTEX (and `xmltex`) referenced on the second line. Figure 4 shows the result of typesetting the complete DocBook example document with PassiveTEX.

XML and PassiveTEX at the heart of the Internet

Different ways of using an XML document in the context of an electronic document repository are shown in Figure 5. At the top right we represent the XML document with its defining vocabulary (DTD or XML Schema [18], [19] and [20]). This document, which is encoded in Unicode, can be viewed, searched, indexed, edited, validated without problems by any of a series of XML-aware applications anywhere on the Internet. The XML document can be typeset using TEX or its Unicode-aware variant Omega [8]. Three methods are shown: (A) uses direct interpretation by `xmltex`, an example of which was shown in Figure 2, (B) uses XSL formatting objects and PassiveTEX, as described in the present article, (C) transforms the XML source into a LATEX file. This procedure was used for typesetting the present article so that the resulting LATEX document could be included in proceedings of TUG 2000 Conference.

Alternatively the XML source can be transformed into HTML for viewing with present-day browsers (X2H via XSLT, see [7] for a detailed discussion). In the (near) future, once browsers can handle XML directly, we can probably skip the HTML intermediate format and let CSS [16] (possibly via XSLT) style the XML file directly for display on the Web. Figure 5 also contains arrows going from left (TEX) to right (XML/HTML,

A Docbook document featuring a few formulae

Michel Goossens
Wednesday, 18 July 2000

Abstract

This XML document is marked up according the the DocBook schema It shows a few elements of the DocBook vocabulary, as well as a couple of examples of mathematical expressions where we used MathML markup.

The Docbook model

DocBook ??? proposes an XML model ??? for marking up technical documents. It is particularly well-suited for software reference guides and computer equipment manuals.

Docbook contains hundreds of elements to markup up clearly and explicitly the different components of an electronic document (book, article, reference guide, etc.), not only displaying its hierarchical structure but also indicating the semantical meaning of the various elements. The structure of the DTD is optimized to allow for customization, thus making it relatively straightforward to add or eliminate certain elements or attributes, to change the content model for certain structural groups, or to restrict the value that given attributes can take.

Norman Walsh developed a set of XSL stylesheets to transform XML documents marked up using the DocBook DTD into HTML or XSL formatting objects. The latter can be interpreted by PassiveTeX and xmltex to obtain PDF or PostScript output.

MathML and mathematics

MathML ??? is a W3C recommendation whose aim is to encode mathematical material for teaching and scientific communication at all levels.

MathML consists of two parts: presentation (notation) et contents (meaning). MathML permits the conversion of mathematical information into various representations and output formats, including graphical displays, speech synthesizers, computer algebra programs, other mathematics typesetting languages, such as TeX, plain text, printers (PostScript), braille, etc.

MathML has support for efficiently browsing long and complex mathematical expressoins and offers an extension mechanism. MathML code is human readable, easy to generate and process by software applications, and well suited for editing (even though MathML syntax might appear unnecessarily verbose and complex to the human reader).

The W3C MathMl Working Group is actively preparing the second version of the MathML Specification, which is planned to be released in the second half of 2000. Two public initiatives that allow the display of MathML code direcly and that are under active development are W3C's Amaya and Mozilla. Commercial programs, such as IBM's techexplorer (a plugin for Netscape or Microsoft's Internet Explorer) or Design Science Webeq (using the Java applet technology) can display MathML formulae in present day browsers. Several computer algebra programs, e.g., Mathematica, or editors using e.g., mathtype, offer a user-friendly interface to enter, produce or read mathematics material marked up in MathML.

`maligngroup/` and `malignmark/` alignment markers

- Enlivening expressions

`maction`	bind actions to a sub-expression

A MathML example

A MathML formula can be typeset inline, as here $E = mc^2$, Einstein's famous equation.

A mathematical equation can also be typeset in display mode using DocBook's `informalequation` element, as is shown in the following example containing a matrix:

$$A = \left[\begin{array}{cc} x & y \\ z & w \end{array} \right] .$$

Note the two attributes open and close on the `mfenced` element to specify the style of the braces to be used. The MathML Specification ??? contains a detailed list of all possible attributes associated to each presentation element.

Content markup

The meaning of mathematical symbols (e.g., sums, products, integrals) exists independently of their rendering. Moreover the presentation markup of an expression is not necessarily sufficient to evaluate its value and use it in calculations. Therefore, MathML defines *content* markup to explicitly encode the underlying mathematical structure of an expressoin.

It is impossible to cover all of mathematics, so MathML proposes only a relatively small number of commonplace mathematical constructs, chosen carefully to be sufficient in a large number of applications. In addition, it provides a *extension mechanism* for associating semantics with new notational constructs, thus allowing these to be encoded even when they are not in MathML content element base collection.

MathML's basic set of content elements was chosen to allow for simple coding of most of the formulas used in secondary education, through the first year of university. The subject areas targeted are arithmetic, algebra, logic and relations, calculus and vector calculus, set theory, sequences and series, elementary classical functions, and statistics linear algebra. Using this basic sets more complex constructs can be built.

The list of the content elements of MathML follows.

token elements	`cn`, `ci`, `csymbol` (MathML 2.0).
basic content elements	`apply`, `reln` (deprecated), `fn` (deprecated for externally defined functions), `interval`, `inverse`, `sep`, `condition`, `declare`, `lambda`, `compose`, `ident`.
arithmetic, algebra and logic	`quotient`, `exp`, `factorial`, `divide`, `max` and `min`, `minus`, `plus`, `power`, `rem`, `times`, `root`, `gcd`, `and`, `or`, `xor`, `not`, `implies`, `forall`, `exists`, `abs`, `conjugate`, `arg` (MathML 2.0), `real` (MathML 2.0), `imaginary` (MathML 2.0), `lcm` (MathML 2.0).
relations	`eq`, `neq`, `gt`, `lt`, `geq`, `leq`, `equivalent` (MathML 2.0), `approx` (MathML 2.0).
calculus and vector calculus	`int`, `diff`, `partialdiff`, `lowlimit`, `uplimit`, `bvar`, `degree`, `divergence` (MathML 2.0), `grad` (MathML 2.0), `curl` (MathML 2.0), `laplacian` (MathML 2.0).
theory of sets	`set`, `list`, `union`, `intersect`, `in`, `notin`, `subset`, `prsubset`, `notsubset`, `notprsubset`, `setdiff`, `card` (MathML 2.0).
sequences and series	`sum`, `product`, `limit`, `tendsto`.
elementary classical function	`exp`, `ln`, `log`, `sin`, `cos`, `tan`, `sec`, `csc`, `cot`, `sinh`, `cosh`, `tanh`, `sech`, `csch`, `coth`, `arcsin`, `arccos`, `arctan`, `arccosh`, `arccot`, `arccoth`, `arccsc`, `arccsch`, `arcsec`, `arcsech`, `arcsinh`, `arctanh`.
statistics	`mean`, `sdev`, `variance`, `median`, `mode`, `moment`.
linear algebra	`vector`, `matrix`, `matrixrow`, `determinant`, `transpose`, `selector`, `vectorproduct` (MathML 2.0), `scalarproduct` (MathML 2.0), `outerproduct` (MathML 2.0).
semantic mapping element	`annotation`, `semantics`, `annotation-xml`.
constant and symbol element (all MathML 2.0)	`integers`, `reals`, `rationals`, `naturalnumbers`, `complexes`, `primes`, `exponentiale`, `imaginaryi`, `notanumber`, `true`, `false`, `emptyset`, `pi`, `eulergamma`, `infinity`.

Presentation markup

The *presentation* part of MathML discribes the spacial layout of the different elements of a mathematical expression. MathML presenation markup has about thirty elements, that form the basis of a mathematical syntax using classical visual layout model. Some fifty attributes provide precise control on the exact positioning of the various components of the math expression.

List of presentation elements

- Token elements

`mi`	identifier
`mn`	number
`mo`	operator, fence, separator
`mtext`	text
`mspace/`	space
`ms`	string literal
`mchar`	non-Ascii character reference
`mglyph`	add new glyph to MathML

- General layout schemata

`mrow`	group any number of sub-expressions horizontally
`mfrac`	form a fraction from two sub-expressions
`msqrt`	form a square root sign (radical without an index)
`mroot`	form a radical with specified index
`mstyle`	style change
`merror`	enclose a syntax error message from a preprocessor
`mpadded`	adjust space around content
`mphantom`	make content invisible but preserve its size
`mfenced`	surround content with a pair of fences
`menclose`	enclose content with a stretching symbol such as a long division sign

- Script and limit schemata

`msub`	attach a subscript to a base
`msup`	attach a superscript to a base
`msubsup`	attach a subscript-superscript pair to a base
`munder`	attach an underscript to a base
`mover`	attach an overscript to a base
`munderover`	attach an underscript-overscript pair to a base
`mmultiscripts`	attach prescripts and tensor indices to a base

- Tables and matrices

`mtable`	row in a table or matrix with a label or equation number
`mtr`	row in a table or matrix
`mtd`	one entry in a table or matrix

The matrix example given in the preceding section in its presentation markup form if recoded here using content markup.

```
<reln>
 <eq/>
 <ci>A</ci>
 <matrix>
 <matrixrow>
 <ci>x</ci><ci>y</ci>
 </matrixrow>
 <matrixrow>
 <ci>z</ci><ci>w</ci>
 </matrixrow>
 </matrix>
</reln>
```

Bibliographic references

[XML98] World Wide Web Consortium, Tim Bray, Jean Paoli, and Michael Sperberg-McQueen. *Extensible Markup Language (XML) 1.0 [http://www.w3.org/TR/REC-xml/]*. See also the annotated version of the XML specification [http://www.xml.com/axml/axml.html]..

[WALSH99] Norman Walsh and Leonard Muelner. *Docbook. The Definitive Guide.*. O'Reilly & Associates, Inc.. Copyright © 1999. 1-56592-580-7. You can also consult the online version of the DocBook reference guide [http://www.oasis-open.org/docbook/.html] and download the Docbook DTD and XSL stylesheets [http://nwalsh.com/docbook/index.html]..

[MATHML99] World Wide Web Consortium, Patrick Ion, and Robert Miner. *Mathematical Markup Language (MathML[tm]) 1.01 Specification [http://www.w3.org/TR/REC-MathML/]*. .

Figure 4: Typeset result of an article marked up according to the DocBook schema

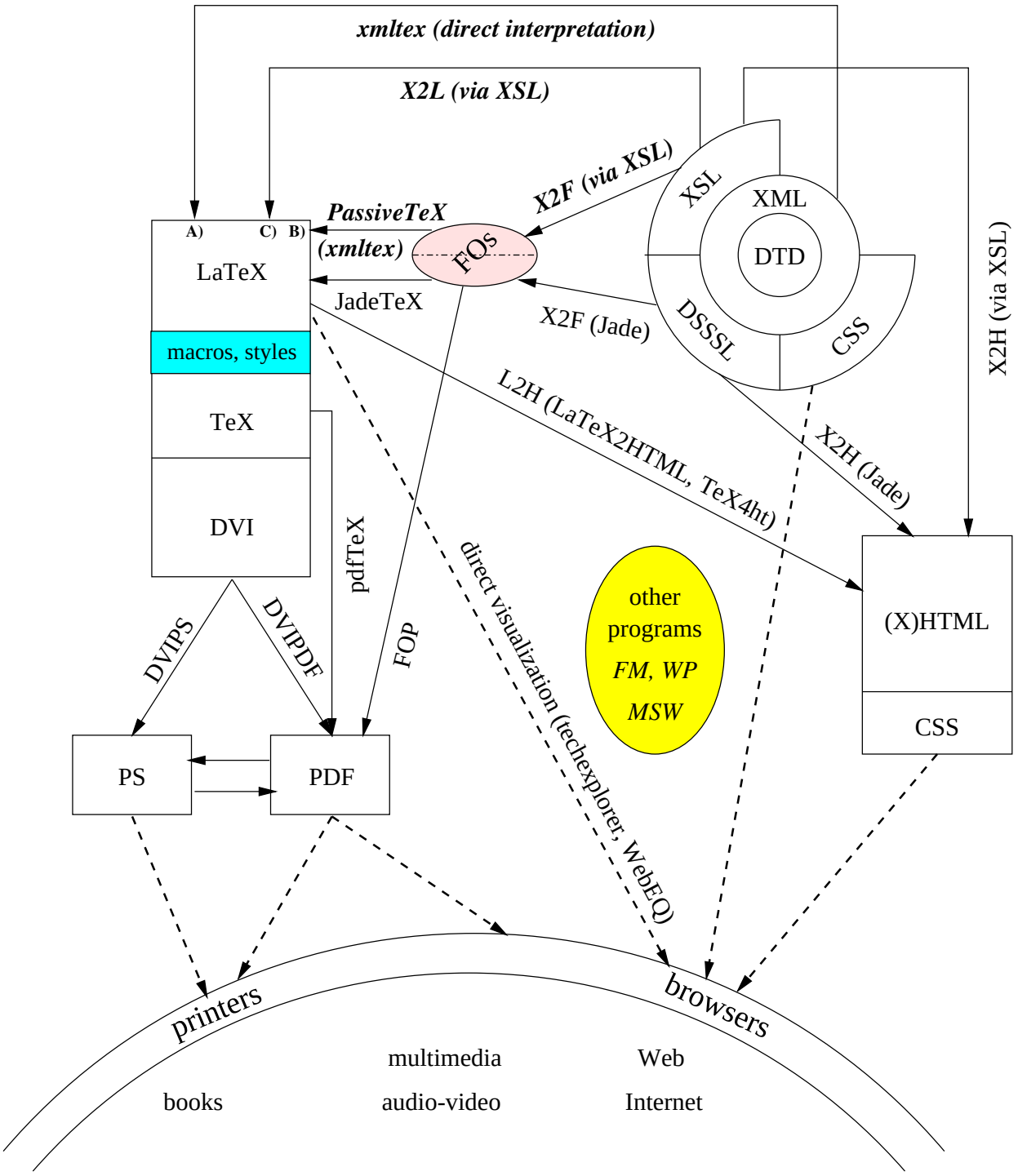


Figure 5: XML as the central part of a document strategy for the Web

browsers). They indicate programs to transform existing LaTeX source documents into XML (using one or more standard DTDs) to store the information for archiving purposes. The vertical ellipse in the centre represents other editing tools, such as Adobe's *FrameMaker* [1] and Corel's *WordPerfect* [6], that allow or are expected to allow import/export of XML documents. Thus, XML genuinely becomes the central element in a global strategy for managing electronic documents by allowing information to be stored, saved, shared, and used by different applications on all computer platforms. PassiveTEX can be truly considered as a much-needed complement to XML for bringing typographic excellence to the Web, just as Knuth with TEX introduced the same excellence to the emerging electronic printing industry a generation ago.

References

[1] Adobe. *FrameMaker 6.0.* http://www.adobe.com/products/framemaker.

[2] Apache XML Project. *FOP, XSL Formatting Object Processor in Java.* http://xml.apache.org/fop/

[3] Lou Burnard and C.M. Sperberg-McQueen. TEI Guidelines for Electronic Text Encoding and Interchange. http://etext.lib.virginia.edu/tei.html

[4] David Carlisle. *xmltex A non validating (and not 100% conforming) namespace aware XML parser implemented in TEX.* Can be downloaded from CTAN in the directory macros/xmltex/.

[5] James Clark. *xt, an implementation in Java of XSL Transformations.* http://www.jclark.com/xml/xt.html

[6] Corel. *WordPerfect Office 2000.* http://www.corel.com/Office2000 (Microsoft Windows) and http://linux.corel.com/products/wpo2000_linux (Linux).

[7] Michel Goossens and Sebastian Rahtz. *The LaTeX Web Companion.* Addison-Wesley, Reading, 1999.

[8] Yannis Haralambous and John Plaice. The latest developments in Omega. *TUGBoat*, 17 (2), pages 181-183, June 1996. (See also http://www.gutenberg.eu.org/omega/).

[9] John Hobby. *A user's manual for MetaPost.* Computer Science Technical Report 162, AT&T Bell Laboratories, 1992.

[10] International Organization for Standardization. *Information Technology—Processing Languages—Document Style Semantics and Specification Language (DSSSL). First edition, 1996* International Standard ISO/IEC 10179:1996.

[11] Sebastian Rahtz. *Passive TEX* http://users.ox.ac.uk/~rahtz/passivetex/

[12] Sebastian Rahtz. *XSL stylesheets for TEI XML.* http://users.ox.ac.uk/~rahtz/tei/

[13] The Unicode Consortium. *The Unicode Standard, Version 3.0.* Addison-Wesley, Reading, 2000.

[14] Norman Walsh and Leonard Muelner. *DocBook. The Definitive Guide.* O'Reilly & Associates, Inc., Sebastopol, USA, 1999. See also http://nwalsh.com/docbook/index.html.

[15] Norman Walsh. *XSL DocBook Stylesheets.* http://nwalsh.com/docbook/xsl/index.html

[16] World Wide Web Consortium. Håkon Wium Lie, Bert Bos, Chris Lilley and Ian Jacobs (editors). *Cascading Style Sheets, level 2.* http://www.w3.org/TR/REC-CSS2.

[17] World Wide Web Consortium. Patrick Ion and Robert Miner (editors). *Mathematical Markup Language (MathML[tm]) 1.01 Specification.* http://www.w3.org/TR/REC-MathML/.

[18] World Wide Web Consortium, David C. Fallside (editor). *XML Schema Part 0: Primer (W3C Working Draft).* http://www.w3.org/TR/xmlschema-0.

[19] World Wide Web Consortium, Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn (editors). *XML Schema Part 1: Structures (W3C Working Draft).* http://www.w3.org/TR/xmlschema-1.

[20] World Wide Web Consortium, Paul V. Biron, Ashok Malhotra (editors). *XML Schema Part 2: Datatypes (W3C Working Draft).* http://www.w3.org/TR/xmlschema-2.

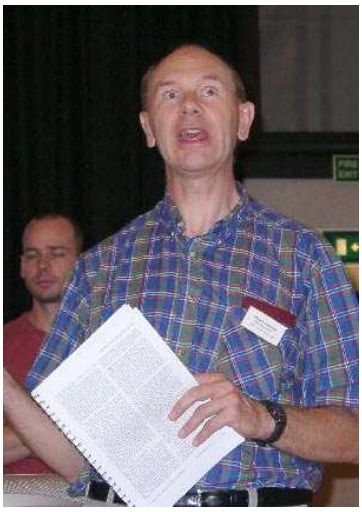
[21] World Wide Web Consortium, Jon Ferraiolo (editor). *Scalable Vector Graphics (SVG) 1.0 Specification (W3C Working Draft).* http://www.w3.org/TR/SVG.

[22] World Wide Web Consortium. Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen

(editors). *Extensible Markup Language (XML) 1.0.* http://www.w3.org/TR/REC-xml. An annotated version of the specification is at http://www.xml.com/axml/axml.html.

[23] World Wide Web Consortium, James Clark (editor). *XSL Transformations (XSLT), Version 1.0 (W3C Recommendation 16 November 1999).* http://www.w3.org/TR/xslt.

[24] World Wide Web Consortium, Stephen Deach (editor). *Extensible Stylesheet Language (XSL), Version 1.0 (W3C Working Draft).* http://www.w3.org/TR/WD-xsl.



Michel Goossens

Sebastian Rahtz

Fast scanners and self-parsing in TEX

Pedro Palao Gostanza
Universidad Complutense de Madrid
Edificio de CC. Matemáticas,
Ciudad Universitaria s/n,
Madrid 28040,
Spain
ecceso@sip.ucm.es

Abstract

In this paper we will explain how to build fast scanners for regular languages in TEX. The resulting scanners can be composed with different parsers if they are written as *self parsers*. This parsing technique allows us to choose the syntactic rigour: from loose parsers with almost no syntax checks, good for pretty-printers, to the strict parser needed to write a compiler.

Introduction

It is customary to embed little languages in TEX; LATEX picture environment, XY-pic [9] and TESLA [2] are illustrative examples. TEX can process these languages in a reasonable fast manner because their syntaxes are chosen with TEX's abilities in mind; for instance, arguments are surrounded with braces or delimited with fixed characters. Also, programs written with these little languages are usually short.

Sometimes, TEX has to process programs written in *bigger* languages, or in languages designed without paying attention to TEX. Pretty-printing structured programming languages is the ubiquitous example [6, 11], but we also meet the same problems when trying to typeset HTML directly or when doing some *Cronopio* [3] activities like executing programs written in high level languages [5] within TEX. Programs to cope with these problems are written around a slow character by character processing engine. The `\FIND` macro ([4, 10]), possible in a modified or subtle variant, is the kernel of this processing engine. This is the method adopted in Doumont's pretty-printer [6] and Woliński's scanners [11]. Slow processing is particularly striking here because programs in these languages can be really long.

We became interested in this problem several years ago, while doing just another Pascal pretty-printer [7]. Since we found no trick to build a fast scanner, we simply put the burden on the user who was forced to write a backslash before every identifier. In this way, every identifier was a control word and we could take advantage of TEX's scanner. Obviously, this was a poor design decision. Fortunately, one year ago we found a trick to write fast scanners in TEX; to our knowledge, it seems to be an unused TEXnique.

The idea to make a fast scanner in TEX is *not to check* every character. But, since a scanner must *see* every character, the implementation must use *burst like processing*: the scanner checks some characters, but TEX internal machinery eats the rest. In this way, the final complexity will be a small constant (due to TEX internals) multiplying a linear factor, plus a big constant (due to scanner checks) multiplying a sub-linear factor. We will call every scanner that works without checking every character in its input a *fast scanner*.

Our trick to make a fast scanner is based on active characters and `\edef`. We will illustrate it with Pascal in the next section. The scanner engine is spread along all the active characters; each character knows how to keep the scanner alive. This organization, where every token knows what to do, and there is no centralized set of processing macros, is what we call *self-parsing*. Self-parsing can be used to build scanners and parsers. Its two main advantages are: firstly, it allows to split scanners and parsers like in a traditional compiler, and secondly, the syntactic rigour can be chosen. The first advantage allows us to write a definitive Pascal scanner that can be used to feed a pretty-printer parser, a compiler parser, etc. The second allows us to write a strict parser for a compiler and a loose parser for a pretty-printer with this same technique. Of course, each parser will have to be developed from scratch.

To illustrate all these ideas we will use, as a running example, a small Pascal subset that we

Program ::= `program` Id `;` Block `.`
Block ::= Decls `begin` Stats `end`
Decls ::= ⟨*empty*⟩ | Vars Decls
Vars ::= `var` SeqVar SeqsVar
SeqsVar ::= ⟨*empty*⟩ | SeqVar SeqsVar
SeqVar ::= Ids `:` Id `;`
Ids ::= Id IdsExt
IdsExt ::= ⟨*empty*⟩ | `,` Ids
Stats ::= Stat StatsExt
StatsExt ::= ⟨*empty*⟩ | `;` Stat StatsExt
Stat ::= Id `:=` Expr
| Id OptArgs | `if` Expr `then` Stat Else
| `while` Expr `do` Stat | `begin` Stats `end`
Else ::= ⟨*empty*⟩ | `else` Stat
OptArgs ::= ⟨*empty*⟩ | `(` Args `)`
Args ::= Expr ArgsExt
ArgsExt ::= ⟨*empty*⟩ | `,` Args
Expr ::= Rel
Rel ::= Term RelExt
RelExt ::= ⟨*empty*⟩ | `=` Term | `<>` Term
| `<` Term | `<=` Term | `>` Term | `>=` Term
Term ::= Factor TermExt
TermExt ::= ⟨*empty*⟩
| `+` Factor TermExt | `-` Factor TermExt
Factor ::= Atom FactorExt
FactorExt ::= ⟨*empty*⟩ | `*` Atom FactorExt
| `div` Atom FactorExt | `mod` Atom FactorExt
Atom ::= Int | Id | `(` Expr `)` | `-` Atom

Id ::= Letter LettersOrDigits
LetterOrDigits ::= ⟨*empty*⟩ | Letter LetterOrDigits
| Digit LettersOrDigits
Int ::= Digit Digits
Digits ::= ⟨*empty*⟩ | Digit Digits
Digit ::= `0` | ··· | `9`
Letters ::= `a` | ··· | `z` | `A` | ··· | `Z`

Figure 1: Mini-Pascal syntax

will call 'mini-Pascal.' Its syntax can be found in Figure 1.

The rest of this paper is organized as follows. In the next section, we explain how to write a fast scanner for mini-Pascal. Although its main idea can be reused, each language has its own tricks that speed the scanner even more; part of this section is devoted to explore useful tricks for Pascal and other structured programming languages. Next we will devote two sections to building a pretty-printer and a compiler for mini-Pascal. Both parsers work on top of the same scanner. Since pretty-printing is a widely studied area, the goal of our pretty-printer is not how to pretty-print a programming language but how to build a loose self-parser. Obviously, the goal of the compiler is how to built a strict self-parser. The section 'Other uses' reviews other projects where we have used fast scanners and self-parsers. Finally, we conclude and suggest some future work.

A Pascal scanner

By far, identifiers and keywords account for most of the characters in a Pascal program. Letters are used exclusively for this purpose.[1] Digits can also appear in identifiers, but do so rather seldom; they almost exclusively appear in numbers. Every other character, apart from white space, is seldom used. White space has a strange occurrence pattern: every line starts with a long white sequence (just after an end of line), and then single spaces split other tokens.

In order to get really good sub-linear behavior, a scanner should operate checking no character in any identifier (including keywords), and no space in every start-of-line white sequence.

The scanner will change each Pascal token into a TEX control word: the identifier `Foo` to `\Id{Foo}`, the number `123` to `\Int{123}`, the keyword `begin` to `\Begin`, `:=` to `\Assign`, etc.

Every character but roman letters will be active; that is, `,`, `.`, `:`, `;`, `(`, `)`, `+`, `-`, `*`, `=`, `<`, `>`, `0`, ..., `9`, and blanks are active characters. Forget for a moment digits and symbols composed with more than one active character, like `<>`. So, every identifier is composed only with letters (non-active characters) and is surrounded with active characters. To catch these identifiers without an explicit character-by-character analysis, it is enough to start a capture at the end of each active character and to finish this capture at the beginning of each active character. The macro `\catchId` starts the capture:

```
\def\catchId{\edef\mayId{\iffalse}\fi}
```

This macro store every following characters in `\mayId`, while expanding active characters. To finish this capture an active: character uses

```
\def\endId{\iffalse{\else}\fi}
```

This capture does not always success; for example, there can be two active characters one after the other. Since every character that cannot take part in an identifier is active, the capture will be successful if and only if `\mayId` is not `\empty`.

[1] Not exactly: `e` and `E` are used in floating point literal numbers.

```
\def\empty{}
\def\flush{\ifx\mayId\empty\else
  \expandafter\Id\expandafter{\mayId}\fi}
```

These three macros are all a white space must do:

```
\def\blank{\endId\flush\catchId}
```

This is also needed at the beginning and the end of the scanner

```
\def\beginScanner{\activeChars\defActive
  \catchId}
\def\endScanner{\endId\flush}
```

Other active characters will produce, in addition, its own Pascal token:

```
\def+{\endId\flush\Plus\catchId}
\def.{\endId\flush\Dot\catchId}
...
```

Of course, some identifiers are keywords, which introduces two problems. First, it is necessary to check every identifier in order to test whether it is a keyword or a regular identifier. Several TEXniques to implement sets may be used, for example:

- For each keyword k there is a control word `\kw@`k trivially defined (but not `\relax`). Checking whether an identifier is a keyword is simple and fast:

  ```
  \def\ifIsNotKW#1{\expandafter\ifx
    \csname kw@#1\endcsname\relax}
  ```

  But each new identifier introduces a new entry in TEX control sequences table. Since TEX never deletes entries from this table, we can exhaust TEX limited hash memory.
- All keywords can be stored in a macro:

  ```
  \def\kws{|begin|end|if|...|while|}
  ```

  Checking an identifier is tricky and long

  ```
  \def\ifIsNotKW#1{\def\aux
    ##1|#1|##2\relax{\ifx\relax##2\relax}%
    \expandafter\aux\kws#1|\relax}
  ```

  but it works in a constant amount of memory.

Since we want to build fast scanners, the first technique is preferable.

The second problem arising with keywords concerns capitalization. Pascal is case insensitive, so we should be capable of recognizing an identifier or keyword without regard to its capitalization. This can be easily solved invoking a `\lowercase` over those characters stored in `\mayId`. Since some parsers care about capitalization, but others do not, it is better to change the identifier `Foo` into `\Id{foo}{Foo}` instead of only to `\Id{Foo}`.

Part of the speed has already been achieved. The other acceleration source is to deal with spaces at the beginning of lines. The trick is to recover the original category codes of spaces at end of lines. Then we look for the next TEX token so that TEX eats all the intermediate spaces.

```
\def\eoln{\endId\flush\catcode`\ =10 \eolnB}
\def\eolnB#1{\catcode`\ =\active\catchId#1}
```

Category code 9 (ignored character) also works.

To consider numbers and composed symbols, an state needs to be added to the scanner; we will call it 'scanner state' because, latter in the paper, some other states will come into play. The scanner state due to numbers is a macro `\mayInt` where its digits will be stored. Composed symbols need two macros: `\maySym` and `\symCode`. `\symCode` is a number that uniquely determines what characters are in the current symbol; it is 0 if there is no character, or a non-zero integer for each of the three characters that can start composed symbols:

```
\chardef\noCode=0
\chardef\colonCode=1
\chardef\lessCode=2
\chardef\greaterCode=3
```

`\maySym` stores which token will be generated if there is no character extending the current symbol. For instance, a colon does not generate a `\Colon` token directly; instead, it is stored, so that, if there is an equal immediately after it, an `\Assign` will be generated.

```
\def:{\endId\flush
  \gdef\maySym{\Colon}\glet\symCode\colonCode
  \catchId}
```

But what if there is an identifier followed by an space; the `\Id` token will be generated before the `\Colon`; it is even possible that the `\Colon` get lost. The solution is simple: `\flush` must not only take care of captured identifiers but also of delayed symbols and stored numbers.

```
\def\flush{\flushSym\flushInt\flushId}
\def\genSym{\maySym\glet\symCode\noCode}
\def\flushSym{\ifnum\symCode=\noCode
  \else\genSym\fi}
\def\genInt{\expandafter\Int\expandafter
  {\mayInt}\glet\mayInt\empty}
\def\flushInt{\ifx\mayInt\empty\else\genInt\fi}
\def\genId{\expandafter\Id\expandafter{\mayId}}
\def\flushId{\ifx\mayId\empty\else\genId\fi}
```

Characters that can be in the second place of a composed symbol cannot simply `\flush`; they should flush integers and identifiers, but can only flush delayed symbols if there is something intertwined:

```
\def\flushInter{%
  \ifx\mayInt\empty\else\flushSym\genInt\fi
  \ifx\mayId\empty\else\flushSym\genId\fi}
```

The definition of '`>`' is illustrative because it can be the first and the second character in a composed symbol:

```
\def>{\endId\flushInter
  \ifnum\symCode=\lessCode
```

```
    \glet\symCode\noCode \NotEqual
  \else\flushSym\gdef\maySym{\Greater}%
    \glet\symCode\greaterCode
  \fi\catchId}
```

Digits do important work to keep the scanner alive, but they do not need to flush because a character will follow that will cause flushing. The main digit task is to know whether it is part of a number or an identifier.

```
\def\digit#1{\endId
  \ifx\mayInt\empty
    \ifx\mayId\empty \gdef\mayInt{#1}\catchId
    \else \catchId\mayId#1\fi
  \else\ifx\mayId\empty
    \xdef\mayInt{\mayInt#1}\catchId
  \else\errmessage{An identifier cannot
    start with digits}\catchId
  \fi\fi}
```

This completes our fast scanner. But a little problem remains. Sometimes, the program we want to parse is not embedded in the document sources, but stored in a separate file. Invoking `\input` inside an scanner context (`\beginScanner\endScanner`) has no use because backslash will loose its usual meaning inside this context. The usual roundabout

```
\expandafter\beginScanner\input p.pas \endScanner
```

does not work either (read `\@@input` instead of `\input` if thinking in LaTeX) because the last active character in `p.pas` launches a `\catchId` that will be closed in `\endScanner` after the end-of-file, i.e., the last active character starts a definition that will be closed pass the end of the file. Since TeX does not allow a definition to span across files, we will get the error "File ended while scanning definition of `\mayId`." Fortunately, TeX appends an end-of-line character to the last line of every file, if absent; so the last character in every file processed with TeX is an end-of-line. We are going to put on EOLN character the burden of crossing the end-of-file boundary before to launch `\catchId`. Of course, if not at the end of a file, an EOLN should behave as before. The cheapest manner to cross the end-of-line boundary is with `\futurelet`:

```
\def\eoln{\endId\flush\catcode`\ =9
  \futurelet\aux\eolnB}
\def\eolnB{\catcode`\ =\active\catchId}
```

Incidentally, `\futurelet` makes leading spaces (ignored characters) in the next line disappear.

A Pascal pretty-printing

Now, we have our scanner ready. Let us use it to build a pretty-printer. The pretty-printer is responsible for choosing a correct definition for the tokens that the scanner generates. These definitions cannot look forward following tokens because the scanner may not have produced them yet and because there can be TeX control words (that remain to be evaluated) before the next token. So the pretty-printer must conform to the self-parsing technique. Of course, each token can change the pretty-printer state, in order to produce a visible effect, to prepare the environment for the next tokens, and to communicate to future tokens its previous occurrence.

Self-parsing is such a natural technique to use in a pretty-printer that it has been discovered and used in several pretty-printers before (at least in [6] and in [7]). But it has never been used in a pure manner, neither recognized as a useful general parsing technique. So, our emphasis will be to explain how self-parsing can be used to build a loose parser. Pretty-printer output will be very simple, just the raw style in [7]: every statement in a line; keywords are in bold face; identifiers are in italics; every expression, assignment and procedure call is typeset in TeX math mode. The formatted program to compute x^n, for $x = 3$ and $n = 9$, follows:

program *power*;
  var *x*, *n*: *integer*;
      x1, *n1*, *pow*: *integer*;
begin
  $x \leftarrow 3$;
  $n \leftarrow 9$;
  $x1 \leftarrow x$;
  $n1 \leftarrow n$;
  $pow \leftarrow 1$;
  while $n1 \geq 1$ **do begin**
    if $n1$ **mod** $2 = 1$ **then** $pow \leftarrow pow \times x1$;
    $x1 \leftarrow x1 \times x1$;
    $n1 \leftarrow n1$ **div** 2
  end;
  write(*pow*)
end.

To keep the pretty-printer alive, every token must do some work. Some, like parenthesis, do a really simple work, without bothering about where it is used.

```
\def\OpenPar{(}
```

Others, like assignments, do a simple work too, but require that other tokens have already opened a math mode.

```
\def\Assign{\leftarrow}
```

An assignment in an incorrect place will cause a "Missing $ inserted" error; this is a syntactic check with a bad error message.

Identifiers behave differently if placed at the beginning of an statement or inside an expression. In the first case, they must open a math mode; in the second case, they only have to write themselves. The best agreement is to use a `\ifinsideexpr` condition

```
\let\ifinsideexpr\ifmmode
```

```
\def\openExpr{\ifinsideexpr\else$\fi}
\def\closeExpr{\ifinsideexpr$\fi}
\def\Id#1{\openExpr\hbox{\it#1}}
```

If a token can appear after an expression it should ensure that the expression is closed; for example:

```
\def\Semicolon{\closeExpr;\par}
```

But not every semicolon behaves in this way; the semicolon after the program name must indent following constructions a bit. There are three techniques to solve this problem:

Conditions technique in which we have a global condition `\ifafterprogram`, which the definition of `\Program` sets true. Other definitions set it false; since there is only one semicolon after a program name, the definition of semicolon is the best place to set it false.

Redefinition technique in which we have one definition for each possible behaviour. Other tokens redefine `\Semicolon` according to the expected behaviour in an immediate future.

Steps technique in which we have a number, a step holder, that records the syntactic construction where the present token occurs. Each token can adjust its behaviour according to the value in the step holder, and change it as necessary.

These three techniques are equivalent, but depending on the problem one is easier than the others. As the syntactic checks become stricter, one should move from the first to the third. These three techniques can be used simultaneously; for instance, in the above fast scanner we have mixed conditions and steps in order to build symbols composed with more than one character.

Usually a stack is needed in order to store values (before changing a condition, making a redefinition or assigning to the step holder) that will be restored when a nested construction ends. For a pretty-printer, TEX grouping is enough, but for stricter parsers it is better to maintain an explicit stack.

The following macros illustrate a pretty-printer built with redefinition technique.

```
\def\Program{{\bf program}\
  \let\Begin\BeginBlock
  \let\Semicolon\SemicolonProgram}
\def\SemicolonProgram{\closeExpr;\par\indent
  \let\Semicolon\SemicolonBlock}
\def\SemicolonBlock{\closeExpr;\par}
\def\BeginBlock{\par\outdent
  {\bf begin}\par\indent
  \let\Begin\BeginStat}
\def\BeginStat{\ {bf begin}\par\indent}
```

A Pascal compiler

Implementing a pretty-printer with self-parsing is an easy task, easier than doing it with a classical and strict parser. So, we wonder how the effort needed to write a self-parser evolves when increasing syntactic checks. We thought that the best test was to write a mini-Pascal compiler.

We envisaged the following organization. To compile a program, it must be surrounded with the pair `\beginPC`/`\endPC`. A program called *power*, for instance, will be translated to a TEX macro called `\power`, that comprises an instruction sequence for a virtual stack machine. Whenever this macro is called, its instructions get executed, and everything written (with *write*) appears inserted in the text.

Since a compiler needs to ensure a complete syntactic conformance, we will use the step technique to produce its parser. But which are the correct steps? This question has already been answered: classical parsing techniques, like LL and LR, rewrite a context free grammar as an automaton. This automaton states are the steps we were looking for.

Here we will work out how to build an LL self-parser because it is simpler than LR parsing and Pascal has an (almost) LL grammar. Nevertheless, the main idea and many details can be reused in an LR self-parser.

The construction of an LL self-parser for a given grammar has been automated with a simple program (written in Haskell [8]) called parTEX. Here we are explaining how to do by hand what this program already does alone. This program expects an LL grammar annotated with semantic actions and semantic checks. Figure 2 shows the production for mini-Pascal statements. Semantic actions are surrounded with braces and semantic checks are also preceded with a question mark. Both semantic actions and checks use several auxiliary macros that read and modify the compiler state; an explanation of their implementation and behaviour is beyond the scope of this paper, but their names are chosen to evoke its meaning (sequences without spaces or end-of-lines are just one macro call with its arguments). Semantic actions (checks) immediately following a terminal that carries information, like an `Id`, get this information through parameters. So, in the semantic actions (checks) following `Id`, `#1` is the identifier down-cased string and `#2` (not used) is the identifier string.

Then, following [1], we add state numbers between every symbol that appears in each production right hand side. The state numbers for the production in Figure 2 (without semantic actions) are:

```
Stat ::=
  Id ?{\isVarQ{#1}}
      {\memDir{#1}\aMemDir
       \emitCode{\lit{\aMemDir}}}
  ":=" Expr {\emitCode{\put}}
| Id {\def\procToCall{#1} \noa=0 }
  OptArgs {\iftrue \isProcQ\procToCall\noa
       \procDir\procToCall\noa\aProcDir
       \emitCode{\call{\aProcDir}}%
     \else \errmessage{No procedure
       "\procToCall" with \number\noa\space
       arguments}\fi}
| "if" Expr {\newLabel\labelse \newLabel\labend
             \emitCode{\jzero{\number\labelse}}}
  "then" {\bgroup}
  Stat {\egroup \emitCode{\jump{\number\labend}}%
        \emitCode{\label{\number\labelse}}}
  Else {\emitCode{\label{\number\labend}}}
| "while" {\newLabel\loopL \newLabel\endloopL
          \emitCode{\label{\number\loopL}}}
  Expr {\addCode{\jzero{\number\endloopL}}}
  "do" {\bgroup}
  Stat {\egroup \addCode{\jump{\number\loopL}}
        \emitCode{\label{\number\endloopL}}}
| "begin" Stats "end"
.
```

Figure 2: Statement production

```
Stat ::=
  Id 42 ":=" 43 Expr 44
| Id 45 ArgsOpt 46
| "if" 48 Expr 49 "then" 50 Stat 51 Else 52
| "while" 54 Expr 55 "do" 56 Stat 57
| "begin" 59 Stats 60 "end" 61
```

There is an obvious map between semantic actions (checks) and automaton states: every semantic action (check) occurs in an automaton state and an automaton state may have one semantic action (check). The semantic action (check) occurring in state n is stored in macro `\sa@`n (`\sc@`n). This macro has as many parameters as information bundles carried by the token preceding the semantic action. For example:

```
\defx{sc@42}#1#2{\isVarQ{#1}}
\defx{sa@42}#1#2{\memDir{#1}\aMemDir
   \emitCode{\lit{\aMemDir}}}
\defx{sa@49}{\newLabel\labelse \newLabel\labend
   \emitCode{\jzero{\number\labelse}}}
```

We will call these macros through

```
\def\semaction#1{\csname sa@#1\endcsame}
```

Due to the behaviour of `\csname`, an action can be called even if it does not exist. To exploit this circumstance, those states after a token that carries information but has no semantic action will have an explicit empty action with enough parameters.

A semantic action is executed when entering its state. A semantic check is executed before entering its state; if it returns true, its state will be entered; if it returns false, another possible next state will be tried.

Traditional parsers encode the automaton and semantic actions in a table. A loop uses the current automaton state and the next token to index this table, to perform some action, and to change to the next automaton state, and a stack is needed to store return states when entering a non-terminal. In a self-parsing implementation the current automaton state and the stack are in the global parser state: `\state` and `\stack`. The table becomes code; the action performed in the loop when looking at the token `\Tok` in state n is stored in the macro `\Tok@`n. Therefore, token `\Tok` behaviour is

```
\csname Tok@\state\endcsname
```

Since most entries in the table are just errors, memory can be saved not defining them. Checks for errors can be factored in the following macro:

```
\def\exe#1{\expandafter\let\expandadfter
  \aux\csname #1@\state\endcsname
  \ifx\aux\relax
    \errmessage{Unexpected token "#1"}\fi
  \aux}
```

So, the definition of `\Tok` can be simplified to

```
\def\Tok{\exe{Tok}}
```

Changing to another state, and pushing and popping states from the stack will be abstracted with `\toState`, `\pushState` and `\popState`. These macros are the best place to call semantic actions.

```
\def\toState#1{\def\state{#1}\semantic{#1}}
```

With all these helper macros, encoding the automaton table with a set of macros is a simple but boring task. For example, a **while** in state 53 only have to change to state 54

```
\defx{while@53}{\toState{54}}
```

But **while** can appear in other states; for example, nested inside another **while**, that is, after a **do** (state 56); in this case, it must push state 57 in the stack, change to state 54

```
\defx{while@56}{\pushState{57}{54}}
```

When nested **while** parsing ends, the state 57 will be restored from the stack and its semantic action executed so that the last instructions of the outer loop were generated. In a self-parser, tokens that may appear after a while statement are responsible for doing this. For example, **end** may appear after every statement type:

```
\defx{end@44}{\popState\exe{end}}
\defx{end@46}{\popState\exe{end}}
\defx{end@51}{\toState{52}\exe{end}}
```

```
\defx{end@52}{\popState\exe{end}}
\defx{end@57}{\popState\exe{end}}
\defx{end@60}{\toState{61}}
```

Notice, that **end** keeps rescheduling itself until reaching state 60.

Encoding entries in the automaton table that need a semantic check to be disambiguated is a bit more complex. For example, `Id` may occur in state 56, just after a **do**; changing to state 42 or 45 depends on the semantic check in state 42:

```
\defx{Id@56}#1#2{%
  \iftrue\semcheck{42}{#1}{#2}%
    \toState{42}{#1}{#2}\else
    \toState{45}{#1}{#2}\fi}
```

As can be seen, there is nothing radically new in this code. The classical parser engine with a monolithic table is split into a lot of little macros. Deciding what to do next does not rely on the parser lookahead but each token checks the correctness of its present occurrence, changes to the next state and invokes a semantic action. Therefore, TEX is a nice source language for a compiler-compiler. Before implementing parTEX, we checked all these ideas by hand; having done this boring work, we are eager to use it wherever possible. So now, we will input again the program pretty-printed on page 238, this time surrounded with `\beginPC`/`\endPC`, just here, only with the intention to execute `\power` to ensure that $3^9 = 19683$.

Other uses

We have used these TEXniques in other projects.

AGL is a small graphic library that our students (and we) write and improve year after year. It is written with the literate paradigm, using `noweb`. Moreover, there is a separate document "getting started and reference". In order to keep the reference up to date, and to allow concurrent development, the description of each element (function, type, constant, etc.) is embedded in the implementation; typesetting the implementation produces an up to date file to be input in the reference. To ensure full agreement and to save some typing, there is no place in a description to put its definition (the head of a function, the structure of a type, etc.); instead, while processing the reference, TEX opens source code files (built with tangle (`notangle`)) and looks for the definition of each described identifier. A fast scanner splits Pascal programs into tokens; then a search engine, organized like a self-parser, stores the tokens constituting each requested definition in a macro; finally, when typesetting an element description, these tokens feed a pretty-printer. So, the same scanner is composed both with a search engine and a pretty-printer. TEX process hundreds definitions in a few seconds, thanks to the fast scanner.

EXercita is a hierarchical, human-readable database of exercises. Every exercise has, in addition to the wording of the exercise itself, an author (or source), its objective and difficulty, and several solutions. A set of macros helps in extracting exercises to be included in a document. The macros to search databases use a self-parser.

HTEXML (HTML in TEX) is a work in progress to make TEX capable of type setting HTML directly. Almost all the processing work comes from HTML tags. It is important to do it as fast as possible because, although hand written HTML has few tags with few parameters, machine generated HTML has large numbers of tags with lots of (usually unnecessary) parameters. The simple syntax of HTML tags makes really ease to write a fast scanner. With the arrival of CSS a lot of new parsing capabilities are needed.

Conclusions

Fast scanners are clearly fast. We have only collected simple figures. For example, in my old Intel 80486, more than 1000 lines of Pascal code are scanned in 5.4 seconds, and pretty-printed in 4.6 seconds more. TEX typesets this same code (thinking that it is plain text) in 1.7 seconds and needs 0.9 seconds to process a file that only loads the scanner and the pretty-printer — so, parsing and pretty printing is only one order of magnitude slower. In general, it is astonishing to see TEX working so fast in every project were we have tried a fast scanner.

Self-parsing is a nice TEXnique to organize reusable parsers. It also allows an adaptable syntactic rigour. Its main drawback is the effort to build a strict self-parser by hand. Fortunately, parTEX removes this burden. Fast scanners, being an application of self-parser, should inherit this complexity; but writing a fast-scanner generator for TEX is a daunting task because each language has its own tricks. Fortunately, writing a fast scanner by hand is affordable because the lexical part of a programming language is simpler than its syntax. In addition, each processing kind needs a new parser, but the same scanner can be used once and for all.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Henry Baragar and Gail E. Harris. An example of a special purpose input language to LATEX. In

Proceedings of the TUG Annual Meeting, 1994.

[3] Julio Cortazar. *Historias de Cronopios y Famas*.

[4] Jonathan Fine. The `\CASE` and `\FIND` macros. *TUGBoat*, 1(14):35–39, 1993.

[5] Andrew Marc Greene. BASIX—an interpreter written in TEX. In *Proceedings of the TUG Annual Meeting*, 1994.

[6] Jean luc Doumont. Pascal pretty-printing: an example of "preprocessing with TEX". In *Proceedings of the TUG Annual Meeting*, 1994.

[7] Pedro Palao Gostanza and Manuel Núñez García. $\frac{\text{pa}}{\text{al}}\frac{\text{sc}}{}$: Formating pascal using TEX. In *EuroTEX*, 1995.

[8] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudack, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98. a non-strict, purely functional language. 1999.

[9] Kristoffer H. Rose. XY-pic user's guide. 1998.

[10] C.G. van der Laan. `\FIFO` and `\LIFO` sign the BLUes. *TUGBoat*, 1(14):54–60, 1993.

[11] Marcin Woliński. Pretprin—a LATEX 2_ε package for pretty-printing texts in formal languages. In *Proceedings of the TUG Annual Meeting*, 1998.

Pedro Palao Gostanza

A Device-Independent DVI Interpreter Library for Various Output Devices

Hirotsugu Kakugawa
Faculty of Engineering, Hiroshima University
1-4-1 Kagamiyama, Higashi Hiroshima,
Hiroshima, 739-8527 JAPAN
h.kakugawa@computer.org
http://kakugawa.aial.hiroshima-u.ac.jp/

Abstract

In this paper, we describe DVIlib, which is a device-independent DVI interpreter library written in C developed by the author. Since DVIlib is completely independent from specific output devices, new printer drivers and previewers (DVIware) can easily be developed. DVIlib is a set of functions to read and render DVI files.

To render a page, DVIlib generates a bitmap for each character in a page and calls a callback function to draw a bitmap on a device. Therefore, what a programmer must do to make a new DVIware program is to write device-dependent routines (initialization and drawing a bitmap on a device). Since DVIlib adopts VFlib as a font module, many font file formats are available, including PK, GF, VF, and Type 1. Thus, any DVIware that adopts DVIlib supports many font file formats.

We developed a program to convert from DVI to bitmap, printer drivers, and previewers for the X Window System. These programs are easily developed by adopting DVIlib.

Introduction

Since TEX is a de facto standard of typesetting software, a set of software for handling the typeset result (a DVI file) plays an important role to print and view documents in TEX. Such software is called DVIware.

In this paper, we introduce a new framework to build a set of DVIware, such as printer drivers and previewers. Many kind of printer drivers and previewers are required for the following reasons:

- Different printers adopt different printer description languages (escape sequences) to represent images to be printed,
- Different previewer programs are required for each window and desktop environment, or
- Novice users and expert users may require different user interfaces.

Although an interpreter for a DVI file itself is simple, development of a new DVIware program is not an easy task, since it requires complex internal modules for handling font files and the 'special' DVI instruction, for example, figures in Encapsulated PostScript (EPS), changes of text colors, and scaling texts. In addition, we expect that the preview image on a window system and the printed image by printer are the same, except for device resolutions.

Most parts of program code for previewers and printer drivers are the same, and therefore, developing previewers and printer drivers individually is not adequate. In [1], Beebe proposed a set of functions written in C that can be used as 'parts' to form a DVIware program. To develop a new printer driver or previewer, such functions are lexically included in a DVIware program; only the device-dependent routine needs to be developed. Since the same 'core' of DVIware (e.g., a DVI interpreter and a font module) is shared among various DVIware, each DVIware program has the same output and functionality, except for resolutions of output devices.

In this paper, we introduce a new framework to develop a family of DVIware. We developed a device-independent DVI interpreter library written in C named DVIlib. Different from the approach by Beebe [1], DVIlib provides a set of functions to be called by DVIware. Since DVIlib is completely independent from specific output devices, new printer drivers and previewers can easily be developed. DVIlib has following features:

- drawing EPS figures,
- handling change of text colors,
- scaling boxes, and

- support for various font file formats (GF, PK, virtual fonts, Type 1, etc.) by adopting VFlib as a font module [5].

Currently, xdvi, dvips, and ghostscript are widely used. Although a combination of them is a strong set of software for handling TEX DVI files, we may be bothered configuring this software (e.g., font definitions) consistently to obtain the same output. This problem often occurs when we use localized (e.g., Japanese) versions of this software.

DVIware which adopts DVIlib shares the same DVI interpreter and font rasterizer module. In addition, the same configuration file can be shared by previewers and printer drivers. Therefore, what we get on paper from a printer is exactly the same as what we see on a CRT screen (except for device resolution).

This paper is organized as follows. First we explain how we can use DVIlib to develop printer drivers and previewers. Then, the internal structure of DVIlib is explained. Next, we explain the supported features of the 'special' DVI instruction. Then, we introduce several printer drivers and previewers using DVIlib. Finally, we give concluding remarks.

Structure

In this section, we explain the structure of DVIlib. (See Figure 1.)

DVIlib is a library which is linked against an application program. It offers a set of functions to render a DVI file.

It uses VFlib [5] to obtain glyphs of characters in various font formats (GF, PK, Type 1 and Virtual Font, and more). VFlib uses a file named "`vflibcap`" as a font database. In this file, we can describe a font definition; for example, a Type 1 font is used for `cmr10.600pk`, or a PK font is used for `cmbx10.600pk`.

DVIlib has an interface to ghostscript, which is a PostScript interpreter, to render figures in EPS format.

DVIlib defines several callbacks to draw a page. Fundamental callbacks are (1) drawing a given bitmap on a page, and (2) drawing a rectangle on a page. DVIlib internally obtains glyphs of characters and converts EPS files to images. Thus, an application program can draw characters and EPS figures by simply implementing a callback to draw a bitmap on a page.

Features and specification of DVIlib are described in detail in the next section.

Using DVIlib

DVIlib provides a set of functions to obtain a page image from a DVI file. DVIware using DVIlib must obey the following framework to handle DVI files.

1. Initialize DVIlib by calling a function named `DVI_INIT()`.
2. Call a function named `DVI_CREATE()`, with a DVI file name as an argument, to create a DVI object. A DVI object contains various information about a given DVI file.
3. Call a function named `DVI_DRAW_PAGE()`. Arguments for this function include a set of callback functions and a page number to draw.

In DVIlib, an output device is abstracted by a structure `DVI_DEVICE` which is a data structure for callback functions and device-dependent parameters. It contains the following members, for example:

- resolution of a device,
- a pointer to a function to draw a bitmap on a page. (This callback is used to draw glyphs of characters and EPS figures.)
- a pointer to a function to draw a rectangle on a page,
- pointers to functions to draw a graymap and pixmap on a page. (If one of these callbacks is defined, it is used to draw EPS figures. Otherwise, a callback to draw a bitmap is used to draw EPS figures.)
- a pointer to a function to print error messages,
- a pointer to a function to change text colors,
- a pointer to a function to change background colors.

Since we can create multiple DVI objects independently and simultaneously in a single application program, for example, we can create a previewer that opens and displays multiple DVI files at the same time.

Now we explain details of three important functions.

`DVI_INIT(char *`*`vflibcap`*`, char *`*`params`*`)`
— Initialization function for DVIlib. The first argument *`vflibcap`* is the path name of a font database file called "`vflibcap`". It is used by VFlib (a font library) to resolve fonts used in DVI files. In a `vflibcap`, variables can be used to customize its contents at run time. For example, output device resolution is parameterized by a variable. VFlib has a feature to define values of variables when its initialization function is called. The second argument

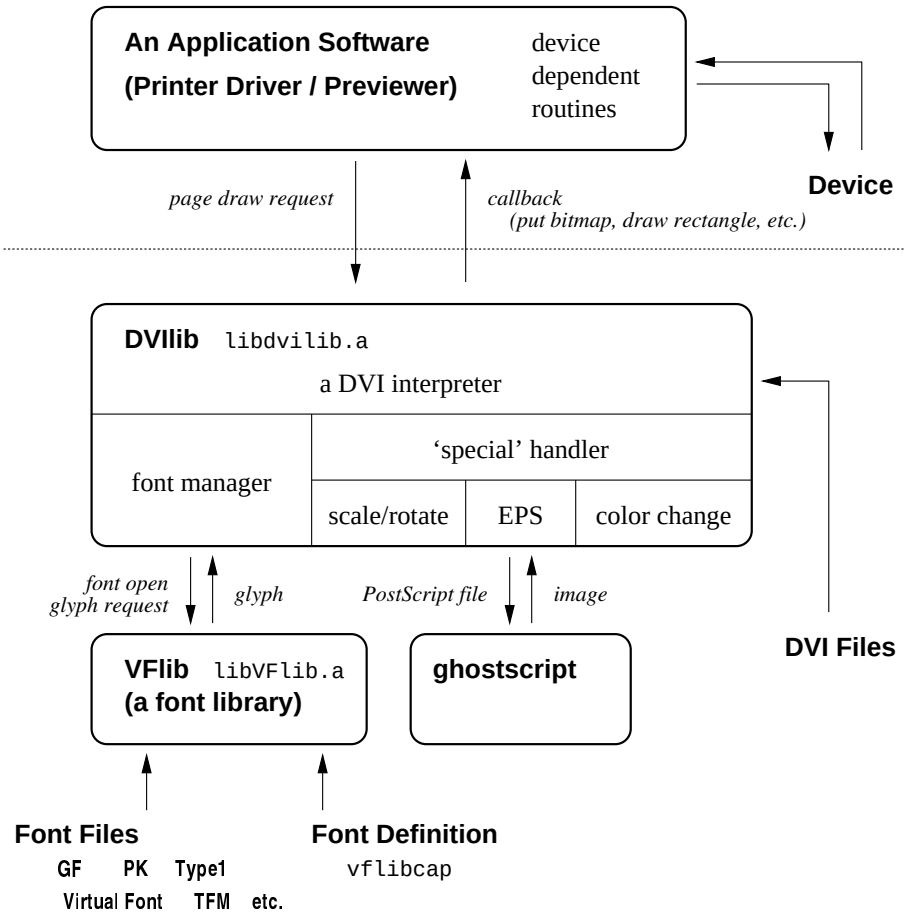


Figure 1: The structure of DVIware based on DVIlib

params is used to pass variables value of VFlib for runtime customization.

If null pointers are given for these parameters, default values are used, i.e., the default `vflibcap` file is used and no variable value customization.

`DVI_CREATE(DVI_DEVICE` *dev*`, char *`*file*`, DVI_PROPERTY` *prop*`)`
— This function creates a DVI object for a given DVI file *file*. The first argument *dev* is a set of callback functions.

The third argument *prop* is used to control the behavior of a DVI interpreter. Features of a DVI interpreter are characterized as *properties*, and they can be enabled or disabled. Data type `DVI_PROPERTY` is used to describe a set of properties to be enabled. DVIlib has a set of functions to operate a data of this data type.

The following features of a DVI interpreter are controlled by the third argument of `DVI_CREATE`. (Although there are other features to be controlled, they are omitted here.)

- Delay opening fonts until they are necessary. This property is effective for previewers to display the first page quickly, since not all fonts need to be opened to draw the first page.
- Invoke the ghostscript program immediately when a DVI object is created. Standard behavior is to invoke ghostscript when a DVI interpreter encounters the first EPS figure. This property is effective for previewers to display the first EPS figure quickly, since ghostscript may finish its initialization before the first EPS figure in a DVI file is encountered.
- Skip execution of the 'special' DVI instructions. For example, drawing EPS figures can be ignored.
- Print a list of fonts in a DVI file.

DVIlib has a function to change DVI properties after a DVI object is created. It can used in a previewer, for example, to control the behavior of a DVI interpreter interactively.

`DVI_DRAW_PAGE(DVI dvi, DVI_DEVICE dev, int page, double shrink)`
— A function `DVI_DRAW_PAGE()` plays an important role in DVIlib. When it is called, it invokes a DVI instruction interpreter for a given page of a DVI file. The fourth parameter *shrink* is the shrink factor; obtained bitmaps and their positions are automatically shrunk by DVIlib.

Whenever a DVI interpreter encounters a `SET_CHAR`, `SET_`x, or `PUT_`x DVI instruction[1], it obtains a glyph by calling a function to obtain the glyph of a character in VFlib [5]. Then, the callback function to draw a bitmap is invoked with the obtained glyph and its position on a page as arguments.

When it encounters a `SET_RULE` or `PUT_RULE` DVI instruction[2], it invokes a callback function to draw a rectangle with position, width and height of a rectangle.

Support for the 'special' DVI instructions are described later.

Callback functions

In this section, we describe callback functions. Fundamental callback functions are (1) drawing a bitmap, and (2) drawing a rectangle. Although many other callbacks can be defined, we explain only important callbacks here.

`dev_put_bitmap(DVI_DEVICE dev, DVI dvi, DVI_BITMAP bm, int font_id, long k, long code_point, long x, long y)`
— This callback function draws a bitmap *bm*. Its position is *x* and *y*. Other arguments are supplementary information, such as font number and character code, for example.

`dev_put_rectangle(DVI_DEVICE dev, DVI dvi, long x, long y, long w, long h)`
— This callback function draws a rectangle of width w and height h on a page. Its position is *x* and *y*.

DVIlib defines other useful callbacks.

`dev_put_graymap(DVI_DEVICE dev, DVI dvi, DVI_GRAYMAP gm, int font_id, long k, long code_point, long x, long y)`
— This callback is used to draw EPS figures in grayscale, if defined. If this callback is not defined, `dev_put_bitmap` is used to display EPS figures.

`dev_put_pixmap_rgb(DVI_DEVICE dev, DVI dvi, DVI_PIXMAP_RGB pm, int font_id, long k, long code_point, long x, long y)`
— This callback is used to draw EPS figures in color, if defined. It this callback is not defined, `dev_put_graymap` or `dev_put_bitmap` is used to display EPS figures.

`dev_color_rgb(DVI_DEVICE dev, DVI dvi, int f, double r, double g, double b)`
— Change text colors. Parameters *r*, *g*, *b* represents intensities of red, green, and blue.

`dev_message_error(DVI_DEVICE,DVI,char*,...)`
— Print an error message. This callback can be used to display an error message on a message dialog window, for example.

Skelton of a DVIware

Following is the outline of a simple printer driver program with DVIlib:

```
#include <dvi-2.5.h>

int  main()
{
  int   p;
  DVI_DEVICE  dev;

  /* Initialize DVIlib */
  DVI_INIT(NULL, NULL);

  /* Make a set of callback functions */
  dev = DVI_DEVICE_ALLOC();
  dev->h_dpi = 300;  /* 300 dpi */
  dev->v_dpi = 300;  /* 300 dpi */
  dev->put_rectangle = dev_put_rectangle;
  dev->put_bitmap    = dev_put_bitmap;

  /* Create a DVI object */
  DVI = DVI_CREATE(dev, file, NULL);

  for (p = 1; p < dvi->npage; p++){
    /* Clear page buffer */
    page_clear();
    /* Invoke a DVI interpreter */
    DVI_DRAW_PAGE(dvi, dev, p, 1.0);
    /* Send page buffer to a printer */
    page_send_printer();
  }

  return 0;
}

void
dev_put_bitmap(DVI_DEVICE dev, DVI DVI,
   DVI_BITMAP bm, int font_id, long k,
   long code_point, long x, long y)
{
```

[1] These are DVI instructions to draw a character on a page.

[2] These are DVI instructions to draw a rectangle on a page.

```
    /* Put a bitmap 'bm' at <x, y>. */
}

void
dev_put_rectangle(DVI_DEVICE dev, DVI DVI,
   long x, long y, long w, long h)
{
    /* Draw a rectangle of width 'w' and */
    /* height 'h' at position <x, y>.    */
}
```

Similarly, previewers can be built easily by adding a user interface to change pages to be displayed by pushing buttons by mouse, for example.

Support for the 'Special' DVI Instruction

When an interpreter encounters the 'special' DVI instruction, it invokes a handling routine according to a parameter string of the instruction.

- A figure in EPS.
  Pass the EPS file to ghostscript to render. Output of ghostscript is a bitmap; call a callback function to draw the bitmap. If a callback to draw a graymap (pixmap) is defined, PGM (PPM) format is selected as an output format of ghostscript. (Otherwise, PGM format is selected.) Then, a callback is invoked to place the obtained image.
- Change of text or background colors.
  If callbacks to change text and background colors are defined, they are invoked with RGB values for new color.
- Change of scaling factors.
  This feature is offered by `graphics.sty` and `graphicx.sty` packages (`\scalebox` and `\resizebox` commands). These macro packages generate embedded PostScript code as a parameter of the 'special' DVI instruction. DVIlib parses embedded PostScript code (by simple pattern matching) and generates scaled glyphs of characters and rectangles.

  Currently, change of rotation angles is not supported.

By this software architecture of DVIlib, every DVIware program can support EPS figures and scaled text and rectangles.

Multilingual Issues

John Plaice and Yannis Haralambous propose Ω as a multilingual extension of TEX [8]. They propose ΩFM (Ω font metric), which is an extended version of TFM. pTEX also defines the extended font metric JFM (Japanese font metric). Since VFlib supports ΩFM, JFM, and ΩVF(Ω virtual font), DVIware based on DVIlib can display and print DVI files by Ω and pTEX. (Currently, level-0 ΩFM is supported by VFlib, but level-1 is not.)

In the Japanese community, a variant of TEX named pTEX is widely used. pTEX supports Japanese characters and vertical writing directionality [2]. For supporting vertical writing, pTEX defines new byte code to change the writing directionality in the DVI instruction set, and a new register in the DVI virtual machine to hold the current writing directionality. DVIlib supports the extended specification of pTEX. This extension is encapsulated inside of DVIlib; DVIware based on DVIlib is not aware of this extension. Thus, all DVIware based on DVIlib can make use of such an extension.

DVIware with DVIlib

We developed several DVIware programs that adopt DVIlib.

- xgdvi
  A previewer on X Window System with GTK+ 1.2 toolkit. This software has a fancy GUI for novice users.
- spawx11
  A simple previewer on X Window System.
- dvi2rpdl
  A printer driver for Ricoh RPDL printers.
- dvi2escpage
  A printer driver for Epson ESC/Page printers.
- dvi2img
  A converter program to generate a PGM image file from a DVI file.
- dvifontlist
  A utility program to print a list of fonts used in a DVI file.
- dvispecials
  A utility program to print a list of 'special' DVI instructions in a DVI file.

xgdvi is implemented about 6000 lines of C code. It supports displaying color EPS figures. Since complex routines for DVI interpretation, handling EPS figures and font files are managed in DVIlib, most of the code of xgdvi is for the fancy GUI. A screen shot of xgdvi is shown in Figure 2. It supports multiple buffers: multiple DVI files can be opened simultaneously and they are switched without opening DVI files again.

dvifontlist and dvispecials are implemented by making use of DVI proporties; a DVI file is opened by disabling character rendering and enabling printing DVI file information. These programs are implemented in about 350 lines of C code.

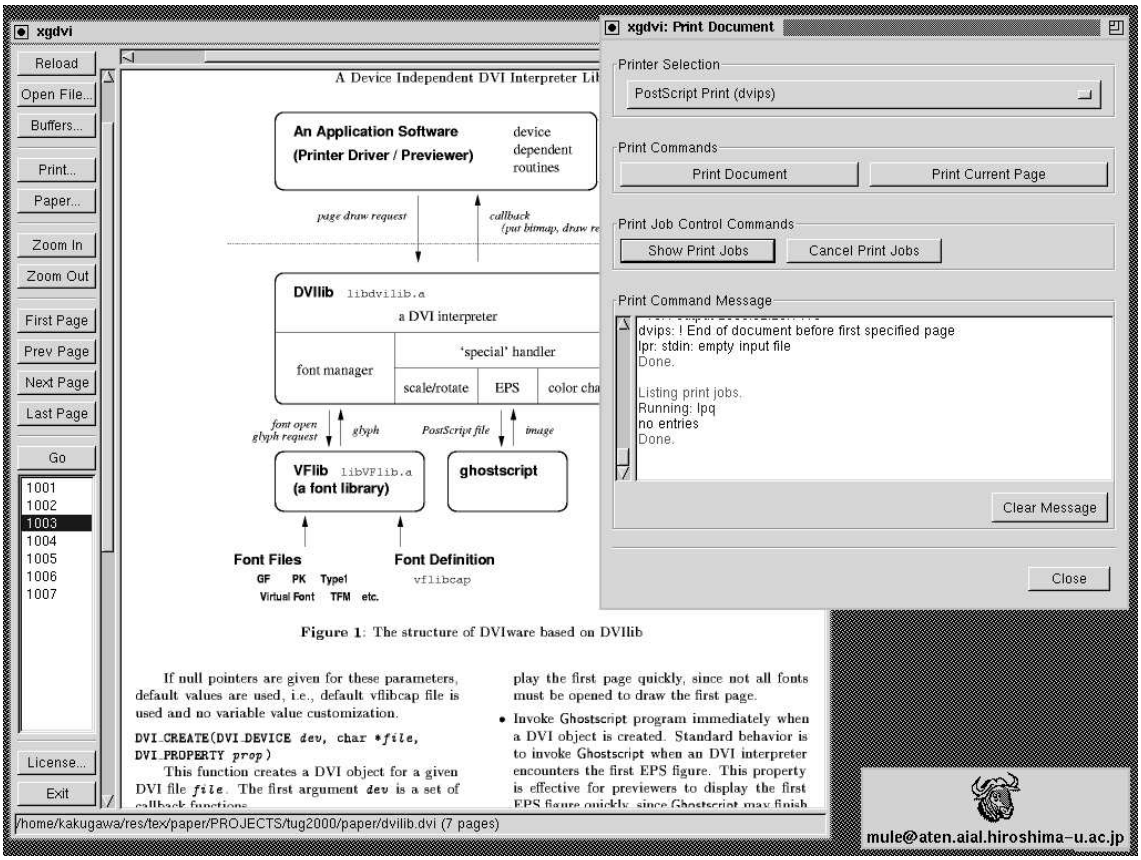


Figure 2: Screen shot of xgdvi

We developed another previewer spwx11[3] on the X Window System with minimum factionality, as a challange. It is implemented by only 140 lines of C code; the program consists of the interface to DVIlib and the X Window System to draw bitmaps and rectangles. Although this program does not support an anti-aliased display, it does support displaying EPS figures and various font file formats. spawx11 is a previewer on X Window System with an anti-aliasing display feature.

Typically, a printer driver can be implemented in about 500 lines of C code (if it does not support various printer description languages).

Conclusion

In this paper, we introduced DVIlib which is a device-independent DVI interpreter. Since it adopts VFlib for its font module, fonts in various font file formats can be available. We also developed several printer drivers and previewers that adopt DVIlib.

By adopting DVIlib, we can develop a simple printer driver within a day. When we develop a previewer, we can concentrate on our efforts for a fancy GUI. Since all DVIware shares the same DVI engine, the printed result is exactly the same as what we see on a display.

Currently, the following features are not implemented, for example.

- support for HyperTEX,
- full support for embedded PostScript literals (e.g., support for PSTricks packages),
- rotation of figures and texts (e.g., the `\rotatebox` command of graphics and graphicx packages).

DVIlib is written in C and about 6400 lines of code, half of which is for handling the 'special' DVI instruction. (VFlib, a font module of DVIlib, is written in C and about 33000 lines of code.)

DVIlib is a part of the TeX-Guy package which is a collection of DVIlib and DVIware based on DVIlib (including xgdvi and spwx11). DVIlib and VFlib are free software and distributed under the terms of the Library GNU Public License. DVIware such as xgdvi is also free software and distributed under the terms of the GNU Public License.

These programs have been tested on FreeBSD 3.2, Solaris 2.5.1; it is not difficult to

[3] spwx11 stands for "the Simplest Previewer in the World for X11".

port them to other Unix-like systems such as Linux. Visit our web pages: `http://TypeHack.aial.hiroshima-u.ac.jp/TeX-Guy/` and `http://TypeHack.aial.hiroshima-u.ac.jp/VFlib/` for further information and download.

References

[1] Nelson H. F. Beebe. DVIxxx. Available on CTAN as `/dviware/beebe/`.

[2] ASCII Coop. pTeX web page. `http://www.ascii.co.jp/pb/ptex/`. Publishing TeX for Japanese, with vertical writing.

[3] Roger D. Hersch, editor. *Visual and Technical Aspect of Type*. Cambridge University Press, 1993.

[4] Hirotsugu Kakugawa. The web page of VFlib. `http://TypeHack.aial.hiroshima-u.ac.jp/VFlib/`.

[5] Hirotsugu Kakugawa. VFlib — a general font library that supports multiple font formats. In *Proceedings of the 10th European TEX conference (EuroTEX 98)*, pages 221–222, March 1998.

[6] Donald E. Knuth. *The TEX book*. Addison-Wesley, 1986.

[7] Donald E. Knuth. *TEX: the Program*. Addison-Wesley, 1986.

[8] John Plaice and Yannis Haralambous. The Omega project home page. `http://www.gutenberg.eu.org/omega/`.

Hirotsugu Kakugawa

"Russian style" with LaTeX and babel: what does it look like and how does it work

M. Y. Kolodin, O. V. Eterevksy, O. G. Lapko, and I. A. Makhovaya
St.Petersburg, Moscow, Russia
myke@mail.ru
http://myke.webjump.com/tex/

Abstract

General and scientific typesetting in Russia has many specific features that should be supported by (La)TeX. Personal habits, local peculiarities and national traditions and standards are involved, as well as (La)TeX possibilities and limitations. The unified TeX implementation, stand-alone or based on babel, is needed to meet all the requirements. These peculiarities and implementation are discussed interdependently in the paper.

Olga Lapko

Mixing TEX & PostScript : The GEX Model

Alex Kostin & Michael Vulis
MicroPress, Inc.
68-30 Harrow Street
Forest Hills, New York, 11375, USA
Phone: 1 (718) 575 1818
Fax: 1 (718) 575 8038

support@micropress-inc.com
http://www.micropress-inc.com

Abstract

VTEX is a (commercial) extended version of TEX, sold by MicroPress, Inc. Free versions of VTEX have recently been made available, that work under OS/2 and Linux. This paper describes GEX, a fast fully-integrated PostScript interpreter which functions as part of the VTEX code-generator.

GEX offers one-pass compilation of text (TEX) and graphics and thus easy incorporations of graphics files (`.eps`) and inline PostScript code (PStricks, PSfrag) within a document. While it is this graphics support which is of primary interest to the end users, the presence of the PostScript interpreter within TEX and its ability to provide feedback to TEX raises interesting questions about mixing text and graphics in general and leads to new graphics-oriented packages.

This article serves as a short introduction to GEX, seeking to explain the design issues behind GEX and the extensions which now become possible.

Unless specified otherwise, this article describes the functionality in the freeware version of the VTEX compiler, as available on CTAN sites in `systems/vtex`.

What did GEX come from?

During the early work on the VTEX PDF backend circa 1998 it became apparent that the only way the backend can handle PStricks graphics is by incorporating a limited PostScript interpreter. GEX (which stands for Graphics EXtensions and is to be pronounced `g-e-ks`) arose primarily from the author's misguided optimism about the amount of work required. By the time GEX fully supported PStricks, the code amounted to more than 20,000 lines of `C++` code, supported almost the entire PostScript language, and even went beyond it. GEX has become powerful enough to handle not only PostScript files (`.eps`) but also the common inline PostScript graphics packages (PStricks, PSfrag, XYpic, or Seminar). In addition, it has become possible to design new macro packages with GEX in mind.

While the `.eps` file and inline PostScript inclusion is the main attraction to the end user, this article has very little to say about it. This is because from the end-user standpoint, using GEX amounts to using standard and familiar commands like `\includegraphics` or `\begin{pspicture}` and seeing the results appear as expected in the output. Instead, we will concentrate on the design issues and the extensions.

What is GEX ?

GEX is a graphics counterpart to TEX. The basic design assumes that TEX is responsible for handling of the text; GEX is responsible for processing the graphics components of the document. Both TEX and GEX contribute to the output; since the overall handling of the document is TEX's responsibility, TEX has overall control.

Usually, but not always, GEX functions within the TEX `\shipout` routine and accepts responsibilities which would otherwise be given to a DVI driver. In more interesting cases, GEX functions during the TEX formatting phase; when so doing, it is capable of returning information to TEX and thus influencing TEX formatting.

Since GEX may exercise subtle influence on TEX (load fonts, or change TEX registers), GEX is optional in VTEX implementations: the default operation of the program is with GEX off; it is enabled by a command-line switch.

Of the four primary output modes of the VTEX compiler (DVI, HTML, PDF, PostScript), GEX is currently supported with two: PDF and PostScript. The majority of GEX-related activities are identical in these two modes. Where a behavioral difference is desired, a macro writer can use the `\OpMode` count primitive (with magic values of 0,1,2 and 10 for DVI, PDF, PS, and HTML output modes).

In PDF mode, GEX is basically a PS↦PDF compiler; in the PS mode, it is a PS↦PS compiler which reinterprets input PostScript and produces output similar to what would be produced by printing PDF to PostScript, albeit faster, often tighter and cleaner. (One of the benefits of this in comparison with DVI↦PS drivers is the combination of the fonts and other resources that are often repeated in included `.eps` files.)

While GEX is a PostScript language interpreter, it is not 100% PostScript; there are subtle design differences, that while not impeding the ability of GEX to process standard PostScript code, allow new applications.

The basic design paradigms

During the design of GEX it has become apparent that a number of extensions will be needed to be added to TEX to support the extra functionality. In all cases, the basic approach was to try to keep the TEX syntax as close to the standard as possible, and avoid introducing additional keywords. Most of the TEX language extensions[1] are merely `\specials` which are understood and resolved by the `\shipout` code in VTEX. Thus, VTEX syntax would not have new words like `\pdfimage` or `\pdfoutline`; these would be backend `\specials`. In practice, we did end up with adding some primitives, but these were primarily new `count` and `skip` registers.

In designing the syntax of the `\specials` themselves, an attempt has been made to avoid dependency on the PDF output mode. This makes them either applicable or at least safely ignorable in other operation modes of VTEX (DVI, HTML), not just in the PDF and PostScript modes, where GEX is fully operational. Thus, VTEX's `\special` never uses PDF-specific code. While a direct write to the output is supported (with `\special{!=...}`, analogous to `\pdfliteral{...}` in pdfTEX), it is generally discouraged.

Finally, wherever possible, the `\specials` are screened from the user, mostly by means of extending the `graphicx` package.

[1] There are also PostScript language extensions in play.

How does it work

The basic model of TEX-GEX interaction is the two `\specials`:

- `\special{pS: .....}` with the argument containing valid PostScript code
- `\special{ps: .....}` with the argument being a name of PostScript file to include

When the backend sees one of these `\specials`, it passes it to GEX for compilation. (In PDF mode, with GEX off, it is simply thrown out; with GEX on, it is compiled. In PostScript mode, with GEX off, the parameter is pasted to the PostScript output, as in traditional DVI↦PS drivers; with GEX on, it is re-compiled).

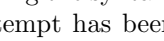
Prior to giving control over to GEX, VTEX updates the information in PostScript's graphics state (setting the coordinates for the current point, for example). Upon the return from TEX, the *relevant* parts of the PostScript graphics state are given back to TEX.

Because of the need to support inline PostScript packages the information about the current font is also shared between TEX and GEX. For example, passing

```
\special{pS: currentfont setfont}
```

to the PostScript interpreter is entirely legal (and is done by PStricks); but the design implication is that GEX is aware of the currently used TEX font and can access it by itself. Access may mean actually loading the font and executing the instructions in the font file; this would happen, for example, if one writes

```
\special{pS: gsave currentfont 2
  scalefont setfont 0.5 0 0 setrgbcolor
  [4 1] 0 setdash (Test stroke) false
  charpath stroke grestore}
```

which yields 

Observe that here we use `gsave`/`grestore` to screen TEX (and subsequent PostScript fragments) from the color and dash changes done in GEX.

Design implications: the font machinery is to be shared between TEX and GEX; GEX should be able to load TEX fonts and operate on them.

Solution: Provide all conceivably useful TEX fonts in Type 1. Extend GEX with command(s) for loading a font given its TEX name and point size (the `.settexfont` PostScript extension which loads the current TEX font into PostScript at the current size, as well as the `.loadfont` extension which allows GEX to select any TEX font by its name. The second extension is of use for MetaPost; see below.).

Unresolved issues: The ability to pass fonts to GEX is currently unsupported on TrueType or CID fonts (used in CJK PDF generation). Thus, no font effects are currently possible on Asian fonts[2].

Error handling

Unlike in TEX, error correction of PostScript code is hardly possible. Errors are therefore converted into TEX-style errors, followed by PostScript-style stack dumps.

For example, passing the following code to GEX

```
\special{pS: 1 2 movetoo}
```

results in

```
! PS interpreter error, code=21
 (Undefined name [movetoo]).
```

(This assumes, of course, that the `movetoo` name has not been previously defined.) GEX errors are usually fatal; while the TEX portion of the document can be compiled through the end, PostScript compilation is abandoned on the first error. Only with several common errors, like font unavailability, or leaving junk entries on the operand stack (as explained below), will GEX continue.

Interesting cases

While the above model is sufficient *for most cases*, there are unusual situations which arise in specific cases.

Typesetting text on a curve which is the activity of the `pst-path` component of PStricks provides one difficulty. In PStricks, this is implemented by redefining the `show` operator. In conventional PStricks, the expectation is that the redefined `show` should hack the code which originated in TEX, but now, after TEX↦DVI and DVI↦PS conversions, the code has become PostScript. In the VTEX case, we want PStricks to work within the TEX `\output` routine, where there is no PostScript code to hack.

Solution: The VTEX backend senses the redefinitions of PostScript text output operators like `show`; if it detects that `show` has been changed it temporary switches to PostScript generating mode; then passes the output to GEX for recompiling.

A similar situation arises when a TEX macro package "cuts out" a piece of PostScript code for reuse or discarding. Both PStricks and PSfrag do it by inserting a definition around PostScript code generated by TEX:

```
/something {
<ps code>
} def
```

Design implications: The TEX backend must sense when GEX is in such a "definition" mode, and switch to PostScript generation if needed. In the above example, upon processing

```
\special{pS: /something {}
```

GEX returns back an indicator that it did not fully handle the operator; only after

```
\special{pS: } def }
```

will the TEX backend be allowed to return to normal processing.

Transfer handling

⟨*int*⟩ **.enabletransfer** A problem which arises with some `.eps` images is the use of the `settransfer` PostScript and related operators. The problem is that these operators are used for both device-dependent and device-independent color manipulations. The first usage is more common and is essentially for minor color adjustments. In such situations the best strategy for producing device-independent PDF files is to disregard the transfer altogether. This is the default behaviour of GEX (and of the Acrobat Distiller).

However, in some (fortunately, rare) `.eps` files the same operators are used to effect major device-independent adjustments. An example of such an adjustment would be to invert a black-and-white picture; this can be done with the

```
{ 1 exch sub } settransfer
```

PostScript code snippet. Disregarding this code will produce an inverted image. Thus, both Acrobat Distiller and GEX allow the user to change their behaviour. In the case of Distiller, the override is a global Job option which will apply to all parts of a document; GEX allows you to override only the handling of an individual image. This is accomplished with the extension operator

```
.enabletransfer
```

With an argument of zero, `.enabletransfer` disables processing of `settransfer` code; a non-zero argument enables `settransfer` processing. Figure 1 is an example of a small `.eps` file that uses transfer code.

MetaPost support

While GEX can handle MetaPost-generated files, it is important to state that MetaPost outputs invalid EPS files. Rather than use the standard fonts or embed fonts into the EPS, MetaPost merely includes declarations like

[2] see `http://www.micropress-inc.com/CJK` for additional information on CJK/PDF support.

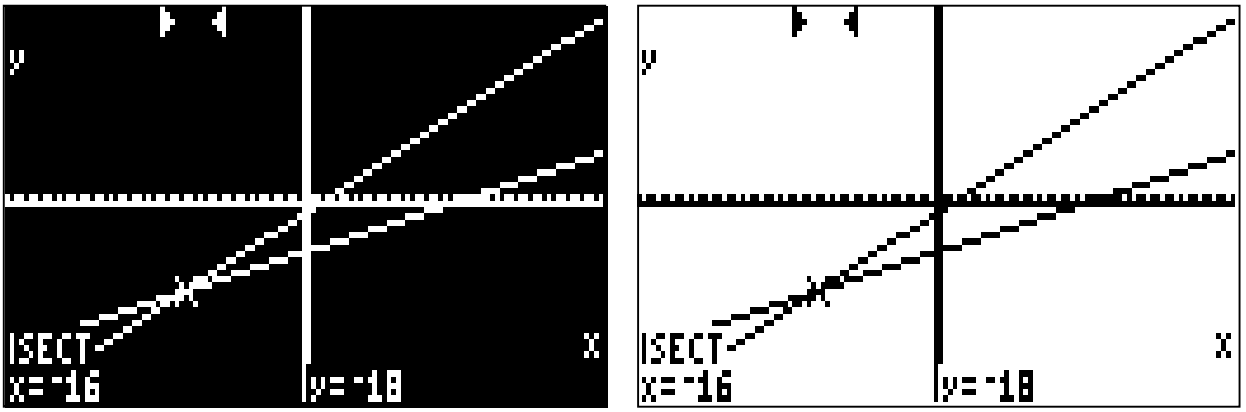


Figure 1: To the left the figure included with default settings, on the right the figure after enabling `settransfer`.

```
/cmr10 /cmr10 def
```

and expects post-processing to find and substitute the fonts. Instead of such post-postprocessing, GEX ignores (i.e., processes and discards the result) this declaration, but requires either explicit loading of needed fonts via the `.loadfont` extension

```
\special{pS: /cmr10 .loadfont}
```

(using one such command for each required font) or by enabling of the autoloading feature via the `.autofontload` extension

```
\special{pS: 1 .autofontload}
```

⟨***string***⟩ **.loadfont** loads the font by its TEX TFM name into the GEX font machinery and makes it available to `findfont` and related operators.

⟨***int***⟩ **.autofontload** If the integer argument is non-zero, GEX will query the TEX font configuration files when the `findfont` operator cannot resolve a font name. The default is *not to load* fonts implicitly and substitute Helvetica.[3]

These commands must be issued before a MetaPost-generated file is actually included.

EPS-specific problems

In the process of testing GEX over many hundreds of "real-life" `.eps` files, some common problems have been discovered. While these are technically bugs in `.eps`, they are common enough so workarounds had to be provided.

The majority of the `.eps` related workarounds (as well as many new options) have been incorporated as new keys to the `\includegraphics` command; this provides for an easy end-user interface[4].

Leaving entries on the PostScript operand stack is surprisingly common misbehavior which we have seen in files generated by many applications. If the `.eps` file is sound otherwise, it will be processed correctly; but an error may occur in handling the PostScript code that comes after.

Because of the common nature of this error (and especially because it causes the error message to point not to the culprit, but some later PostScript code), this GEX error is reported TEX-style:

```
! junk on PostScript stack, 4 items
? h
The PostScript code you just executed
has left some junk on the operand stack.
I'm taking it off; cross your fingers
and pray that this is all to it.
```

The fix required from the user is to add 4 `pops` at the end of the `.eps` image.

Degenerate matrices *Near-degenerate* matrix transforms often cause serious problems with the Acrobat's 16-bit computational limit. One can show that the problem is not solvable correctly in general; and Adobe Acrobat Distiller would fail on degenerate transforms.

The example file

```
% lwid.ps
0 0 moveto
gsave 100 200 lineto 2 3 scale
 1 0 0 setrgbcolor stroke grestore
gsave 200 100 lineto 0.5 0.3 scale
 0 1 0 setrgbcolor stroke grestore
```

[3] One of the subtle differences between GEX and PostScript is substituting Helvetica rather than Courier for fonts that are not available; in the author's opinion Courier is not a font to be used for *any* purpose.

[4] Special thanks to DC for providing the ability to define custom keys in `graphicx`.

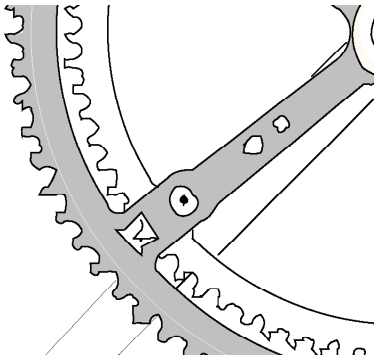


Figure 2: Fragment of a sample `gears.eps` included with `\includegraphics{gears.eps}`

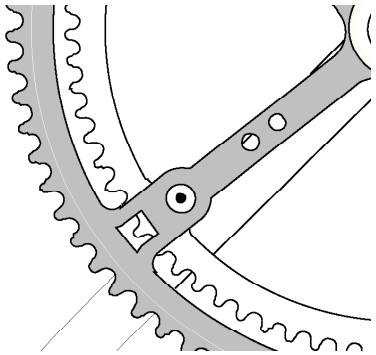
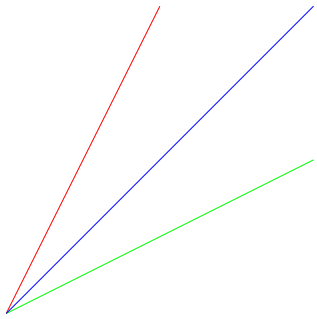


Figure 3: Same as Fig. 2 included with `\includegraphics[innerscale=4]{gears.eps}`

```
gsave 200 200 lineto 0 0 1 setrgbcolor
 [0.18672 -0.565306 0.87384 -2.64563 0 0]
 setmatrix
 stroke grestore
showpage
```

should produce three lines from the origin. Distiller, however, will miss the middle line. GEX, on the other hand, will produce correct output:



Near-degenerate matrices are not a perverted aberration: they tend to be generated by common software, such as CorelDraw. The particular set of numbers in the source above came from a Corel example.

While GEX does the work correctly in most cases (the precision limit in PDF guarantees that no PS↦PDF conversion can work correctly in all cases): some distortion of the line widths is possible and is not avoidable.

Level 1 strokeadjust Some graphics applications (for example, Freehand) output Level 1 PostScript code which fits the coordinates to an integer grid. This code, if executed literally, will produce rather disastrous results with GEX. Figure 2 shows one of the "real-life" examples.

The nature of the problem is a bug (or *feature*) in the Freehand adjustment code which does not bother to check for the device matrix and assumes that it corresponds to the output pixel resolution of 300 dpi or higher (which would imply a device matrix `[4 0 0 4.. ..]`). However, the GEX device matrix is chosen to be an identity, to avoid extra rounding by TEX's ⇔ GEX's coordinate translation. This causes extremely coarse coordinate rounding (72dpi) in the default case.

GEX's workaround to this problem consists of `\specials` that switches the device matrix to an appropriate one; this is incapsulated in the

```
innerscale=
```

option to the `\includegraphics`.

The corrected picture on Fig. 3 was processed with `innerscale=4`.

Level 1 / 2 differences While PostScript Level 2 is supposed to be a superset of Level 1, it is wrong to conclude that PostScript graphics displayed correctly on a Level 1 interpreter will appear the same way (or at all) on a Level 2 interpreter. It is all too common for `.eps` files to actually check the interpreter version and then execute totally different code, depending on the version found.

Both images came from the same PowerPoint-produced `.eps` file. Since in this case the end user might prefer the Level 1 appearance (but in some other `.eps`, perhaps in the same document, Level 2 may be required), GEX provides an ability to switch between Level 1 and Level 2 dynamically. On the lower level, this is done by the

```
N .setlanguagelevel
```

extension operator. Alternatively, the user might prefer to use the `gexlevel` option provided for the `\includegraphics` command and enter

```
\includegraphics[gexlevel=N]{paths.eps}
```

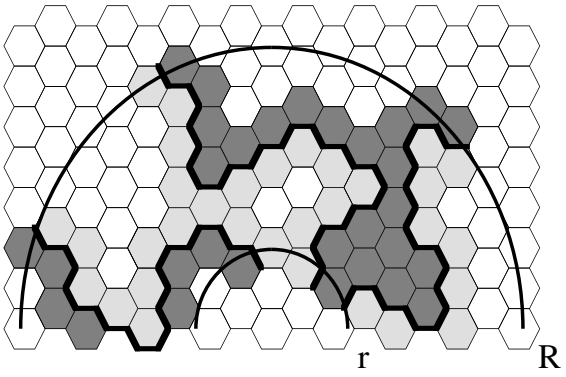


Figure 4: Level 1 appearance

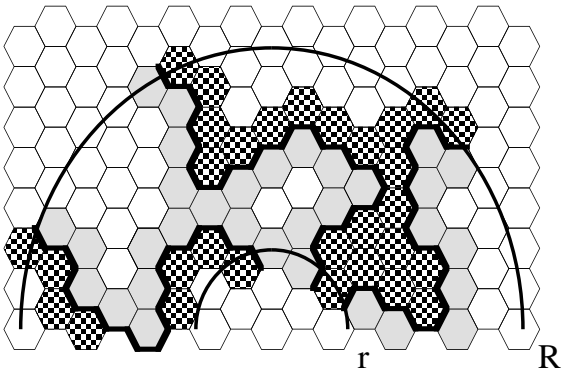


Figure 5: Level 2 appearance

Feedback to TEX

Since both TEX and GEX operate at the same time, it is possible to make them share information. While passing information from TEX to PostScript is trivial and has been done for ages (by putting them inside the PostScript language `\special`, in the case of VTEX, `\special{pS:..}`), getting information back from PostScript is new.

In GEX this is accomplished by PostScript syntax extensions that allow access to TEX `\toks` registers within GEX. The three new operators are:

- ⟨*int*⟩ ⟨*string*⟩ `.tkread` ↦ ⟨*int*⟩ ⟨*string*⟩

  where the ⟨*int*⟩ parameter should be between 0 and 65535 and designate a TEX token register[5]; the ⟨*string*⟩ parameter is the receiving string. In the output, the integer value is the new length of the string; the string contains the contents of the `\toks` register.

  Control sequence tokens are converted to spaces during `.tkread`; they are counted as one character for `.tklength`.

  During `.tkread` a `rangeerror` may occur if the `\toks` register contains more characters than can be placed into the receiving string; one can use the `.tklength` operator to find out how big the receiving string should be before allocating it.

- ⟨*int*⟩ `.tklength` ↦ ⟨*int*⟩

  where the ⟨*int*⟩ parameter should be between 0 and 65535 and designate a TEX token register; the output integer is the length of the contents of the TEX `\toks` register.

- ⟨*boolean*⟩ ⟨*int*⟩ ⟨*string*⟩ `.tkwrite` ↦

  where the ⟨*boolean*⟩ argument determines if the data should be appended to the `\toks` contents (`true`) or overwrite it (`false`); the ⟨*int*⟩ parameter is between 0 and 65535 and designates a TEX token register; the contents of the ⟨*string*⟩ parameter will be globally placed into the specified `\toks` register.

  Token strings produced by `.tkwrite` contain only tokens with TEX `\catcode` 12 (other).

The interface is deliberately kept very general; It is assumed that a TEX macro writer would unpack the `\toks` string as desired.

Here is how one can try to use GEX to generate a few random numbers:

```
\def\rand{%
  \special{pS: false 100 rand
    10 string cvs .tkwrite}%
  \the\toks100
  }

The numbers are = \rand, \rand, \rand.
```

The numbers are = 16807, 282475249, 1622650073.

Immediate execution

While the syntax above provides a way to deliver information from GEX to TEX, the information will arrive too late to be of much use. This is because `\special`s are executed during the page building (`\shipout`), when it is too late to use the returned data. For this reason, the example as written above will actually not work as specified.

While it is possible to overcome the problem with usual TEX multi-pass tools (the `.aux` file), we chose to instead enhance TEX with the

```
\immediate\special{...}
```

[5] Not a typo; the VTEX compiler has larger limits than other versions of TEX and `\toks10000` is legitimate.

form. The semantics here are identical to those when the `\immediate` command as used with the file operations which are already in TEX.

Besides making the example above work, the immediate form of `\special` proves very handy in a number of other cases, for instance:

- setting the background color for a page
- defining a PostScript header file
- thumb generation specials (PDF mode)

These actually cause difficulties for VTEX: a DVI driver can scan a page ahead to see if such `\special`s are present, but the one-pass nature of VTEX compilation requires them to be processed before the `\shipout` gets under way; the immediate form solves exactly this.

Creating objects in immediate mode

The GEX feedback can be used for many purposes, some of which can be accomplished by TEX means (if barely) and some which cannot be. One reason is because PostScript is a better computational language than TEX, and the `\immediate` form of `\special` makes it fully available to TEX. `trig.sty`, for example, is one casualty of this approach.

More interestingly, PostScript is more aware than TEX of the nature of the graphics objects that are in the document. For example, it is possible to use GEX to compute the exact locations of the extremes in a graph and then pass these locations back to TEX for placing of tags.

To allow development of this type of applications, we provide some additional machinery:

- It is allowed to have GEX compile and generate code for graphics objects in the `\immediate` mode; this code, however, is written to a memory stream.
- A memory stream can be frozen and closed with the `\special{ice}` command (naturally, another `\immediate`); when a stream is closed, its handle is provided in the `\pdflaststream` register.
- As graphics are drawn, the placement of the TEX tags can be computed as well, and reported to TEX via `\toks`.
- A stream is placed into the output page with the `\special{!stream ...}` command. Here we do not use the `\immediate` form, since the graphics should be emitted and properly placed during the usual `\shipout`.

Note: Code emitted prior to the `\shipout` cannot go to the output page right away since the formatting of the output page is not yet known. Thus, such code is emitted relative to the $(0, 0)$ origin; during the actual `\shipout` the code is shifted to the position of the `\special{!stream...}` command.

The technique outlined above has been successfully utilized in several new macro packages, including `vfplot`. They, however, use the extensible nature of GEX as well, and it would be prudent to explain this first.

Extending GEX

While in principle PostScript has as much computational power as a conventional programming language, writing computations in PostScript is much more time consuming than in, for example, C or Pascal. (Complex PostScript code may also take a long time to be interpreted.) The `.extend` operator in GEX seeks to add the extra power of conventional programming to GEX. In essence, a user can implement extra computational (or drawing) abilities in a compiled dynamic library (DLL for Windows/OS2, SO for Unix/Linux), then have these abilities available as new PostScript operators.

We call such addon DLLs **GEX plugins**. Syntactically, one writes

```
(pluginname) .extend
```

where `pluginname` refers to the name of a DLL (Windows/OS2) or a shared library (Linux/Unix) which contains the implementation of new extension operators. Upon encountering the above line, GEX will

- look for the requested plugin module
- ensure that its version matches the version of the GEX interpreter
- find out which new operators are implemented in the library, add their names to the PostScript namespace, and record the location of their implementation code.

Upon encountering a new operator, GEX calls the implementation code in the plugin.

GEX API

The GEX Application Programming Interface (API) represents PostScript internal operators as callback functions. For example, where a PostScript program would execute

```
10 20 moveto
```

a GEX plugin written in C shall do

```
GeXi->moveto(10,20);
```

The C/C++ API is specified in the `gexi.h` header file; it generally parallels PostScript drawing operators.

Rather than list the entire API here, we shall outline its principles:

- GEX API functions call the GEX kernel and are generally equivalent to PostScript operators.
- GEX API functions return 0 on success, and the PostScript error number on failure; it is the plugin's responsibility to handle the errors.
- GEX API functions cover most PostScript drawing abilities, but not text output. This is because plugins are not intended to do text formatting; this task should be passed over to TEX.
- The GEX API includes functions for working with the PostScript operand stack.
- An exception to the above is the `show()` function which is provided for the purposes of debugging only.
- Just like GEX itself, plugins can talk to TEX; this is done with the `tkwrite()`, `tkread()`, and `tklength()` functions.

For example, an extension operator `square` can be defined to draw a 10×10 square with C code like this:

```
int square(GEXI GeXi) {
   double x,y;
   if (GeXi->currentpoint(&x,&y)!=0)
      return error_nocurrentpoint;
   GeXi->lineto(x+10,y);
   GeXi->lineto(x+10,y+10);
   GeXi->lineto(x,y+10);
   GeXi->setrgbcolor(1,0,0);
   GeXi->closepath();
   GeXi->fill();
   return 0;  //success
   }
```

This example is, of course, useless: it is so simple that the task can be accomplished much easier in PostScript. However, the benefits of C programming become clear in more complicated cases.

PieChart

The first GEX plugin was the PieChart plugin. The implementation consists of

- `piechart.[dll|so]`, the extension library.
- `piechart.sty`, a matching LATEX 2ε style, to shield the interface details from the end user.

The code below shows how the end user may be using the plugin; a TEXnically minded person might also want to examine the sources which are available together with the GEX API description.

```
%% Define some colors
\definecolor{lightyell}{rgb}{1,1,0.75}
\definecolor{peach}{cmyk}{0,0.50,0.70,0}
\definecolor{orange}{cmyk}{0,0.61,0.87,0}
\definecolor{navyblu}{cmyk}{0.94,0.54,0,0}

\begin{center}
Shares of \TeX\ dialects:\par
\fbox{\begin{PieChart}[rt]{1.8in}
\PieSlice{lightyell}{65}{\LaTeX\ 2e}
\PieSlice{green}{20}{Plain \TeX}
\PieSlice{navyblu}{10}{AmS\TeX}
\PieSlice{yellow}{4}{\LaTeX\ 2.09}
\PieSlice{orange}{1}{Other}
\end{PieChart}}
\end{center}
%%
```

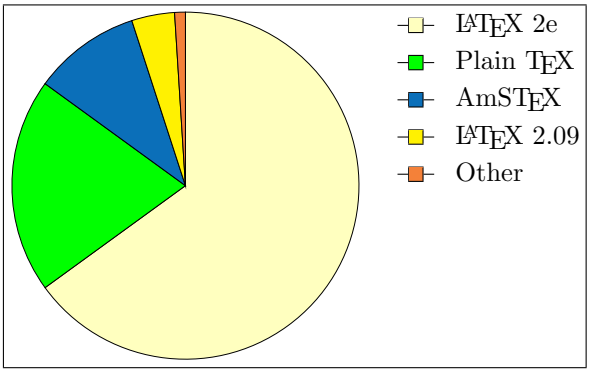


Figure 6: Shares of TEX dialects

A sample PieChart produced by this extension is shown in Figure 6.

Vchart

While PieChart is simple enough that it can be implemented in TEX/inline PostScript, its descendant, Vchart, breaks the barrier.

The Vchart package implements several formats of business graphs. Like PieChart, it is a combination of a plugin and a macro package.

To structure the user input, Vchart provides several environments. One defines the colors:

```
\definecolor{c1}{rgb}{.565,.592,1}
\definecolor{c2}{rgb}{.565,.184,.373}
\definecolor{c3}{rgb}{1,1,.753}
```

the headers:

```
\begin{header}{sides}
\entry[fillcolor=c1]{West}
\entry[fillcolor=c2]{East}
\entry[fillcolor=c3]{South}
\end{header}

\begin{header}{ABCD}
\entry{A}\entry{B}\entry{C}\entry{D}
```

```
\end{header}
```

and the data:

```
\begin{datatable}{example}
20.4 & 27.4 & 90 & 20.4 \\
30.6 & 38.6 & 35.6 & 31.6 \\
45.9 & 46.9 & 45 & 43.9 \\
\end{datatable}
```

and applies the `\DrawGraph` command.

The first graph in the series below has been produced with

```
\colorbox{grbkcolor}{%
  \DrawGraph{graphdata=example,
  graphtype=column,width=100pt,
  height=70pt,rowheader=sides,
  colheader=ABCD}}
```

The other graphs differ only in the `graphtype=` setting. The separation of data from the actual command allows to produce different charts from the same values.

Vfplot

The most powerful GEX plugin designed so far is Vfplot. The name stands for the Visual Function Plot; Vfplot converts functions given as formulas into the plots within the document. Unlike the facilities offered by standard plotting tools (MathCad or MatLab), Vfplot was specifically designed with TEX in mind; the plots it produces are visually compatible with the TEX document (plots use the document fonts and TEX-formatted text).

Like the plugins mentioned above, Vfplot comes with a comprehensive macro package (`vfplot.sty` for LATEX 2_ε and `vfplot.tex` for Plain TEX) which screens the plugin details from the end user.

Samples of inputs to Vfplot and its outputs are shown in figures 8, 9, and 10.

Vfplot and PSfrag

In principle, plots similar to Vfplot's can be also be achieved using a standalone math plotting system (like MatLab) in conjunction with the PSfrag package. The basic advantage of Vfplot is that the plot is an integral part of the TEX document; it can be changed by changing the plot code within a TEX file directly, or employing the Visual plot editor (see below). PSfrag, on the other hand, is essentially a write-once format, which requires a separate program for making the plot and additional manual work in setting the substitution tags.

However, PSfrag also has an advantage of being more portable; Vfplot (and plugins in general) are VTEX-specific.

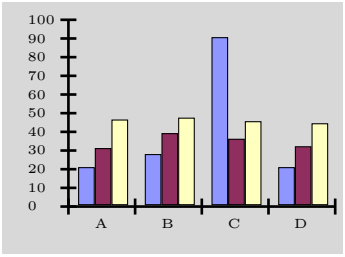
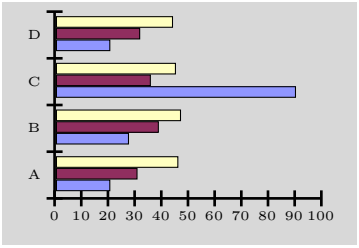
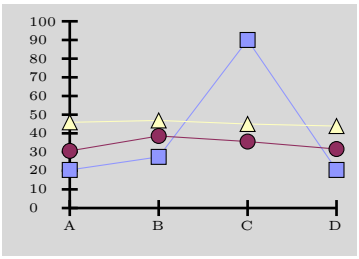
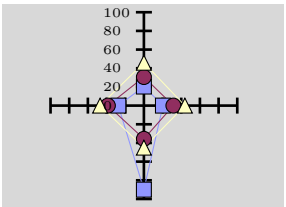
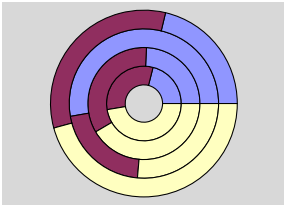
Graph Type/`type`	**Result**
Column graph/ `graphtype=column`	
Bar graph/ `graphtype=bar`	
dots and lines/ `graphtype=dots`	
Radar graph type/ `graphtype=radar`	
Doughnut graph type `graphtype=doughnut`	

Figure 7: Sample Vchart output. Notice that Vchart does not have a `piechart` type of graph; but a pie, after all, is just a degenerate doughnut.

```
\begin{plot}[legend=rt]{x-axis=MyAxis1,y-axis=MyAxis2,plotfill=CoorFill0}
 \function[linetype=MyLine1]
    [minlimit=-3.14,maxlimit=3.14,level=0,hatching=FunHatch,fill=FunFill]
    {x/2+sin(x)+cos(x^2) | x in [-5,5]}{$+{x\over 2}+\sin x+\cos x^2$}
\end{plot}
```

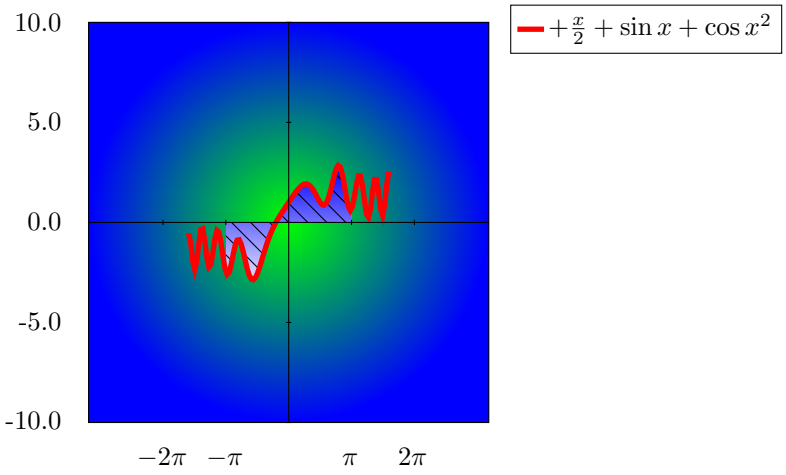


Figure 9: Vfplot drawing: Color Map

```
\begin{plot}{
x-axis=SineXAxis, % use a predefined axis
y-axis=SineYAxis,
gapsfill=Sunset2  % use a predefined
                  % gradient stretch
}
\function[linetype=MyLine]{sin(t)}{}
\end{plot}
```

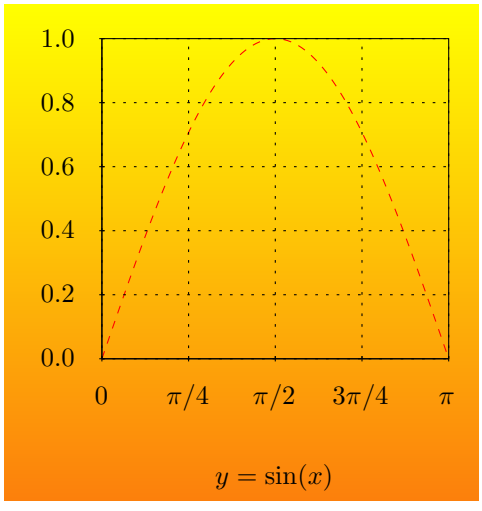


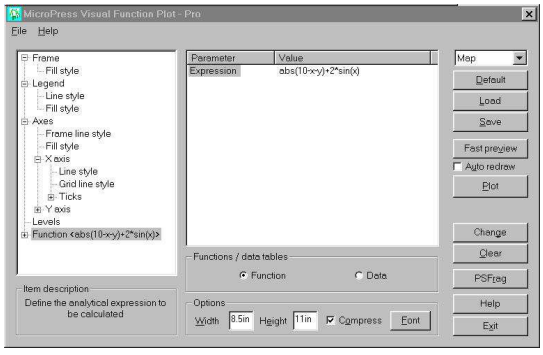
Figure 8: Vfplot drawing: 1D Plot

We, therefore, support exporting Vfplot environments into `.eps`/`.inc` file pairs; the `.eps` file contains the plot itself, while an `.inc` file contains a PSfrag wrapper for it.

Vfplot Visual Frontend

While it entirely possible to create Vfplot input "by-hand", the number of possible parameters is so large that a Visual frontend becomes useful. Such a frontend currently exists under Windows only; its functionality includes abilities to

- edit `plot` environments within TEX documents.
- visually manipulate all possible options of such environments:



- instant preview of the plot:

```
\begin{plot3d}{x-axis=AxX,y-axis=AxY,
   z-axis=AxZ,isolines=false}
\function[x-numpoints=60,y-numpoints=60,
   uppersidefill=MyFill3,
   lowersidefill=MyFill2,lineoff]
 { (1-x)^2+100*(y-x*x)^2 | x in [-1.5,1.5];
    y in [-0.5,1.5]}
\end{plot3d}
```

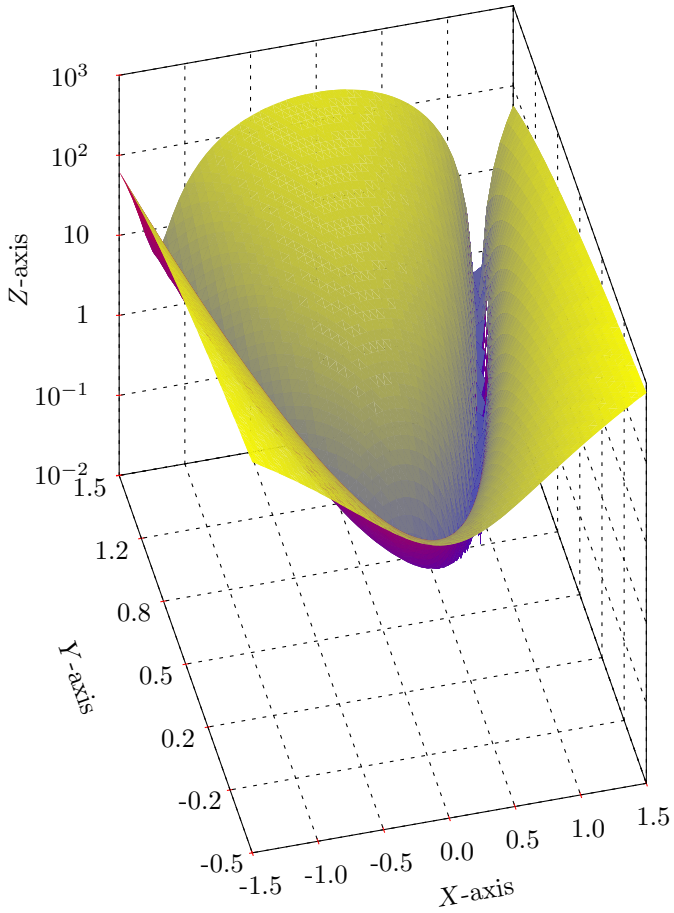
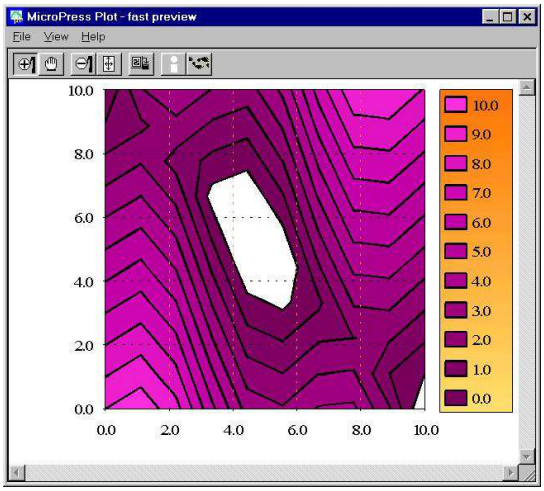


Figure 10: VFPlot drawing: 3D Plot. The Rosenbrock function, $z = (1-x)^2 + 100(y-x^2)^2$



as well as somewhat slower full plot. (The instant preview does not expand TEX notation).

- export plots in many bitmapped formats.
- export plots as `.eps`, or `.eps` together with a PSfrag header (`.inc`).

Image processing

One important place where GEX differs from ordinary PostScript is the image handling. In "normal" PostScript, `image` is a primitive operator. By default, it is the case in GEX as well; however, `image` is implemented as a combination of two new extension operators:

.loadimage does the unpacking and produces an *image* object on the operand stack.

.produceimage emits the *image* object to the output stream.

Thus, one can define

```
/image {.loadimage .produceimage} def
```

without changing the way `image` operates. Similarly, `imagemask` and `colorimage` use

.loadimagemask for the unpacking of the `imagemask` data

.loadcolorimage for the unpacking of the `colorimage` data

and are internally defined as

```
/imagemask{ .loadimagemask
  .produceimage} def
/colorimage{  .loadcolorimage
  .produceimage} def
```

By itself, this adds nothing. However, it opens a door for inserting a new operator between the two components of `image`:

```
/image {.loadimage myfilter
  .produceimage} def
```

Such an operator can manipulate the image data in memory.

A filtering operator as defined above cannot be written in PostScript — there is no *image* data type in the PostScript language; and from the point of view of the PostScript processor, *image* is just an `int`. A curious user can see this by trying

```
/image {.loadimage pstack
  .produceimage} def
```

However, image filters can be easily implemented via plugins.

Two plugins have been developed to perform image manipulations:

TransBit can alter the color model of the image. One of the applications is to convert color (RGB or CMYK) images to grayscale for printing purposes. TransBit functionality also covers the brightness and the contrast of the image.

Degrade downsamples the image; this can be used to decrease (often, greatly) the size of the resulting output file.

Both plugins take additional parameters. For example, if we want to brighten an image by 10 units, we would issue

```
\special{pS: (transbit) .extend}
\special{pS: save}
\special{pS: /image {.loadimage
  (toBright 10) transbit
  .produceimage } def
\includegraphics{mypic.eps}
\special{pS: restore}
```

The `save`/`restore` pair is needed to restore the original definition of `image`.

The functionality of both plugins has been incorporated in the `\includegraphics` command, so the end user will merely write

```
\includegraphics[brightness=10]{mypic.eps}
```

Note: The `graphicx` package automatically loads the required plugins upon seeing keys that are implemented in plugins.

Non-PostScript images

Although the above functionality applies to images stored within PostScript code (like the ones produced by `jpeg2ps`), we can easily extend it to the bitmapped image files.

The idea here is to be able to load image files into the GEX/PostScript environment; this is done with the `.readimage` extension operator. This operator takes a string argument with the image file name and converts it to an *image* object on the PostScript operand stack; such an object can be followed

up by a `.produceimage` or by imaging filter(s), and then by a `.produceimage`.

The end-user interface is again trivial. For example:

```
\includegraphics[colorspace=grayscale 16]
  {picture.gif}
```

will load the `picture.gif` file into GEX, and convert it to a 16-color (4-bits) grayscale using the TransBit plugin.

TransBit example

The examples in this and subsequent sections show the same image, `macaw.jpg`, processed with different `\includegraphics` keys. The original picture appears in the middle of the first example. TransBit related keys are `brightness`, `contrast`, and `colorspace`; these keys force the image processing via `.readimage`, followed by a plugin application.

Sample code

```
\includegraphics[width=1.3in,
  contrast=-0.3]{macaw.jpg}
\includegraphics[width=1.3in,
  contrast=0]{macaw.jpg}
\includegraphics[width=1.3in,
  contrast=+0.3]{macaw.jpg}
```

results in

`contrast=-0.3`

`contrast=0`

`contrast=+0.3`

Color model conversion would be, in particular, of use when the document is to be eventually printed on paper. For example, type

```
\includegraphics[width=1.3in,
   colorspace=bw]{macaw.jpg}
\includegraphics[width=1.3in,
   colorspace=grayscale 16]{macaw.jpg}
\includegraphics[width=1.3in,
   colorspace=grayscale 256]{macaw.jpg}
```

to produce

`colorspace=bw`

`colorspace = grayscale 16`

`colorspace = grayscale 256`

Color space conversion to grayscale also often substantially reduces the size of the output.

Degrade example

The Degrade plugin is triggered by the `degrade` key of `\includegraphics`; `degrade=1` corresponds to no downsampling.

```
\includegraphics[width=1.3in]{macaw.jpg}
\includegraphics[width=1.3in,
  degrade=0.6]{macaw.jpg}
\includegraphics[width=1.3in,
  degrade=0.4]{macaw.jpg}

\includegraphics[width=1.3in,
  degrade=0.3]{macaw.jpg}
\includegraphics[width=1.3in,
  degrade=0.2]{macaw.jpg}
\includegraphics[width=1.3in,
  degrade=0.1]{macaw.jpg}
```

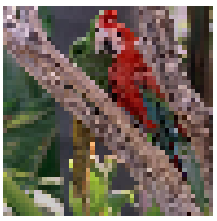
degrade=0.6

degrade=0.4

degrade=0.3

degrade=0.2

degrade=0.1



The transformations given above usually result in a drastic decrease of the size of the output. A minimal TEX source file which consists of a solo `\includegraphics` with different `degrade=` coefficients will result in PDF files of decreasing sizes:

Coeff.	File Size	
	Uncompressed	Flate-Compressed
1.0	1,278,158	1,029,377
0.6	459,780	392,863
0.4	204,804	181,256
0.3	115,908	104,714
0.2	51,970	47,932
0.1	13,952	13,242

Note: Downsampling can also be accomplished by means of the Discrete Cosine Transform; this is triggered by the `dct` and `dctquality` keys for `\includegraphics`. For photo-quality images this often leads to better results.

Color stack issues

The `color` package offers two distinct ways to maintain color: rely on the color stack in the backend (usually, a DVI driver), or — when such a stack is not available — emulate it within TEX.

As turns out, with GEX neither approach is fully adequate. The color stack within TEX is generally incapable of preventing color leaks from one page to another; but full use of the backend color stack is not possible since GEX already implements the full PostScript graphics state stack (GSS). While the GSS saves colors, it also saves the current point. This breaks some of the PStricks sub-packages, such as `pst-text` or `pst-path`.

The workaround used in GEX is to support both TEX and backend color stack approaches:

- `vtex.def` provides a macro `\if@colorstack`; when `true`, the `color` style uses the GSS stack; when `false`, `color` emulates the color stack with TEX means.
- By default, `\if@colorstack` is true; the driver color stack is used.
- Environments like `pspicture` are redefined to set `\if@colorstack` to `false`; this assures that PStricks are not broken.

Credits & Acknowledgements

The VTEX/GEX system itself was written by Michael Vulis. Most of the supporting macro packages were written by Alex Kostin. Vchart was written by Kirill Lebedev. Other plugins quoted in the article have their respective authors.

The authors wish to express thanks to

- Walter Schmidt and Taco Hoekwater for extraordinary efforts in making the freeware versions of VTEX possible.
- Denis Girou and Timothy van Zandt for cooperation and help in cleaning bugs in PStricks and Seminar which made their use with GEX possible.
- David Carlisle for providing an extendable version of the Graphics package which makes a natural interface to GEX features possible.
- Heiko Oberdiek for outstanding efforts in making sure that Hyperref manages all the multiple modes of VTEX.
- Many end users who discovered and reported bugs in GEX — thank you all — and please send more.

Abstract: The AsTEX Assistant and Navigator

Michel Lavaud
Département des Sciences pour l'Ingénieur, CNRS
F-91405 Orsay Cedex France
Michel.Lavaud@univ-orleans.fr

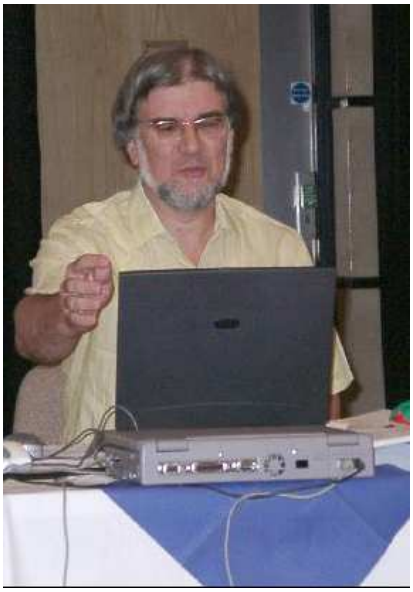
Abstract

The *Assistant AsTEX* is a program to aid the use of TEX under Windows 95/98/NT. It permits the entry of commands or collections of commands in any Windows editor via use of toolbar buttons rather than entering at the keyboard.

The *Navigateur AsTEX* allows navigation in `dvi`, `ps`, or `pdf` documents by clicking on elements provided in a navigation window.

It provides an interface independent of the program used to visualise the document. The programs that work with this standard include Dviwin, Dview, GSview and Acrobat Reader.

Michel Lavaud



LaTeX And The Personal Database

Bernice Sacks Lipkin
9913 Belhaven Road
Bethesda MD 20817
USA
bslipkin@erols.com

Abstract

Ignoring fixed-size and coded field formats, text databases can be viewed as either ASCII-delimited or ID-prepended. ASCII-delimited databases reserve a particular symbol as record delimiter, and another symbol as field delimiter. ID-prepended databases mark the start of each new field with stylized text unique to that field type. *Personal* means that the record can take any form the database owner desires, from the rigidly-structured, where the informational fields are in the same order in each and every record in the database, to the totally unstructured records of ordinary documents.

TADS is a set of integrated programs that manipulate the text in a personal database. Using TADS, field-specific LaTeX instructions can be made an intrinsic part of an ASCII-delimited database from its inception. To create a bibliographic data base and incorporate items from the database into manuscript text requires that the writer do two tasks:

1. Using TADS, the writer creates a customized data entry program that will prompt for keyboard input to create the field order he wants; e.g., *author, title, journal, etc.*. The data entry program writes a skeleton LaTeX file, complete with skeleton commands, one per field. These macro names, which also prepend the fields in each database record, define font size and shape.

2. In writing a document for publication, the writer alludes to citations in the document, using whatever phrases he can recall. The allusions are written within brackets; for example, *Smith et al <smith*199?*tadpole*> suggested...* Under a script that *stacks* TADS program modules with the appropriate options, these allusions are extracted from the document, and serve as wildcard match words to pull out records in the derivative database. Records are automatically sorted by number or by Name-Year and the numbers (or Name-Year) substituted for the allusions in the document. Fields can be rearranged or omitted in the bibliography; and the accompanying LaTeX field macros can be redefined for the font and style requirements of the particular journal.

Introduction

This paper presents some of the ideas that are explicated more fully in a monograph in preparation on human-aided computer manipulation of bibliographic databases.

TADS[1] is a set of gofer utilities for text processing. It has no artificial intelligence, no understanding of the meaningfulness of the text that it finds and manipulates. At the other extreme, in contrast to data mining, it doesn't pull out patterns statistically. It does recognize features of the text. It can pick out vowels or digits or all the letters of the alphabet or all the alphanumerics. It knows the start of a field from the rest of the field and where fields are located on disk. It can match words exactly or find a word in the field or do wildcard matching. With these simple skills, it does text substitution

1. TADS is an acronym for Text Analysis, Description and Synthesis. It is the latest reincarnation of a set of text manipulation and string processing programs. Variations of many of these programs were first written in the 1970s in Sail; the system was called *MaText*. A later version called *TXT* was written for the Microsoft C compiler and a DOS-Windows environment. It is documented in *String Processing and Text Manipulation in C*, which came with a copy of the source code for many of the functions on floppy. The book is currently being revised for a Linux-Unix environment. The programs now work with any size record.

and sorts records alphabetically or by class. There's even a job called *lazyboy sort*, where the machine figures out subclasses by examining the text. It compresses records vertically and links files horizontally. It makes statistical tables by tallying frequencies of linked words. It has no preconceived ideas about how to terminate a record or a field; you tell it the record delimiter, and if the record has more than one field, the field delimiter. It does insist that you reserve these marks — usually punctuation marks — as delimiters; that is, you can't use them in the body of the text.

TADS works with the text you give it, whatever it is. It works with your personal database, one that you design. If the input is a flat relational database — where the number of fields in each record is the same, the types of information in the sequence of fields are the same and fields are not subdivided — it can treat the fields as columns, and do all the *find* and *retrieve* tasks we expect from a search engine operating on a highly-structured simple database. But it is just as happy if it is directed to operate on a semi-structured database, where the first fields are predictable, and the later fields are unstructured — clinical notes for example. It can handle hierarchical fields, which include subfields or even subsubfields, a format that truly keeps data that belong together together. TADS treats ordinary manuscripts as records, if you declare the period as record delimiter. To partition a sentence, declare the comma as field delimiter. Obviously, not as many operations can be done on unstructured fields. But you can, for example, treat an entire book as a single record and extract all the test phrase that are bracketted by <> or [] or () or whatever.

In Figure 1, the first example is a flat database record, one where each field contains a single item of information. The second keeps all the data on jobs in one field. It is an example of a hierarchical database, where the field is subdivided into subfields, and the subfields are partitioned into subsubfields. The number of subfields usually vary from record to record. Subsubfields are generally rigidly structured because they are usually designed to provide quantitative results dependent on information that is ordered in time or in some other dimension.

Adding an unstructured Notes field to the first record would not affect its processing but would contribute to its functionality. It could act as a reminder of the times the consultant was used. The field could also be made private for evaluations and assessments by filtering the Notes field out before the file could be examined publically.

TADS searches are fast, but not as fast as *google* or *dogpile*. Much of what it does is done by commercial database managers. It does, however, have some nice features. It can handle any size record, any size field. It operates just on the fields you want processed. Any and all databases can be read and modified by you whenever you wish. (Of course, if you change a file that is searched by way of a TADS-created index, you will have to — or TADS will have to — redo the index.) The original database is always readonly, but TADS can send records that were modified to one file, records that were unaffected to another, eventually creating a genealogical *tree* that, properly manipulated, does the equivalent of AND, OR and NOT booleans.

When the programs were applied to databases with relatively simple structure — lists of bibliographic citations — I ran into a practical problem that was not large and interesting but small and annoying: the impossibility of picking a font format that would need no revision. For efficiency, markup instructions are usually embedded directly into the database text from the start. This can, however, produce problems downstream. What if one journal wants Volume numbers bolded, another wants them italicized, the next slanted. Redoing font instructions can be laborious, especially if you need to change the markup instruction in only one or two specific fields in every record, whenever you must use a different font. Alternatively, it is possible to maintain a database in unformatted text, adding exact instructions as needed. But this suffers from the same need to revamp much of the text.

More recently, I've started working with the ID-prepended format. An ideal example is an item from a MEDLINE download, such as the one shown in Figure 2. (I've omitted the Abstract field.) Keys to the article are listed individually in the Mesh Heading (MH) fields. The download format is stylized. Field names are two-letter, followed by two spaces, a hyphen and a space. What is a single hierarchical field in an ASCII-delimited record is split into individual fields. Field order is always the same. The Universal Identifier (UI) is always first. It is always followed by the author(s), the title, the language, and so forth.

Look at the figure. It is obvious that if you prepend a backslash to the field ID, substitute a left brace for the " - " and add a right brace at the end of the field, you have a LaTeX macro command with its text argument, a different macro command for each type of field. Naturally, you must define this new command, flesh out the font size and shape. By defining a macro for each type of field in the preamble, you can control the print characteristics

Flat field. The record delimiter is **!**. The field delimiter is /. They can not be used in the body of the record. Notice that the end of the final field is not a diphthong; i.e., it terminates with the record delimiter, not the field and record delimiters.

```
ConsultantFile0027/1994/
Smith, John M./electrical
engineering/Ph.D., U
Calif./99 First Street,
Lakeview 11111 WI!
```

Hierarchical field. The field delimiter is /, the subfield delimiter is % and the subsubfield delimiter is **$**. Two fields are shown: one simple, one hierarchical. John Smith may have held any number of positions, but each position subfield has a fixed number of subsubfields, three in this example — title, university and starting date.

```
John M. Smith, Ph.D./
Ph.D.$U. of Calif.$1957%
PostDoc$Yale$1959%
Asst Prof$Yale$1963%
Assoc Prof$Wisconsin$1967%
Prof$Wisconsin$1971/
```

Figure 1: Two types of database fields.

of the individual fields. This doesn't solve all the problems of modifying print appearance on the fly. But it helps.

The notion of tagging a record with a combined command name and field identifier when it is added to the database can be applied to ASCII-delimited records. The format I'm currently exploring is transitional; it has characteristics of both the ASCII-delimited and the ID-prepended formats. It still relies on a record delimiter, the ~, and field delimiters, the /, to isolate and partition the record. This is an example after it was keyed in under a data entry program designed specifically for these field types and the fields rearranged.

```
\ACNUM{DemoFile01}/\NAMEYEAR{Eisthen, 1992}/
\AUTHOR{Eisthen, H.L.}/\YEAR{1992}/\TITLE{Phylo-
geny of the Vomeronasal System and of Receptor
Cell Types in the Olfactory and Vomeronasal Epi-
thelia of Vertebrates}/\PAGES{1-21}/\JOURNAL{Mi-
crosc. Res. Tech.}/\VOLUME{23}/\ISSUE{1}/ / /
/ / /\NOTES{93004928}~
```

A database is stored in ASCII with prepended IDs, which are actually LaTeX macro command names. The end of a field in the current version is redundant: it has both a right bracket that we need for LaTeX anyways plus a field delimiter. In some files, I've eliminated the field delimiter altogether, using the } both for LaTeX syntax and field delimiter. But I won't do this in a general way until I've convinced myself that there's no interference with TADS in general, or at least in the major ramifications and combinatorics of using its modules in a sequence to get a particular result. The current format clearly does not interfere with the programming jobs that turn a database record festooned with tags and extraneous information into a well-behaved reference suitable for publication.

At this point, you are probably thinking BibTeX. BibTeX is indigenous to LaTeX. It has a multitude of formats, is easy to use and does much of its work transparently. TADS was not designed as a bibliographic database manager. It has little terminology specific to bibliography. In fact, it has little terminology. Labeling fields is a major innovation. But it remains a set of general programs, each with multiple options, that leaves it up to you to work out the correct sequence of programs and the appropriate options to do a particular task. Once a database is created and a map for obtaining the desired result is drawn — e.g., run *dork* task 3 with these options, then run *addtext* task 4 with these options, and so forth — it is simple enough to run the job from a script.

One such job is integrating a canonical bibliography and manuscript for a particular journal. To run the programs that do this, you first need to do two things:

Acquire a database.
To use TADS, you must add at least a record delimiter or, better still, a record delimiter and field delimiters, to each record in the file. That's it. TADS will extract a list of citations from a database, alphabetize it, number it, and substitute the numbers in the manuscript. But there are advantages to building in LaTeX macro names from the start. This paper describes a program that writes a data entry program to supervise the creation of a database from text

```
UI  - 20002969
AU  - Keverne EB
TI  - The vomeronasal organ.
LA  - Eng
MH  - Action Potentials
MH  - Afferent Pathways
MH  - Animal
MH  - Behavior, Animal
MH  - Chemoreceptors/chemistry/*physiology
MH  - Female
MH  - GTP-Binding Proteins/metabolism
MH  - Human
MH  - Hypothalamus/physiology
MH  - Male
MH  - Neurons, Afferent/*physiology
MH  - Olfactory Bulb/physiology
MH  - Pheromones/physiology
MH  - Receptors, Cell Surface/chemistry/genetics/*physiology
MH  - Signal Transduction
MH  - Vomeronasal Organ/anatomy & histology/innervation/*physiology
RN  - EC 3.6.1.- (GTP-Binding Proteins)
RN  - 0 (Pheromones)
RN  - 0 (Receptors, Cell Surface)
PT  - JOURNAL ARTICLE
PT  - REVIEW
PT  - REVIEW, TUTORIAL
DA  - 19991105
DP  - 1999 Oct 22
IS  - 0036-8075
TA  - Science
PG  - 716-20
SB  - M
SB  - X
CY  - UNITED STATES
IP  - 5440
VI  - 286
JC  - UJ7
AA  - Author
EM  - 200001
AD  - Sub-Department of Animal Behaviour, University of Cambridge,
      Madingley, Cambridge CB3 8AA, UK. ebk10@cus.cam.ac.uk
RF  - 56
PMID- 0010531049
PID - 7933
SO  - Science 1999 Oct 22;286(5440):716-20
```

Figure 2: An Item from a MEDLINE Download.

entered from the keyboard. It treats the keyed-in text as arguments to LaTeX commands. You can, instead, add macro commands to an existing database or to one you download from the Net or get from a scanned image.

Add allusions to the manuscript.
As you write your manuscript, you tuck snips of information about particular references inside brackets. The program extracts these allusions, bounces them against the bibliographic database, extracts the matched records, orders them by accession number or Name-Year, and substitutes the order tags for the allusions in the manuscript.

Using TADS, this is the sequence of tasks that results in a database:

1. Write a file of instruction records. We will call it *Lbiblio.ins*, but you name it as you like. Each instructional record provides the directives to control the entry of a single field in what will eventually be a single database record.
2. Run *Lquegen*. It will use the information in *Lbiblio.ins*, together with your answers to its online questions, to write you a customized data entry program. We will call the source code for the data entry program *Lbiblio.c*. You can run *Lquegen* as often as you like, using different instructional files to develop different styles of database.
3. Compile *Lbiblio.c*. You don't have to be a programmer. A Make file is provided.
4. Run *Lbiblio*. It has sufficient flexibility to create databases for different scientific disciplines, each with its own control file, each of which has the format given it by *Lquegen*. Actually, if you know C, you can go into the source code and make some minor adjustments to change the format.
5. *Lbiblio* needs to know where to send the processed text. Suppose we call this file *Lbib.db*. The first time you run *Lbiblio* to create records for *Lbib.db*, it ships the start of a LaTeX file with a set of commands, one per field, to *Lbib.db*. The macro name for a field is based on the prompt for that field.
6. Each time you run *Lbiblio*, it will store the values for the options you choose in the control file, the TRL file, whose name you specify. In this example, it's called *Lbib.trl*. The TRL file written the previous run stores the correct starting accession number for the current run. And it keeps a log of each run.

Lbiblio.ins, a file of instructions

Prompts and macro names are specified by giving *Lquegen* an Instruction File *Lbiblio.ins* that describes the prompt features of the database entry program as a set of records. Each field that is part of the record structure in the eventual database requires a separate record in *Lbiblio.ins*. If there will be 10 fields in each database record, you need to write 10 records. Figure 3 has an example of an instruction file; it has 13 fields in each record.

The delimiters in *Lbiblio.ins* must be the same as the ones you will use for the final database. We use '~' as the record delimiter and '/' as field delimiter. Notice that the record is terminated by '~', not '/~'

Each Instruction File record must have 5 fields. You need not fill in all the fields. A field may be empty, *but it must end in a delimiter*.

1. FIELD NAME. This *must* be a single word. When the data entry program is run, this will be the main prompt. The field name will also be the field's LaTeX macro command name, so only alphabetic characters are acceptable. I've used upper case on the field names, but there's no particular reason to do so.
2. EXTRA. This is optional extra prompt text. It helps conformity, if different people are typing in the data. It can remind them about punctuation and/or word order. It is reproduced as written, tabs, spaces, whatever.
3. DEFAULT ANSWER: If you just press ENTER, this will be the default text. The default answer is copied to the database as written. You can override it by typing in text. If you want no text in the field in some record, type a space and then press ENTER.
4. SUBFIELD: Is this field to be subfielded? YES/NO.
5. AND: Is this subfielded field to be ANDed? YES/NO.

The last two fields require explanation. The database(s) that will be created will be structured so that each field is the argument of a specific LaTeX command, which operates on the whole field. To make small changes on the text, the less information you store in a field, the better. In *Lbiblio.ins*, notice that VOLUME and ISSUE, which usually are neighbors in a citation, are in separate fields. Unfortunately, we don't usually write individual authors and editors in separate fields; and most of the microvariation in print appearance between journals is precisely in these two fields. Using subfielding and `\AND` is an attempt to solve one problem: does the

```
AUTHOR/:[FirstField] (SUBFIELD.) ex:Brown, A.//YES/YES~
YEAR///NO/ NO~
TITLE/: Title of article, NOT of the book//NO/ NO~
PAGES/: Separate with hyphen. Ex: 164-169//NO/NO~
JOURNAL/: Name of Book-Journal. Default:'Tech Manual X234L'/
          Tech Manual 234L/NO/NO~
VOLUME/: '3' is the default/3/NO/NO~
ISSUE/: TM234L-0ZN-X34523 is the default/TM234L-0ZN-X34523/NO/NO~
ISBN///NO/NO~
EDITOR/: (SUBFIELD.) Name of editor. ex: A.B. Smith//YES/YES~
CITY/: city where book was published//NO/NO~
PUBLISHER///NO/NO~
BPAGES/: number of pages in book//NO/NO~
NOTES/ keywords, code words//YES/NO~
```

Figure 3: A File of Prompting Instructions

journal want an *and* before the last author or an '&' or nothing?

A subfielded field is divided into subfields, just as the record is divided into fields. Each subfield is terminated by a character you reserve as subfield delimiter. Notice that, in the example, the AUTHOR, EDITOR and NOTES fields are subfielded. When *Lbiblio*, the data entry program, prompts for text in a subfielded field, it repeats the prompt over and over again, until you press ENTER with no previous text. Subfielding has different uses — separating authors to facilitate indexing, separating titles and subtitles — but its usefulness here is that, with recycling, the program knows when you've typed the last author.

YES in Field 5 requests that the program insert an `\AND` before the last subfield in the field (`\AND` is a command we define; the user may redefine it later). It can only be used with subfielded fields. (In this example, it isn't used in NOTES.)

Lquegen interprets the records in *Lbiblio.ins* and writes out its understanding in a file called, in this case, *Lbiblio.ins.decode*. In a Linux-Unix environment, you can pause while running *Lquegen*, read *Lbiblio.ins.decode* and compare it to what you wrote in *Lbiblio.ins*. This is its analysis for fields 1 and 6 in our example design.

```
Record [1]:
    FieldID = AUTHOR
    extra = :[FirstField] (SUBFIELD.) ex:Brown, A.
    defaultans = (null)
    subfld = YES
    AND = YES

Record [6]:
    FieldID = VOLUME
    extra = : '3' is the default
    defaultans = 3
    subfld = NO
    AND = NO
```

Lquegen, a program that writes programs

Choosing delimiters Conflict between different programming systems that operate on a database is almost unavoidable. There is no set of symbols that are exclusively and universally reserved for program instructions, with non-intersecting subsets for the different programming languages. What is text in one language is a directive in another. It is a happiness-making happenstance when the sequencing of programs is such that while the text is being manipulated by one programming language, it is transparent to the others, until it is their turn. But it takes careful planning to avoid difficulties. Particularly in choosing delimiters.

First, you don't want to use a character commonly used in the text itself. This lets out the comma and maybe the colon and semicolon. Invisible characters are poor choices. Second, the program reserves control-X, control-Y, `{` and `}`. All other control characters are OK if LaTeX, your machine, compiler and/or script don't reserve them. Records from the database will be formatted by LaTeX, so `%`, `\`, `&` and `#` are very bad choices. Avoid `$`, `^` and backspace, which are used in LaTeX math mode.

Good delimiters for a database that will be LaTeX-processed are: `/`, `*`, `@`, `‘` (octal 140), | (which prints as a dash), `’` and `"` (single and double quotes). The characters `=` and `+` are OK if you don't bring in arithmetic values.

Adding ID fields It is useful to have a permanent identifier (ID) to tag each of the records in the

database that *Lbiblio* will prompt you to construct. *Lquegen* can add two fields at the end of the fields designed by the Instruction File (see the example in the Section on Database Record Format). They can be transferred to the top of the record by *genio:rearrange*, a TADS program that rearranges and outputs the fields you specify.

An accession number field.
This is an easy way to provide an ID for each record. The first record in the file is 1, the second 2 and so forth. Because you may eventually be merging several data files, it is a good idea to have the ID also indicate the source of the record or some other name that tells you instantly where the record came from. You can tell *Lquegen* what text should precede the accession number when it asks for Leading Characters; e.g., *Molbiol2000:* or *ClinStudyAA*. This is a permanent tag for the record. It is not the accession number that is eventually given to records used by any particular manuscript.

You may, if you wish, make all the accession numbers the same size. If you say you want a minimum width of 6, say, the program will pad each accession number with zeros to make it 6-digit wide. (You can write a lot of records before you overflow a 6-digit width.) If you use the default, the program won't pad the number. It will use the actual length.

A Name-Year ID field.
The program can construct an ID field using the name of the senior author and the year of the publication. You will need to tell it the fields where these items of information are to be found. Actually, *Lquegen* only knows that it is to use the text of the first field up to the comma by default; or up to whatever size you stipulate. It uses all the text of the second field. Case can be set for the name: set all letters uppercase, set all letters lowercase, or leave case as is.

The style of the ID will depend in part on the text you tell the program to insert between the two items; e.g., *Smith2000*, *Smith:2000*, *Smith,2000*, *Smith, 2000*, *Smith:-2000*. Or you can put a large piece of fixed text between the name and year. And, if you wish, you can select a width, so that the name is chopped or padded to conform.

You can also vary the appearance of the records when they are finally ensconced in the database file. You can start each field on a separate line, if you wish. And you can set line width. If the text you type in has no space (at least 1 complete word) within the requested line width, the program will add a hyphen to the end of the line prior to outputting it, and will alert you to the hyphen by causing the bell to ring.

Lbiblio, a data entry program

By the time you type in the last answer, *Lquegen* has written you the C source code for a data entry program called, in this example, *Lbiblio*. You compile it by running a *make* file that comes with the program:

```
make FILE=Lbiblio
```

Once compiled, *Lbiblio* is immediately ready to act as a prompter for text data that you enter through the keyboard and to do housekeeping chores such as adding LaTeX macro names and record/field delimiters.

Initializing the data entry program The program needs some specific information before it can start its work. These values can be declared as a set of options on the command line when *Lbiblio* is run. The minus sign is the signal that the next letter is an option. There is no space between the option and the value. Options are separated by spaces. There is no input file.

OPTION	VALUE	MEANING
-h	YES or NO	Want extra information?
-o	⟨*filename*⟩	The output database file
-q	integer	Starting accession number
-t	⟨*filename*⟩	The Control-Log (TRL) file

-h Is initialization help wanted?
If the answer is YES, the program provides short essays at the start of the run. The default is NO.

-o The name of the output file.
If the file doesn't exist, the program will create it, otherwise it will append to the file; it never overwrites existing text. So it is possible to come back time and again to the same file to add references. Or you can use the same program, if it is general enough, as most bibliographic records are, for databases from difference scientific disciplines, each in a separate database file. There is no default. If the program is creating an output file, it prepends a LaTeX skeleton file, which includes one macro in the preamble for each field in a record. The macro definitions can be modified to meet the font requirements of any particular journal or document structure. The skeleton LaTeX file is shown in Figure 4.

-q The next accession number.
It is assumed you will be adding to the database. To anticipate, it is a convenience that you can direct the program to get the value from the control file. But you can also write it on the command line.

-t The name of the TRL file.
Whenever the program is run, it writes the current values of the options to the TRL file, so that it can read them the next run. The program also records other data: how many records were created, the date and time, and characteristics of the data entry program: line width, accession number width, the file delimiters, which fields are subfielded, and so forth.

There are various ways to start the program. If you just write the name of the program on the command line, you will be in Interactive mode. This is the simplest way to jumpstart the program, but it takes the most time. If you use the command line, the options can be written in any order.

Interactively. Just type the name of the program; i.e.,

```
Lbiblio
```

The program will query you for the values of the options. The very first time you run the program, there is no Control File, so you need to name it interactively. If there is no file with the name of the database, the program will create it. And any time you want to start a brand new database with a new TRL file, run the program interactively.

From the command line. Type the options directly on the command line; for example,

```
Lbiblio -hn -oqmolbiol.db -q40 -tmolbiol.trl
```

means you don't want extra explanation, the database file is called *qmolbiol.db*, the TRL file is called *molbiol.trl* and the first record you write this run will have 40 as its accession number. If there is no file with the name of the database, the program will create it.

From the Control File. Type the name of the control file option on the command line as the only option; for example,

```
Lbiblio -tmolbiol.trl
```

will use the values stored in *molbiol.trl* the previous run. The accession number will be correct, because at the end of a session, the program writes the starting accession number for the next session to the TRL file. The TRL file also maintains a permanent record of the previous data entry sessions that involve the data entry program, the database and the TRL file.

You can call some of the values from the TRL file and override other TRL file values by giving the parameters new values on the command line; for example,

```
Lbiblio -tmolbiol.trl -hy
```

requests that all the previous options, those stored in *molbiol.trl*, be used, except this time, you'd like some help.

You can maintain several databases on different subjects, each with its own database file and its own TRL file.

What the data entry program does

- prompts for the necessary information for the field, using the prompt text from the Instruction File
- writes the predefined default answer for the field (if there is one) to the database, if you press ENTER. You can override the default answer by writing in other text. To get an empty field in a field that has a default answer, type a space and then press ENTER.
- adds the specified record and field delimiters to each record.
- ignores any record and field delimiters typed in the body of the record
- adds the subfield delimiter to subfielded fields. It adds the `\AND` command to subfielded fields, if that was requested.
- writes out the full record to the database file with the specified line width
- writes out the record as a paragraph or writes each field to a separate line
- appends a stylized and padded Accession Number field to the record, if this was requested in *Lquegen*. This permanent accession number should not be confused with the numberings that will be given to records in the file that contains the citation list for a particular manuscript.
- appends a Name-Year ID field to the record, using the name of the senior author and the year of publication, if this was requested in *Lquegen*.
- writes a LaTeX header to the top of a newly-created database. The header includes a command macro definition for each field in the record, where the macro name for that field is the user-specified prompt in the first field of *Lbiblio.ins*. And it prepends the same macro

```
\documentclass[10pt,letterpaper]{article}
\usepackage{alltt}
\usepackage{multicol}
\usepackage[dvips]{graphicx}
\usepackage{color}
\usepackage{boxedminipage}
\usepackage{pandora}

%PAGE/PARA LENGTHS
\flushbottom
\parindent=0pc
\setlength{\baselineskip}{14pt}
\setlength{\parskip}{13pt}
%PAGE STYLE
\setlength{\textheight}{7.4in}
\setlength{\textwidth}{5.5in}
\setlength{\oddsidemargin}{1in}
\setlength{\evensidemargin}{1in}
%HEADERS/FOOTERS
\pagenumbering{arabic}
\setcounter{page}{1}
\pagestyle{myheadings}
\markboth{}{Demo Bibliographic Database}

\newcommand{\etal}[1][et al.]{\textit{#1}}
\newcommand{\AND}[1][ and ]{\textup{#1}}
\newcommand{\AUTHOR}[1]{\textup{#1}}
\newcommand{\YEAR}[1]{\textup{#1}}
\newcommand{\TITLE}[1]{\textup{#1}}
\newcommand{\PAGES}[1]{\textup{#1}}
\newcommand{\JOURNAL}[1]{\textup{#1}}
\newcommand{\VOLUME}[1]{\textup{#1}}
\newcommand{\ISSUE}[1]{\textup{#1}}
\newcommand{\ISBN}[1]{\textup{#1}}
\newcommand{\EDITOR}[1]{\textup{#1}}
\newcommand{\CITY}[1]{\textup{#1}}
\newcommand{\PUBLISHER}[1]{\textup{#1}}
\newcommand{\BPAGES}[1]{\textup{#1}}
\newcommand{\NOTES}[1]{\textup{#1}}
\newcommand{\ACNUM}[1]{\textup{#1}}
\newcommand{\NAMEYEAR}[1]{\textup{#1}}

%ATTENTION: Before you process the file
%through Latex, make sure there is an
%\end{document} after the last reference.
%Remove any \end{document} in the body of
%the file.
\begin{document}
```

Figure 4: The Top of the Database File.

name to the start of that field in each citation in the database. See Figure 4.

What the data entry program does not do Aside from adding the `\AND` command, the program does not modify the text you key in. On the other hand, as you key in text, you can use your own macros to reduce typing time and errors: macro names for long journal names, an alias for an author with a long and difficult name. As an example, I've included a `\etal` command (see Figure 4), because it is usually italicized. And it is inconvenient to format it after the fact.

In general, unless you write for a single journal, it's almost impossible to write a canonical style for names. One strategy is to adopt a style that works fairly well for the journals in which you publish. LastName-Initial is more common than LastName-FullFirstName, so it's fairly safe to use that style. It doesn't, of course, prevent the need for small polishings: one journal wants last name, just initials; another wants last name, followed by initials with periods. I use periods, because they are easier to erase than to add. *Science* uses an Initials-LastName format, which makes alphabetizing on the field difficult. You can, however, alphabetize on the NameYear field. If you publish often in both in a *Science*-style journal and in one that uses LastName-Initial, it might be worthwhile keying in author fields in both versions. Depending on where you send the article, you will use one or the other field in the final List of References.

If you plan on making microadjustments to the AUTHOR field in Emacs or some other text processor, it is a good idea to customize your data entry program to write each field in the record to a new line. You then search on the command name.

What you can do in response to a prompt

Create the citation fields, one by one. In our example, once the program is initialized, it will ask a series of questions for each citation, where a citation can be a reprint, a book or a technical publication. The typed responses will be confined to separate fields. Answers may be of any length, including zero length; i.e., the field can be empty, but the program will add a field or record delimiter. Depending on the Instruction File, a field can be simple; i.e., the prompt is displayed and you type in text. Then the prompt for the next field is displayed. Alternatively, a hierarchical field prompt can be displayed, where the prompt is repeated again and again, each time defining another subfield. To stop a subfield and/or a field, don't type any text — just press ENTER.

Jump between fields in the record. You can jump between fields in the record by typing ^Y (control-Y). ^Y+6 or ^Y6 will jump forward 6 fields. ^Y-2 will jump back 2 fields. You can not jump out of a record. If you jump forward some large number,

you will land in the last field of the record. If you jump backward some large number, you will end up in the first field of the record. It is not advisable to jump from a subfielded field; it can mess up the record.

Jumping forward to the last field is useful when, as in the example database template, you've essentially completed the citation for an article and want to skip the BOOK questions. You can't jump past the record, because the program does its housekeeping in the final field of the record, including chopping a clumped record into lines of the width specified in *Lquegen*.

Jumping back repeats previous prompts. When the program jumps back, it does not erase the intermediate fields. It just starts prompting from whatever field it has jumped to. If used judiciously, this feature lets you recycle a cluster of fields. It is a way of creating ID-prepended fields, such as those in the MEDLINE download in Figure 2. However, the record is no longer well-structured as an ASCII-delimited file.

Stop the program. Type ^X to stop the program. It will stop immediately. The best place to stop the program is at the prompt to the first field, so that the previous record has been completely processed. If you stop in the middle of a record, you will lose some text, and the record will be incomplete.

Database Record Format Next is an example of two database records that were keyed in under the control of *Lbiblio*, which was itself created using the example instruction set. The first has a single author and no subfields; the second one has a subfielded AUTHOR field. Notice that the fields between the ISSUE and NOTES fields are blank. They don't apply to a journal article, so you would want to skip to NOTES. The number in the NOTES field was taken from the MEDLINE ID for the article. NOTES is also a good place to store the authors' first names, for the few times some publication will request full first names. And it can be utilized as a depository for keywords that can later be used for cross-indexing citations and sorting them by subject matter. The NOTES field can also serve to indicate the physical location of the reprint, editorial comments, and so forth. The two last fields were added by the program. Records are stored as shown. For publication, fields will be extracted, rearranged and beautified, using available TADS routines.

```
\AUTHOR{ Eisthen, H.L. }/ \YEAR{ 1992 }/
\TITLE{ Phylogeny of the Vomeronasal System
and of Receptor Cell Types in the Olfactory
and Vomeronasal Epithelia of Vertebrates }/
\PAGES{ 1-21 }/
\JOURNAL{ Microsc. Res. Tech. }/
\VOLUME{ 23 }/
\ISSUE{ 1 }/ / / / / /
\NOTES{ 93004928 }/
\ACNUM{DemoFile01}/
\NAMEYEAR{Eisthen, 1992} ~
\AUTHOR{ Freitag, J. @ Ludwig, G. @ Andreini,
I. @ Rossler, P. @ \AND Breer, H. }/
\YEAR{ 1998 }/ \TITLE{ Olfactory Receptors in
Aquatic and Terrestrial Vertebrates }/
\PAGES{ 635-650 }/
\JOURNAL{ J. Comp. Physiol. }/
\VOLUME{ 183 }/
\ISSUE{ 5 }/ / / / / /
\NOTES{ 99056834 }/ \ACNUM{DemoFile04}
/\NAMEYEAR{Freitag, 1998}~
```

Writing The Manuscript

This is a short manuscript that illustrates the technique for writing allusions, using ordinary wildcard syntax: a ? allows any single letter in the ? position; a * allows any amount of text or no text to intervene between the two neighboring text phrases. The spelling error in the last line is deliberate.

> Phylogeny of vertebrate pheromonic sensory systems is complicated by the proximity and similarity of the adjacent but distinct olfactory system [@??sthen*vertebrate*olfactory]. The presence of sex pheromone systems in goldfish and the anatomic analogies of distinct olfactory systems [@Dulka*sex*pheromone] clearly establish both the antiquity and the complex olfactory/brain relationships that seems to characterize most if not all vertebrates [@??sthen*microsc], [evolution@vertebrate@olfact]. The hypothesis that the Class II receptors are specialized for recognizing volatile odorants is questionable since some fish, e.g. Latimeria, possess both classes [Freitag*aquatic*vertebrate]. The presence or absence of an accessory olfactory bulb is not in and of itself sufficient to affirm or deny a functional vomeronasal system in a given species [@bhatnag?r*diversity*mammalian], [bhatnag?r*bats*phylogenetic]. Attempts to infer the form of the earliest vertebrate pheronomic structures by comparative anatomy of hagfish and lamprey are made difficult in that

the necessary physiologic data on these forms are not available [sennsory biology].~

All the text phrases must be found in the citation for a match, so an allusion is intrinsically a boolean AND. A difficulty with this technique is that the text phrases that make up any wildcard match word must be sequential. This is fine only if you remember the exact field sequence. So there is an alternate wildcard format: an @ prepends the word. The @ syntax tells the machine to match each phrase in the allusion from the beginning of the field.

I've elected to use square brackets, but any bracket pair will do to delimit the allusions. An option in *dork:keepbracketedtext* leaves the empty brackets in the manuscript, when it strips the text from the brackets. So the square-bracket format is good for Name-Year tags.

For a number tag, I'd still use the square bracket but I would write ([⟨*text*⟩]), and use the option that deletes the text brackets; the square bracket would be deleted, not the parens. ([@??sthen*vertebrate*olfactory]) might eventually be written as (6).

If an entire manuscript is to be searched for allusions, add a single, unique record delimiter at the end. The file can now be described to the program as a single-fielded single record. Size is not critical for journal articles. But one could encounter problems if you were to treat an entire book as a single record. However, I've dummied up a 7 megabyte record by repeating a real book several times. No problems were encountered.

Linking Database and Manuscript

Various TADS modules come into play. This is the general plan. Recall that the input file is *always* considered *read only*. Any modifications are reflected in the file where the processed records are shipped.

Using *dork:keepbracketedtext*, the allusions are stripped out and reappear in some output file, one per line,

The allusions become a list of wildcard match-words that *finder* compares to records in a single database or in a database that is a virtual merge of records from several databases.

What constitutes a good allusion? The answer is tautological: a good allusion is one that is sufficient to attract the reference you want and only the reference you want. In practice, these are some of the types of errors that are encountered when the Citation Allusion (CA) in the Manuscript (MSS) is pitted against the Citations List (CL) in the source database.

1. The citation is multiply listed in CL.
2. The citation is not listed in CL.
3. The citation is incorrectly written in CL.
4. The CA is incorrectly spelled in MSS.
5. The CA informational items are incorrectly concatenated.
6. The CA is inaccurate and retrieves no citation from CL.
7. The CA is so general it retrieves multiple citations.
8. There are multiple CAs to the same reference.

Depending on the circumstances, the citation is not retrieved or unrequested citations are retrieved or a correct citation is repeated.

You can increase the probability of accurate retrieval by utilizing LaTeX macro names. Instead of writing @199?*green*pheromone*goldfish, you would write @\date*199?*\author?green*\title{pheromone*goldfish. But some people object to doing this while in the throes of creative writing.

The next step — checking the allusions agains the retrieved citations — is crucial. And it requires your participation. *finder* ships copies of the citations it matches to a file. The key, i.e., the allusion that identified the bibliographic reference is prepended as a field to the citation.

If the number of citations extracted equals the number of CAs, it may mean we have no error. On the other hand, sources of error could balance out so that the number of CAs equals the number of citations. So we can not rely on counting the number of citations and simply comparing this to the number of CAs.

TADS can present the original list of allusions and the list of citations in ways that facilitate comparison of the lists. *genio:rearrange* will split off the allusions field to a separate file, so the original list of allusions and the allusions prepended to the retrieved citations can be compared. *alp* sorts lists alphabetically; using any of the fields as key. *squish* eliminates duplicates. Spelling errors may have to be corrected 'by hand'. When you are sure the corrected allusions and citations correspond in a one-to-one fashion, hand the text back to TADS. The program tags the final list of retrieved citations in an orderly sequence. You make two separate choices:

1. You chose between listing the references by first appearance in the manuscript or in alphabetical order. The same choice applies if you list references by chapter in a book.
2. In either case, you choose whether to number the records sequentially or by Name-Year ID.

The order tag — accession number or Name-Year ID — is substituted for the allusions in the manuscript. This is done in four steps.

1. The citations can be left in the order in which they appear in the manuscript. Or they can be alphabetized using *alp*. If they are alphabetized, it is usually by author field or by Name-Year ID.
2. If ordering is by Name-Year ID, the order was done in the previous step. You will, however, need to add a final letter to the ID, if there is more than one publication by the senior author for that year. Alternatively, *addtext:acnum* can add an accession number to each record. The accession number becomes the first field in the expanded record.
3. *genio:rearrange* creates a derivative database of records, each with two fields: the allusion and the accession number (or Name-Year). The allusion and number are now considered a substitution pair. The entire file of records has become a list of substitution pairs.
4. *addtext:sbstitute* brings in the file of substitution pairs and substitutes accession numbers or Name-Year IDs for allusions in the manuscript.

The List of Citations needs to be dressed up for publication. *genio:rearrange* extracts particular fields in each record in the order you specify. Case can be modified on a per field basis. Revamping the macro definitions in the preamble will take care of the general appearance of the font. A beautify program *addtext:mssformat* substitutes commas for (sub)field delimiters, periods for record delimiters. Currently, this step usually calls for some small adjustments in the text — such as adding or removing commas and periods in the AUTHOR field, adjustments that are not easy to make globally. I work in Emacs, using the macro name to get to the right field per record and make changes locally — changing case, deleting periods and transposing words, using Emacs commands. Writing the fields in the records on separate lines simplifies the work considerably.

The list of references is appended to the manuscript, `\end{document}` is inserted, and the manuscript is ready.

Conclusion

In dealing with a database you have designed yourself, there are non-trivial advantages to prepending a field name that is also a LaTeX macro:

1. A prompting program instills style conformity in the records of the database and reduces transcription errors.
2. You can design a canonical record style across multiple databases.
3. You can manipulate font size and shape by field by redefining the field macro in the preamble.
4. You can use field names as part of a wildcard search of the database to reduce errors in matching an allusion to the correct (read *wanted*) record in the database.
5. To effect small text changes while in a text editor such as Emacs, you can get to the right field by searching on the field name.

Bernice Sacks Lipkin

Formatting documents with floats
A new algorithm for LaTeX 2ε*

Frank Mittelbach
LaTeX3 Project
frank.mittelbach@latex-project.org

Abstract

This paper describes an approach to placement of floats in multicolumn documents.

The current version of LaTeX was originally written for single-column documents and extended to support two-column documents by essentially building each column independently from the other. As a result the current system shows severe limitations in two column mode, such as the fact that spanning floats are always deferred to at least the next page or that numbering between column floats and spanning floats can get out of sequence.

The new algorithm is intended to overcome these limitations and at the same time extend the supported class of document layouts to multiple columns with floats spanning an arbitrary number of columns.

Editor's note

This paper describes facilities offered by the author's new algorithm for use with LaTeX; it therefore seemed appropriate that the paper itself should be typeset using an implementation of it, and the author was enthusiastic in support of the plan.

With some help from members of the LaTeX Team, we have managed to typeset all but the present page of the paper using a version of LaTeX that incorporates a prototype implementation of the new algorithm.

While customising the algorithm to produce the standard layout that readers of *TUGboat* have come to expect, the paper also exhibits the following capabilities of the new algorithm:

- Alignment of text lines throughout the article on an invisible grid.
- Support for spanning bottom floats; examples are on pages 285 and 288.
- Restriction of float placement.

The float placement restrictions selected for this article are as follows: floats have to appear after their call-outs, can only occupy bottom areas, and are not allowed there if footnotes are present in the column. This accounts, for example, for the placement of figure 1, which was moved from the second column of page 280 to the bottom of the first column of page 281.

In the lingua of the algorithm the exact specification used was:

```
float-callout-constraint = after,
float-callout-span-constraint
                           = flexible,
bottom-float-footnote-constraint
                           = forbidden,
max-float-num = 2,
area-list = {b12,b11,b21},
```

These settings are admittedly rather bizarre and were solemnly chosen by the author for illustration purposes.

In order to illustrate clearly the effect of the page layout grid alignment used throughout, on page 279 a grid of lines is superimposed; we hope this does not detract too much from your enjoyment in reading the article.

In general it should be noted that the *TUGboat* layout isn't really suited to be typeset using an underlying grid; headings at the top of the column need to drop to avoid a large gap between the heading and the following text (see page 279) and of course with a flush bottom setting you will get widows and orphans since there is no stretchability on the page.

This title page has been set using the standard (released) LaTeX output routine because the prototype implementation does not at present support switching the number of columns in the middle of the page.

Introduction

One problem with formatting documents containing floats is the number of potential formatting solutions that need to be checked out. The number of trials grows combinatorially in the number of floats and areas which can receive them. If we have n floats waiting to be placed and m areas in which we can place them on the current page being built (not counting the "deferred area") then the number of different placements is given by

$$\#\text{trials} = \binom{n+m}{m} = \frac{(n+m)!}{n!\,m!} \qquad (1)$$

assuming that the order of floats has to be preserved, i.e., if the call-out of float f_i is before the call-out of f_j in the text stream then the float f_i will be placed earlier than float f_j where "earlier" is a defined relation of float areas.

For example, if we have 8 floats waiting to be distributed among 12 areas (which corresponds to a three column page with float areas at the top and bottom allowing for partial spans) then we have to check 125970 possible distributions; if two additional floats appear we end up with 646646 trials.

Even though a large number of these distributions would be unacceptable and discardable straight away, after some initial test, the resulting running time of the algorithm would clearly be beyond any acceptable speed. (Assuming we could do 1000 trials per second, which is ridiculously high since many of them would require trial-typesetting the whole page, then the case of 646646 trials would still take roughly 10 minutes to form a decision.)

Thus it is important to find algorithms with complexity that is at worst linear in both the number of floats on the trial list and the number of possible float areas, even if this means that in a few cases a relatively good layout will not be found. It is even better if they have minimal redundancy.

Note that assessing the actual running time of TEX code is not straightforward since some activities are very much faster than others. For example, performing a test by using a reasonable number of macro expansions and register assignments may be very much slower than running through a long typeset list and then doing a simple test.

The algorithm we have implemented fulfills the requirement of being (essentially) linear in the number of floats and the number of float areas.

The document source model

The document source is a single stream of continuous text containing call-outs to floating objects. (At the moment the call-outs are marked by placing the objects into the stream but it would be possible to provide them as separate objects.) Floating objects (as of today) come in three incarnations:

- Objects where the call-out and the placement requires a strict spatial relationship, e.g., same line in the margin. An example would be marginal notes as implemented by `\marginpar` in LaTeX 2ε.
- Objects where the call-out and the placement are required to fall onto the same column/page/spread, e.g., footnotes.
- Objects where there is a defined relation between call-out and object placement, e.g., "not in an earlier column", or "on the same page or later", etc. These are the traditional floats.

Float objects in the last group are typed where the type is defined by the logical content of the object, e.g., "figure", "table", and so on.

The document formatting is achieved using a minimal but customizable lookahead (typically the considered galley material is the equivalent of one page/spread of textual material ignoring the additional size taken up by embedded float objects).

While making up pages the main "quality" guidance for the algorithm is to try to place each float as early as possible without violating defined constraints.

The document layout model

Page layout grid The algorithm supports the specification of a page grid on which it will align text columns and other elements. This will allow (if suitable parameters for various elements are chosen) to have text lines of different columns all lying on grid points.[1]

Columns The page layouts which are supported by the new algorithm support an arbitrary number of text columns of equal width.

The number of columns per page as well as their

[1] On the current page lines are drawn to highlight the grid. Note that headings, lists, and other "display" objects are not aligned.

width can be changed at forced page breaks such as the start of chapters.

Balanced columns Balancing columns (as done by the `multicol` package) is planned but not implemented. The major problem in that area is the handling of column floats during the balancing process.

Float areas Float objects are distributed into float areas which are rectangular in shape. Float areas span one or more text columns; their horizontal size is therefore given by the following formula (where c is the number of columns spanned):

$$\langle \textit{area-width} \rangle = c \times \big(\langle \textit{col-width} \rangle + \langle \textit{col-sep} \rangle \big) - \langle \textit{col-sep} \rangle$$

The naming conventions for float areas is as follows:

$$\langle \textit{identifier} \rangle \langle \textit{start-column} \rangle \langle \textit{span-count} \rangle .$$

The ⟨*identifier*⟩ is a single letter denoting the type of area, e.g., `t` for top, `b` for bottom. The ⟨*span-count*⟩ is a single digit denoting the number of columns to span. The ⟨*start-column*⟩ is a single digit[2] denoting the start column of the area. Thus `t23` is a top area starting at column two and spanning three columns, i.e., two, three and four. A restriction due to the naming scheme is that currently no more than 9 columns are possible.[3]

Only a subset of the float areas is allowed to be populated on a page. In essence the new algorithm does not support placements that result in "splitting" the text of a column due to a float (other than column "here" floats).[4] This means that population of some float areas must be prevented, namely those satisfying these conditions when *pcs* (where $p = \text{pos}$, $c = \text{column}$, $s = \text{span}$) has just been populated:

$$pij \text{ with } i < c \le i + j < c + s$$

or

$$pij \text{ with } i \le c + s < i + j \le \langle \textit{number-of-columns} \rangle$$

The first formula describes the areas which partly overlap from the left, the second formula describes those that partly overlap from the right. Areas which are sub- or super-areas, e.g., `t13` and `t22`, do not affect each other. The above restriction is necessary to prevent situations like the one shown in figure 1 on the facing page, i.e., where the float area `t32` (represented as b's) would result in splitting the fourth column into two independent text areas.

The possibilities, as well as the restrictions, are equal for both top and bottom areas. This means that the new scheme in particular supports spanning bottom areas.

Float pages and columns Float pages, i.e., pages consisting only of floats, will be supported as well as float columns.

Float types The type of float influences the formatting, e.g., where the caption is placed in relation to the float body, how it is formatted, what kind of fixed strings are added, etc. It also restricts the placement algorithm in respect to which float areas can be populated as explained below.

Margins The marginal areas can receive marginal notes which are aligned with the corresponding text line. In documents with more than two columns marginal notes are currently not supported though one could envision allowing them even there. If marginals have to compete for space the later marginal will be moved downwards if there is enough space on the page, otherwise the line containing the marginal will be moved to the next column/page.[5]

An alternative usage of the margin is to place footnotes into it. A prototype version of this is provided already, see section "Footnotes" on the next page.

Another potential use of the margin areas is to use them (or parts thereof) as float areas in their own right. The problem with this would be that these float areas would have a horizontal width which is different from the column width, thus allowing only a limited class of floats to appear therein.

Another potential extension would be to allow float areas that border on a margin to use the marginal space as part of the float area, thereby allowing the filling of such an area with floats which are wider than the nominal float area. A special case of this, the placement of the caption in the margin beside the float body, is already provided by choosing a suitable caption formatting instance.

[2] With a bit of care in the code this could be extended to allow more than one digit.

[3] The scheme is different from the original one used, where `t23` would have denoted an area starting at column two and spanning until column three.

[4] Perhaps this restriction will be lifted one day.

[5] This is not yet implemented — right now they overprint each other.

Footnotes Footnotes can be regarded as a special type of floats. They are objects which are associated with lines of text (their call-out) but in contrast to normal floats such as "figures" or "tables" their placement constraints are stronger, e.g., they typically have to appear at the bottom of the column which contains their call-outs, or at least they have to appear on the same page as their call-outs.

In its current version, the model supports footnotes beneath the call-out column (normal behavior); all footnotes in the last column (as with the `ftnright` package for two-column mode); all footnotes in the outer (or inner) margin.

Without an extension to the page makeup algorithm (but instead with a suitable redefinition of the footnote commands) they could be processed as marginal notes or alternatively as "end-notes".

Headers and footers The header and footer areas may use data received from individual columns. An extended version of TEX's mark mechanism is made available which allows the definition of arbitrarily many independent classes of marks. Within each mark class information about the top mark (i.e., the mark active at the top of the column), the first mark and the last mark is made available for retrieval.

This allows the production of correct running headers and footers for various types of applications such as dictionaries, manuals, etc.

The processing model

Float placement concepts To build a page or spread the algorithm first assembles enough textual material to be able to fill the page without placing any floats. During this process all floats that have their call-outs within the assembled galley are collected. They form, together with unplaced floats from previous pages, an ordered trial list of floats.

```
aaaaaaaaaaa 444
aaaaaaaaaaa 444
aaaaaaaaaaa 444
111 222
111 222 bbbbbbb
111 222 bbbbbbb
111 222 bbbbbbb
111 222
111 222 333 444
111 222 333 444
111 222 333 444
```

Figure 1: Overlapping float areas

The allowed float areas on the page under construction are totally ordered as well.

The algorithm proceeds by taking the first float from the trial list and trying to place it into the first float area from the area list. It then checks if all constraints (see below) are met and if not the algorithm will try to place the float into the next area until either all constraints are met or the areas in the float area list are exhausted. A trial that does not fail means that this distribution of floats becomes the best solution so far and all further trials will be based on adding to this solution (no backtracking). If the algorithm fails to place the float into any area it means that the float will be deferred to a later page.

As floats are added to areas, the constraints for further trials are changed. There are several reasons for this: on one hand, the call-out positions of various floats move since the float will occupy space on the page; on the other hand, placing a float in some area might result in disallowing the placement of other floats in the same or in other areas.

Float pages and columns At the moment there is only rudimentary support for float pages available: at the start of each page the algorithm will try to form a float page out of all floats that have been deferred from previous pages. However there is no layout control available to define the conditions under which such a trial will succeed.

Float storage Float bodies are typeset into boxes at the point of 'call-out', as with the `figure` and `table` environments in the standard LATEX; it may also be possible to specify at the call-out point a logical pointer to a float whose typesetting is specified elsewhere (e.g., an external file).

However, text sub-elements such as the caption, etc. (e.g., from `\caption`), are not typeset at this stage but are stored as token lists; this allows for trying different possible layout specifications, e.g., for its measure, during the float-positioning trials. At present this is confined to at most a single caption element per float.

Caption processing When a float is placed into an area the caption is trial formatted and mounted onto the float body. This process can take into account various information about the float positioning trial, such as the area to format it into, the fact that it formats onto a verso or recto page, etc. It might try several possibilities before making a decision, e.g., if

one formatting of the float results in violating some constraint(s) it might try a different formatting at this point.

Flushing floats It is possible to mark points in the source document as boundaries beyond which floats whose call-outs are prior to the boundary cannot pass. In other words a "flush point" directs the algorithm to place all affected floats into areas which are "before" the flush point.

If due to other constraints the float could not be placed in such an area the algorithm first retries all potential areas using a less rigid set of constraints (for example, restrictions on the number of allowed floats per area are dropped) and if this still doesn't enable the algorithm to place the float properly it will as a last resort move the flush point to a later column, which means breaking the column text before the flush point.

Flushing of floats can be done either for all floats or on a per float type basis, e.g., it is possible to flush only floats of type "figure".

A flush point can be given an additional attribute which controls the "fuzziness" used by the algorithm. By default the flush point algorithm uses `strict` flushing as described above. The attribute `column` modifies the algorithm's behaviour by enabling a float to move past the flush point as long as it will be placed on the same column. Similarly the attribute values `page` and `spread` will enforce that the float will not be deferred further than the current page or the current spread. This way it can be guaranteed that a float is always visible from its call-out.

Float sequence classes Float sequence classes are collections of float types; each float type belongs to exactly one float sequence class. Within each sequence class the call-out order in the document is always preserved by the float placement algorithm, e.g., if $c_1, c_2, \ldots, c_n$ are the call-outs of all floats of a float sequence class then the corresponding floats will be placed such that f_i will be placed before f_j whenever $i < j$. Thus by putting all float types into a single float sequence class all floats are placed in the order of their call-outs. At the other extreme, if each float type has its own sequence class[6] then floats from one type might move before floats of other types even though the corresponding call-outs are in a different order.

Float and call-out relations The algorithm also keeps track of the relation between an individual float and its call-out. This allows one to define constraints which guide the algorithm during the float placement phase. It is always permissible to place a float "after" its call-out, e.g., in a later column/page. At the moment the following constraints can be specified:

none which means that the relation between call-out and float placement is not relevant for placing floats.

page which means that the float can be placed anywhere on the page with the call-out (it is visible from the call-out).

column which means that the float can be placed before the call-out as long as it is placed in the same column.

after which means that the float has to be placed strictly after the call-out.

When extending the algorithm to directly support spreads the above list is going to be extended by an option that allows floats to move backwards on the whole spread.

Spanning float and call-out relations For floats that span two or more columns there are several possibilities to interpret the spatial relationship between call-out and float areas. For example, if a float, whose call-out is in the second column, has been placed into area `b12`, is this float "before" or "after" its call-out? The answer to this question depends on whether we consider the float being placed into the first or the second column, both of which are valid interpretations.

At the moment the following behaviour can be specified:

strict which means that the leftmost column spanned by the float is regarded as the column in which the float was placed.

flexible which means that the rightmost column spanned by the float is regarded as the column in which the float was placed.

These settings are only relevant if the main float/call-out relations are set to `column` or `after`.

Float and footnote relations It is possible to direct the algorithm to check on each column if there

[6] This is the LaTeX 2_ε default.

are footnotes, and if so to prevent it from placing floats in the bottom area. In theory it might be possible that a forbidden constellation might resolve itself once the algorithm has added further floats, e.g., it could be the case that by adding additional floats the offending footnote gets moved to a different column. However, checking for this would mean potentially large backtracking so the algorithm uses a conservative approach and simply considers a trial as failed if footnotes and bottom areas collide.

It is planned to allow a designer the choice of specifying where the footnotes should be placed in relation to any bottom floats (if the combination is allowed). Right now this is not implemented and column footnotes will always appear below the text column, i.e., above any bottom floats.

Area statuses For each area the algorithm keeps track of whether or not it is closed for individual float types, e.g., is not accepting any more floats of type "figure" or closed for all types. The status of an area can change due to floats being placed into other areas (this might, for example, close earlier areas, or areas that overlap) or it can change due to the fact that the area became too full in some way (e.g., a size constraint or a number of floats constraint).

Some of these constraints can be "relaxed" in certain situations, e.g., if the algorithm is directed to flush out remaining floats prior to a certain point in the galley it will drop constraints related to number of floats per area or size restrictions. However, if an area was closed due to a different float being placed into some other area, this area will stay closed in all circumstances to ensure proper sequential placement of floats and to ensure that overlapping areas that are forbidden as explained in section "Float areas" on page 280 will not receive floats at the same time.

Area constraints The algorithm offers several possibilities for the designer to specify how and under what circumstances a float is allowed to be added to a certain area on the page.

As explained above all areas on a page are tried in a specific order. This order can be specified and changed for specific parts of the document. Areas that are closed for the current type will be bypassed as well as areas which do not span the right number of columns to fit the horizontal size of the float. If these initial tests succeed the float may still fail to be placed into a certain area if it doesn't fulfill the following set of constraints:

- There is an upper limit on the total number of floats that can be placed on an individual page.
- Each area has an upper limit of floats that can go into it.
- After placing the float the remaining space in the text column must be larger than a specified value.

All such constraints are customizable.

Additional constraints will probably be implemented once there has been some experience of what controls are actually needed to allow the specification for a reasonable number of layouts.

For example, LaTeX 2ε allows the designer to restrict the maximum size of an area, but should one provide this or should there be a constraint on the size of all stacked areas? Or should there be both?

To "Here" or not to "Here" LaTeX 2ε allows the user to control the placement of an individual float by specifying one or more areas into which the float would be allowed to move using single letters. As a special notation an `h` would denote a so-called "here" float. Its advertised semantics is to try placing the float "at the position in the text where the environment appears" [1, p. 197]. If this is not feasible LaTeX 2ε would try the remaining allowed possibilities on the next page, thus a float with an `ht` specification would either appear within the text or at the top of the next or a later page.[7]

In many cases people however prefer a "here" which always means "here". The latter form is implemented in some add-on packages for LaTeX 2ε, however usually at the cost of allowing floats to appear out of order.

The new model supports only the absolute "here" form for floats; however, correct ordering of floats in the output is guaranteed (if the tag generating the here float issues flushing of floats for the current type). If there is not enough space to place the float in a column, the float plus the preceding text line[8] is moved to the next column/page.

Grid layout To produce layouts with elements placed on an underlying grid (typically with grid

[7] In two-column mode this can in fact result in a placement on the top of the second column even though the call-out position finally falls into the middle of that column.

[8] More precisely the column is broken at the last breakpoint preceding the current position which is normally one line above but could be more (or less).

points vertically separated by `\baselineskip`) the algorithm assumes that certain parts of the text column, e.g., normal text, will automatically align on the grid as long as the first line is positioned on the grid. A further assumption is that such parts of the column do not contain stretchable amounts of vertical glue so that they are not subject to stretching or shrinking if the material is adjusted to fit a given size.

Given these assumptions, the algorithm proceeds by ensuring that the space taken up by floats (including their separating white spaces) is always of a size such that the remaining space for the text part of the columns allows for an integral number of grid lines. This is achieved by stretching or shrinking the space separating the areas from the text appropriately while building the page as explained in section "Float placement concepts" on page 281.

Within the text column there are typically a number of "display objects" such as headings, equations, quotations, lists, etc., which should not be aligned on the grid. Instead, typically the text before and after is supposed to lie on the grid.[9] This is supported by allowing to mark lines of text (or more generally points in the galley) to "snap to the nearest grid point". One can think of the implementation working by taking the column material up to the marked line and putting it into a vertical box of the size of the nearest possible grid point. By this approach stretchable glue around such a display object will allow the text line that should snap to the grid to move into the correct position. This box is then given back to the page builder to assemble more material for the column. In this way the preceding part of the column becomes rigid; thus a later request for snapping to the grid will only stretch or shrink material further down the column.

A prototype implementation that makes most standard LaTeX objects, like headings, displays, etc., support grid design is available with the package `xo-grid`. It is used for typesetting this document.

User control

Column and page breaks Breaking of columns and pages can be controlled from the source document by placing special tags into it. The `\columnbreak` command ends the current column after the current line (if used in horizontal mode). Similarly the `\pagebreak` command ends the current page.[10]

Manual float flushing The flush float functionality is available within the source document via the command `\flushfloats`. This command takes two optional arguments which, if present, denote the float type to flush (by default all) and the "fuzziness" of the flush (by default `strict`). Other allowed values for the fuzziness are `column`, `page`, or `spread`. If a type is specified for flushing, effectively all types with the same float sequence class are flushed to preserve the ordering.

Specifying preferred areas At the time of writing, the document source interface for specifying the group of areas into which a float is allowed to move is not yet decided. One could envision keeping the original LaTeX interface to float environments with optional argument. In that case something like `[t]` could be internally interpreted as "any top area that exists" and translated into a list such as `t12 t11 t21`. But other interfaces are conceivable as well.

Manually position all floats Any algorithm that automatically places all floats may fail to produce adequate results in some situations. In LaTeX 2_ε the user was offered only the optional arguments of the float environments and by this method and by moving floats slightly in the source document one was finally able to change the formatting as needed.

This was a time-consuming and error-prone manual task and any slight change in the source document text was likely to result in making this work obsolete.

To improve on this situation the new algorithm can be directed to write out a file containing all of its float[11] selections (an example is shown in table 2 on the facing page). By simple drag and drop the user can produce alterations to this selection. If such a modified file is stored as `\jobname.fpc` then the algorithm will use these selections without attempting to apply any of its internal rules. Thus the formatting

[9] In some cases, depending on the design, parts of the structure might be supposed to aligned as well.

[10] At the moment these commands force a break; there is no possibility, as in LaTeX 2_ε, only to suggest that the current point is a good or bad break.

[11] Floats in this context mean "traditional" floats, not footnotes or marginpars.

will happen exactly as specified.[12]

Beside moving floats between float areas it will be possible to move floats in and out of the special area called `hhh` which represents a list of all "here" floats on the page. If a float is moved into the "here" area it means that it will be positioned as a here float at the point of its call-out.

As an extension to this method we are experimenting with restricting the manual control only to parts of the document, e.g., allowing the user to manually fix a single chapter but have the algorithm determine the remainder. We also plan to integrate column length control in this way, so that it becomes easily possible to run a page or double-spread long or short by specifying this externally rather than via tags in the source document.

Tracing the algorithm's behavior In contrast to the LaTeX 2ε output routine, which is a black box as far as the user is concerned, the new algorithm tries hard to make its decision process comprehensible. Table 3 shows a sample output produced by it. It shows for each float which areas have been tried, why they were rejected, etc. There is also an option which produces about 1000 times as much information but the latter is probably useful only for debugging the system in case there are errors in the code.

[12] If the floats are stored within the source document at the point of their call-outs, the algorithm will be able to position a float only if it has already encountered the float in the source document. This means that one can move a float arbitrarily forward but only to a limited extent before its call-out position. If the floats are stored externally to the source document this restriction does not apply.

Manually aligning text in grid layout If the algorithm produces grid layout it automatically aligns certain text lines on the underlying grid. For manual control this functionality is also provided with the command `\TextAlignGrid` which will align the current text line on the grid. By issuing a `\IgnoreAlignToGrid` command grid alignment will be temporarily disabled, while `\ObeyAlignToGrid` will reestablish automatic grid processing.

Layout Specification

In the class file the designer is given control over the algorithm's behavior in all the aspects described above (and several more).

The layout specifications are done through the new template and instance concept, see [2]. Addi-

Table 2: An example `fpl` file

```
Page: 1 (1)
    Area: t13
        Float: 4 (figure 4) []
    Area: b21
        Float: 2 (figure 2) [mylab:fig1]
    Area: t31
        Float: 3 (figure 3) [mylab:fig2]

    Area: hhh
        Float: 11 (table 1) []

Page: 2 (2)
    Area: t13
        Float: 8 (figure 8) []
    Area: t22
        Float: 5 (figure 5) []
    Area: b11
        Float: 6 (figure 6) [mylab:fig3]
    Area: b31
        Float: 7 (figure 7) [mylab:fig4]
```

Table 3: Progress output of the algorithm

```
======================================================================
 STATS: floats waiting = 2 on page 13
======================================================================
Float: \bx@E   {5}  {table} (floats)  {5}   {Statistics from the algorithm}
 area trial: b12  -> failed: span count b12 /= 1
 area trial: b11  -> accepted
Float: \bx@F   {6}  {table} (floats)  {6}   {Running times of the algorithm}
 area trial: b12  -> failed: span count b12 /= 1
 area trial: b11  -> failed: b11 float num reached (1)
 area trial: b21  -> failed: area below flush point (2=2, b21)
 -> failed: --> retry with relaxed conditions

 area trial: b12  -> failed: span count b12 /= 1
 area trial: b11  -> accepted
 STATS: trials = 7
```

tional information such as experimental code, further documentation, etc., can be found on the LaTeX project web site at:

`http://www.latex-project.org`

In contrast to the algorithm itself, which in its basic functionality now seems to be stable and reliable, the design interface is far more experimental. Thus the example declarations given below represent only the current state of thought (or of implementation) and are likely to be modified at any moment.

Float type declarations Float types are declared using the command `\DeclareFloatType` which takes two arguments: the name of the type which is declared and in the second argument a list of key/value pairs which describe the properties of the float type, e.g.,

```
\DeclareFloatType{figure}
  {
   sequence-class-id  = floats,
   toc-extension      = lof,
   caption-text       = \figurename,
   numbered-boolean   = true,
   numbered-id        = figure,
   numbered-within-id = section,
   numbered-action    =
        \thesection.\arabic{figure},
   body-decls         = ,
  }
```

The `sequence-class-id` key defines to which float sequence class the type belongs to. If it is absent a sequence class with the same name as the type is assumed. The sequence class will be automatically initialized if not referenced before.

The `toc-extension` key defines the extension to be used to write the caption to when generating "List of floats" listings. By using the same extension with different types it is possible to generate combined listings, such as "List of tables and figures".

The `caption-text` key defines the fixed text to be used as part of the caption text together with the float number if present, e.g., `Figure`. This information is passed to the caption formatting template so the actual formatting is defined there.

The `numbered-boolean` defines whether or not floats of this type are numbered.

The `numbered-id` key defines the name of the counter to use when numbering floats. If absent a counter with the same name as the type is assumed. By using the same counter with different types it is possible to use a single numbering scheme—in that case the `sequence-class-id` for these types should probably be identical as well to avoid strange numbering sequences within the document.

The `numbered-within-id` key defines the name of the "within" counter, i.e., the counter which if stepped resets the numbering. If the value is empty or not set the float type is numbered in a single sequence throughout the document.

The `numbered-action` key defines the representation of the float number, as used in the caption and by the `\ref`, `\label` mechanism. The default is `\arabic{`⟨*counter*⟩`}`.

The `body-decls` key can hold formatting instructions that should apply to the float body. They can assume a normalized formatting environment already set up by the algorithm.

The declaration of a new float type automatically defines the necessary user document environments.

Float area declarations Any float area that is going to be used at some stage by the algorithm needs to be declared beforehand. This is done through the `\DeclareFloatArea` command which takes two arguments: the name of the area (which has to follow the conventions explained in section "Float areas" on page 280) and a list of key/value pairs describing the characteristics of the area.

```
\DeclareFloatArea{t22}
   {
     class-close-list = {t11,b11},
     all-close-list   = {t12,t32},
     max-float-num    = 2,
   }
```

As of today an area is characterized through the maximum number of floats it is allowed to receive (`max-float-num`) and through two lists which tell the algorithm which other areas are affected by adding a float to the current area. The list `class-close-list` enumerates all areas which are not allowed to receive additional floats of the same sequence class as the float that has been placed into the current area, while the list `all-close-list` contains the information about all areas that are to be completely closed the moment a float is received in the current area.

The `class-close-list` key is primarily intended to specify a partial order on the areas to ensure that floats are not getting out of sequence in the

output. For example, the above declaration says: if a float is placed into area `t22`, i.e., a top area starting at column two and spanning two columns, then the single column areas `t11` and `b11` (i.e., those of the first column) are closed for floats of the same class. However, assuming this example is part of a declaration for a four column layout which could have areas like `t14` or `t13`, there is nothing said about closing those areas. Thus in this particular layout a float spanning three or four columns would still be allowed to go on top.

On the other hand the `all-close-list` key is available to ensure more visual constraints, e.g, "if `t12` gets filled we don't want to have `b12` filled as well, we only want `b22` in this case." In addition it is needed to implement the restriction about overlapping float areas as described in section "Float areas" on page 280, e.g., in the example declaration `t12` and `t32` are closed since they partly overlap with `t22`.[13]

Footnote formatting declarations The formatting of footnotes is specified by declaring instance(s) of type `footnotesetup`. At the moment three templates are available though they should be considered only as prototypes: the template `std` produces conventional footnotes below each column, the template `ftnright` collects all footnotes and typesets them in the rightmost column, and the `margin` template collects and typesets them in the right outer margin.

The keys of the above templates provide only a rudimentary flexibility (to say it positively); in a production version all of them would need a large number of extensions. As an example,

```
\DeclareInstance{footnotesetup}
  {mainmatter}{std}
  {
   text-sep    = 14pt plus 3pt,
   max-height  = 8in,
  }
```

would declare the named instance `mainmatter` that provides footnotes below columns with a separation of `14pt+` and a maximum height for footnotes per column being `8in`.

Instances like this can then be used in the declaration for a particular page layout as explained below.

[13] As mentioned before, this restriction might be lifted in a later version of the algorithm; as long as it is required one could alternatively add those areas behind the scenes to avoid runtime problems.

Alternatively one could use unnamed instances there using the `\UseTemplate` method.

Page setup declarations At the heart of the layout declaration are instances of the type `pagesetup2.`[14]

An example setup showing all currently available keys is given in table 4 on the following page.

Column specification The first four keys (`column-num`, `column-width`, `column-height`, and `column-sep`) describe the column structure of the page layout being defined, i.e., in this case a two-column layout.

Float constraint specification The following four keys define the standard constraints for the algorithm when placing floats: `max-float-num` is the maximum number of floats that can go on a normal page; `float-callout-constraint` defines what kind of relations between float and call-out are allowed (possible values are explained on page 282); `float-callout-span-constraint` handles the interpretation of spanning floats and is explained on page 282; and `bottom-float-footnote-constraint` defines whether or not bottom floats are allowed in case of footnotes.

The last three constraints are replaced by `flush-float-callout-constraint`, `flush-float-callout-span-constraint`, and `flush-bottom-float-footnote-constraint` in case flushing can't be done without relaxing the conditions (`max-float-num` is disregarded in that case automatically).

Float area specification The key `area-list` defines all float areas that are allowed in this page layout as well as defining the order in which the areas are tried when placing floats. The keys `defer-class-close-list` and `defer-all-close-list` define the "closing actions" for the special area which receives the floats that could not be placed. E.g., if a float of a certain class can't be placed then all areas listed in `defer-class-close-list` will be closed for this class of floats. In other words the two keys are comparable to the ones available for area declarations.

Thus these keys together with the keys from the area declarations are most important to guarantee a sensible order of floats on the formatted page.

In an earlier implementation of the algorithm a simpler scheme was used: there was a single area list which was shortened whenever a float couldn't be

[14] The number 2 has historical reasons and will vanish at some point in the future.

placed into it thereby confining the remaining floats to this restricted selection. This works fine as long as there are mainly single column floats since in this case the area can be reasonably ordered into a single sequence. However the moment spanning floats are supported the situation gets less straightforward. Is it allowed to place a later float into `t12` if there is already a float in the area `t11`?

It is quite likely that the current controls will turn out to be too crude. This will be seen once a suitable number of layouts have been produced under this scheme (or couldn't be produced because they turned out to be unspecifiable).

There needs to be space between floats in an area and areas need to be separated from each other, as well as from the column text. For this we have the following keys: `float-float-sep` is the separation between two floats in an area, `float-area-sep` is the separation between two vertically adjacent areas, and `float-text-sep` finally is the separation between a float area and the column text.[15] The separation between inline floats and surrounding text is given by `float-inline-sep`.

Grid specification To produce a grid based design the `grid-point-sep` needs to be given a positive dimension. This defines the distance between grid points on which the algorithm aligns column text, inline floats, etc.[16]

To align column text at a grid point the algorithm will extend the `float-text-sep` space. Alternatively, if the nearest grid point can be reached by shrinking that space (assuming its specification contains a `minus` component) the algorithm will use that grid point instead. In a similar fashion the space around an inline float will be determined by the value of `float-inline-sep`.

Footnote, etc., specification Finally the key `footnote-setup` receives an instance of a `footnotesetup` template, thereby defining how footnotes are handled and presented.

What is clearly missing here is handling of other page elements such as running headers and footers,

[15] A possible extension would be to allow ornamental material in place of white space.

[16] Setting this parameter is not sufficient: to make grid setting possible several other parameters need to be set to suitable values as well, e.g., the distance between baselines should be compatible and the column height needs to be a multiple of this value.

Table 4: Example declaration for the `pagesetup2` template showing all currently available keys

```
  \DeclareInstance{pagesetup2}{mainmatter}{std}
    {
% column specification
      column-num                              = 2,
      column-width                            = 220pt,
      column-height                           = 610pt,
      column-sep                              = 20pt,
% float constraint specification
      max-float-num                           = 3,
      float-callout-constraint                = after,
      float-callout-span-constraint           = strict,
      bottom-float-footnote-constraint        = forbidden,
      flush-float-callout-constraint          = page,
      flush-float-callout-span-constraint     = flexible,
      flush-bottom-float-footnote-constraint  = none,
% area specification
      area-list                               = {t12,t11,b11,b12,t21,b21},
      defer-class-close-list                  = {t12,t11,b11,b12,t21,b21},
      defer-all-close-list                    = ,
      float-float-sep                         = 15pt,
      float-text-sep                          = 30pt minus 8pt,
      float-area-sep                          = 15pt,
      float-inline-sep                        = 6pt minus 2pt,
% grid specification
      grid-point-sep                          = 12pt,
% footnote etc specification
      footnote-setup                          = mainmatter,
    }
```

the folio, etc. This will be added soon.

Float formatting declarations For the attachment of captions to floats there exists a prototype interface using templates of the type `buildfloat`. At the time of writing, available templates are `centeredbelow`, `centeredabove`, and `bottomright`, which center the caption below or above the float body or place it to the right of it, aligned with the bottom of the float body. All of them would need to be generalized for a production system to become more flexible.

When trial-formatting a float the algorithm checks for the existence of a number of `buildfloat` instances and uses the first one that exists to build the float. More precisely it first checks if an instance with the name ⟨*area*⟩-⟨*type*⟩ exists, then it looks for ⟨*area*⟩, then for ⟨*type*⟩, and finally, if none of them exists, for an instance with the name `default`. So at least the latter instance has to be declared by the class.

```
\DeclareInstance{buildfloat}{default}
   {centeredbelow}{}
\DeclareInstance{buildfloat}{table}
   {centeredabove}{}
\DeclareInstance{buildfloat}{t31}
   {bottomright}{}
\DeclareInstance{buildfloat}{t22}
   {bottomright}{}
```

The example declaration above defines the placement of captions above tables and below for all other types, with the exception of the areas `t31` and `t22` where the captions are set to the side.

Performance of the algorithm

To test the performance of the algorithm we prepared a somewhat ridiculous test file containing three types of floats ("figures", "tables", and "algorithms") with a total number of 47 floats. The chosen layout had 3 columns and 11 potential float areas. Figure captions have been placed below the float while with tables and algorithms the caption was placed on top. The exception was the top areas adjacent to the outer margin: floats placed there got their captions placed to the right and partly into the margin. Footnotes were collected for all columns and placed in the outer margin.

Floats had to strictly follow their call-out and a maximum of ten floats was allowed per page, i.e., roughly three per column.

Since the document contained many floats early on (24 on page one) and the first of these was especially constructed to be not placeable the first time around, the algorithm had to work hard to place all the dangling floats. Table 5 shows some statistics as produced by the algorithm on the number of trials necessary (the highest number was 397 for 37 floats; by comparison, equation (1) on page 279 would give 22595200368 which would probably take a bit longer to evaluate). Note that on the third page the algorithm was able to produce a float page; on all other pages the float page trial was unsuccessful.

Table 6 on the following page shows the running times needed to produce the final document of 13 pages when the algorithm is used with different tracing settings. The test machines were a Pentium III 650 machine and an older laptop with a 486 processor. In both cases TEX was run straight from a TEX Live 4 CD.

These times show that the algorithm has an acceptable time performance since even on a 486 the average time to produce a page is roughly 2 seconds.

Outlook

While the current algorithm performs well there are

Table 5: Statistics from the algorithm

```
STATS: floats waiting = 24 on page 1
STATS: trials = 286
STATS: floats waiting = 19 on page 2 (float page)
STATS: trials = 159
STATS: floats waiting = 37 on page 2
STATS: trials = 397
STATS: floats waiting = 19 on page 3 (float page)
STATS: trials = 166
STATS: floats waiting = 7 on page 4 (float page)
STATS: trials = 41
STATS: floats waiting = 20 on page 4
STATS: trials = 204
STATS: floats waiting = 5 on page 5 (float page)
STATS: trials = 27
STATS: floats waiting = 12 on page 5
STATS: trials = 108
STATS: floats waiting = 0 on page 6 (float page)
STATS: trials = 0
STATS: floats waiting = 6 on page 6
STATS: trials = 57
         ...
STATS: floats waiting = 6 on page 12 (float page)
STATS: trials = 26
STATS: floats waiting = 6 on page 12
STATS: trials = 37
STATS: floats waiting = 0 on page 13
STATS: trials = 0
```

several areas in which its functionality could and probably should be extended. The most important points are given in the following list.

- Balancing of partial pages, comparable to the way the `multicol` package works, should be implemented to allow for layouts where, for example, a heading should span across all columns.
- We intend to provide more control over the marginal areas, allowing for marginal floats as well as other objects in the margin, properly interacting with each other.
- Without much effort the algorithm could be extended to properly support double-spreads so this should be added some time soon.
- Once the algorithm has decided which floats to place onto a page one could add a post-processing step in which the placement could be reconsidered according to different rules. For example, if the call-out relation is `page` then floats will tend to be placed in the left-hand columns. This is fine as long as there are many floats to process but on a page with only a few floats one might want to redistribute them differently once it is clear which floats could go onto the page.
- Since it is known beforehand how many floats are actively waiting to be placed, one could use a different algorithm that tries all possible combinations as long as there are only a limited number of floats to be placed. The boundary at which the algorithm changes behavior could be made customizable so that people with faster machines (or more patience) could have the search for optimum running for as many floats as they like.

Table 6: Running times of the algorithm

	PIII (650MHz)	486DX4 (75MHz)
	no tracing	
real	0m1.533s	0m27.633s
user	0m1.460s	0m26.940s
sys	0m0.050s	0m0.690s
	progress information	
real	0m3.116s	0m36.885s
user	0m1.740s	0m34.470s
sys	0m0.080s	0m2.420s
	full tracing	
real	0m7.833s	1m22.480s
user	0m2.720s	1m7.890s
sys	0m0.280s	0m12.360s

References

[1] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[2] Frank Mittelbach, David Carlisle, and Chris Rowley. New interfaces for LaTeX class design. *TUGboat*, 20(3):214–216, September 1999.

Frank Mittelbach

Abstract: The Penrose notation: a LaTeX challenge

Timothy Murphy
tim@maths.tcd.ie

Abstract

Over 30 years ago, Roger Penrose — Oxford mathematician and AI scourge — invented a notation for *tensors*, which has become a kind of secret language among a coterie of aficionados.

This notation lies somewhere between the classic index notation of relativists and the functorial notation of multilinear algebraists. By general consensus, Penrose' notation avoids the complexity of the first and the chilling abstraction of the second, providing a concrete model for tensor algebra and calculus of great pedagogical value.

The aim of this talk is to describe the Penrose notation — and it should be emphasized at the outset that there are absolutely no mathematical 'prerequisites' to understanding this description — and to present a LaTeX package for incorporating the notation into mathematical documents.

This package is far from perfect; indeed it is its very imperfection which encourages the author to place it before this audience, in the hope (and trust) that its collective TeXpertise will suggest improvements in the package, or even an entirely new approach.

Many mathematicians down the centuries have developed their own private languages, necessarily restricted to their notebooks by the exigencies of metal type. It is interesting to speculate on the influence of the press in ironing out idiosyncrasies of thought, in the same way that it ironed out variations in grammar and spelling.

But the digital press opens up a new possibility: these once secret languages may enter the universal realm of mathematical discourse. The one-dimensional age ushered in by Gutenberg may be at an end.

Notations like this are much more than diagrams. They hold the meaning of the document, and enter into the mathematical syntax. In our case, the first step is to express the Penrose notation in BNF form, or its fashionable equivalent, XML,

Timothy Murphy

A Perl port of the mathsPIC graphics package

Dedicated to the fond memory of Mikhail Syropoulos, my beloved brother.
— Apostolos Syropoulos

Apostolos Syropoulos
Department of Civil Engineering
Democritus University of Thrace
Xanthi, Greece
apostolo@obelix.ee.duth.gr

Richard W. D. Nickalls
Department of Anaesthesia
City Hospital, Nottingham, UK
dicknickalls@compuserve.com

Abstract

This article describes the authors' experience of porting the mathsPIC graphics package to Perl. The motivation for using Perl is described, as well as the reasons for developing it using the Noweb literate programming system. Finally, a simple example is presented.

Introduction

This article is a short *work in progress* report of our porting of the graphics package mathsPIC (`CTAN/graphics/pictex/mathspic/`) to standard Perl; a project designed to make it available for a wide range of platforms.

MathsPIC (Nickalls, 1999) is a filter program for use with the excellent PiCTeX drawing engine[1]. MathsPIC differs from other graphics packages in that it provides an environment for manipulating named points, which greatly facilitates the drawing of geometrical figures. It also accommodates relative addressing, scalar variables, and file input for data points and other commands. MathsPIC was originally written in PowerBASIC 3.5, a commercial version of BASIC available only for MS-DOS systems.

However the original MS-DOS version of mathsPIC does have certain limitations; for example, it is unable to parse mathematical functions. Furthermore, maths libraries for PowerBASIC are all commercial. Consequently, in order to significantly extend mathsPIC, the authors felt it was necessary to reimplement it in a systems programming language consistent with the philosophy 'write once, run everywhere', namely Perl.

[1] The original PiCTeX files have been significantly improved and made memory-efficient by Andreas Schrell—see `pictexwd.sty` (`CTAN/graphics/pictex/addon/`)

This reimplementation of mathsPIC started in January 2000, and has been conducted as a collaborative project over the internet, by the authors. The authors used different platforms during the development (Perl 5.6 on a Solaris x86 machine, and djgpp Perl 5.005 on an MS-DOS machine).

Probably the most difficult aspects were maintaining a consistent syntax, and downwards compatibility. We decided to maintain case-insensitivity for command names, as this was found to be particularly useful in promoting readability. The log file (`.mlg` file) was structured to mirror the usual TeX and LaTeX log files in order to allow tools which process these files to work similarly with mathsPIC. This was also an appropriate time to revise and improve the syntax of the language. So, for example, the original `variable` command format, e.g., `variable(x){b,advance(4)}`, has been improved to allow an algebraic syntax, e.g., `var x=b+4`, as well as allowing several variables to be defined using one command, e.g., `var y=3*(j-2), j27=r/3, p4=AB`.

Why Perl?

Perl is a high-level scripting programming language with an eclectic heritage designed by Larry Wall. It is an interpreted language which has been ported to most operating systems, and so is a particularly good choice for the implementation of programs

In this section we define a few global variables. More specifically: variable `$version_number` contains the current version number of the program, variable `$commandLineArgs` contains the command line arguments. These two variables are used in the `print_headers` subroutine. Variable `$command` will contain the whole current input line . Hash `%PointTable` is used to store point names and related information. Hash `%VarTable` is used to store mathsPIC variable names and related information. Variable `$no_errors` is incremented whenever the program encounters an error in the input file. Variables `$xunits`, `$yunits` and `$units` are related to the `paper` command. In particular, variable `$units` is used to parse the unit part of the `unit` part of the `paper` command. Variable `$defaultsymbol` is used to set the point shape. Variable `$PI` holds the value of the pi constant.

```
<Define global variables>=

$version_number = "0.0 September 1, 2000";
$commandLineArgs = join(" ",@ARGV);
$command  = "";
$curr_in_file = "";
keys(%PointTable) = 0;
keys(%VarTable) = 0;
$no_errors = 0;
$xunits = "1pt";
$yunits = "1pt";
$units = "pt|pc|in|bp|cm|mm|dd|cc|sp";
$defaultsymbol = '$\bullet$';
$PI = atan2(1,1)*4;
```

Used above.

Figure 1: Literate (woven) extract from mathsPIC source

likely to be used on many different operating systems. Its design has been largely influenced by the C programming language, but also by other tools and languages, for example, SED, AWK, and the Unix shell. Perl's process, file, and text manipulation facilities make it particularly well-suited for tasks involving quick prototyping, system utilities, software tools, system management tasks, database access, graphical programming, networking, and world wide web programming. Finally, and importantly, Perl is in the public domain, and so there are no commercial restrictions.

Literate programming

The Perl mathsPIC program was developed throughout using so-called 'literate program' methodology (Knuth, 1992; Syropoulos, 1999; Hatzigeorgiu and Syropoulos, 1998), in conjunction with Norman Ramsey's freely available Noweb literate programming tool (Ramsey, 1994).

Noweb was chosen partly because it is language-independent, but also because it can generate the weaved file in variety of formats, e.g., plain TEX, LATEX, HTML and nroff. Noweb allows programs to be built up of named chunks in any order with documentation interleaved, and has powerful indexing and cross-referencing facilities. Furthermore, Noweb's pipeline makes it easy to extend, and different stages of the pipeline can can be implemented in different programming languages (Ramsey, 1994). Noweb uses its notangle and noweave tools to extract code and documentation as required. Figure 1 shows a formatted code chunk of the resulting literate program.

Example

By way of example we show the mathsPIC script file (`fig2.m`) which produced Figure 2, and also the associated output LaTeX file (`fig2.mt`) generated by mathsPIC. Note the use of `pictexwd.sty` (`CTAN/graphics/pictex/addon/`).

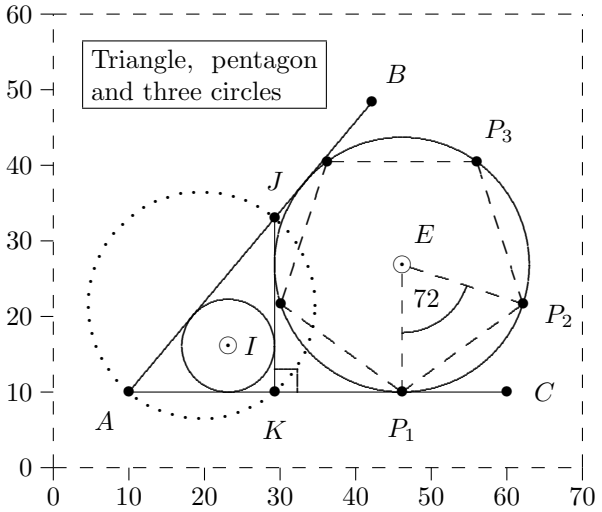


Figure 2: Example diagram (mathsPIC source in text)

```
%% mathsPIC script file (fig2.m)
%% Figure 2
\documentclass[a4paper]{article}
\usepackage{pictexwd}
\begin{document}
\beginpicture
\setdashes
paper{units(mm),xrange(0,70),yrange(0,60),
                axes(LBT*R*),ticks(10,10)}
\setsolid
point(A){10,10}   %% anchor point
point(B){A, polar(50,50 deg)}
point(C){A, polar(50,0 deg)}
point(J){pointonline(AB,30)}
point(K){perpendicular(J,AC)}
drawRightangle(JKC,3)
drawLines(AB,AC,JK)
drawIncircle(AJK)
drawExcircle(AJK,JK)
\setplotsymbol({\large .})
\setdots
drawCircumcircle(AJK)
point(I){IncircleCenter(AJK)}[$\odot$]
point(E){ExcircleCenter(AJK,JK)}[$\odot$,1.2]
point(P1){perpendicular(E,AC)}
var r = EP1     %% radius of excircle
var d = 360/5   %% angle for pentagon
var a1=-90,a2=a1+d,a3=a2+d,a4=a3+d,a5=a4+d
point(P2){E, polar(r,a2)}
point(P3){E, polar(r,a3)}
point(P4){E, polar(r,a4)}
point(P5){E, polar(r,a5)}
drawPoints(ABCJKIEP1P2P3P4P5)
\setplotsymbol({\tiny .})
\setdashes
drawLine(P1P2P3P4P5P1,EP1,EP2)
\setsolid
drawAnglearc{angle(P2EP1),radius(9),
                    internal,clockwise}
\newcommand{\figtitle}{%
\   \begin{minipage}{30mm}%
\   Triangle, pentagon and three circles%
\   \end{minipage}%
\   }%
text(\fbox{\figtitle}){20,52}
variable(s){5}
text($A$){A, polar(s,230 deg)}
text($B$){B, polar(s,50 deg)}
text($C$){C, polar(s,0 deg)}
text($J$){J, polar(s,90 deg)}
text($K$){K, polar(s,270 deg)}
text($E$){E, polar(s,0 deg)}
text($72$){E, polar(5.5,-54 deg)}
text($I$){I, shift(3, 0)}
text($P_1$){P1, polar(s,a1)}
text($P_2$){P2, polar(s,a2)}
text($P_3$){P3, polar(s,a3)}
\endpicture
\end{document}
```

The axes and bounding box used while constructing the figure are shown here in order to make it easier to understand the mathsPIC script file and the output LaTeX file (`fig2.mt`). They are easily removed simply by commenting out the `axes` and `ticks` options from the mathsPIC `paper` command.

Note that in addition to mathsPIC commands (*not* prefixed with a backslash), the script file can also contain PiCTeX, TeX and LaTeX commands, since mathsPIC processes the input file according to the following rules.

- mathsPIC commands are converted into their equivalent PiCTeX commands. The mathsPIC commands are also copied verbatim but commented out—this makes the output file easier to understand.

- Lines having a leading backslash followed by *one or more spaces* (e.g., \␣) are copied verbatim except for the leading backslash.

- Lines having a leading backslash followed by *a non-space character* (e.g., `\setdashes`) are copied verbatim.

Note that the output LaTeX file (`fig2.mt`) also contains some additional information (e.g., coordinates of derived points, radius of circles etc.) which is usually included at the end of the original mathsPIC command, but sometimes as a separate line. Once the figure is finished, the output file can be generated *without* any comment lines simply by using the `-c` command-line switch during the final mathsPIC run.

References

Hatzigeorgiu, N and A. Syropoulos. "Literate Programming and the 'Spaniel' Method". *SIGPLAN Notices* **33**(12), 52–56, 1998.

Knuth, D E. *Literate programming*. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information, 1992. Co-published by Cambridge University Press.

Nickalls, R W D. "MathsPIC: a filter program for use with PiCTeX". In *Proceedings of EuroTeX'99*, pages 192–210, Heidelberg, Germany. 1999. A slightly updated version of this paper was published (in English) in the Greek TeX Friends' journal *Eutopon*, No. 3 (October 1999), pp. 33–49.

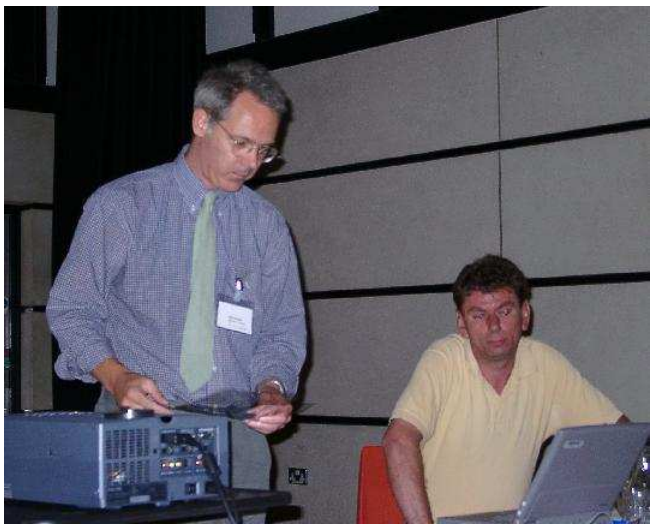
Ramsey, N. "Literate programming simplified". *IEEE Software* **11**(5), 97–105, 1994.

Syropoulos, A. "Literate programming: the other side of the coin". *RAM Magazine* (129), 248–253, 1999. In Greek.


```
%% mathsPIC output file (fig2.mt)
%% Figure 2
\documentclass[a4paper]{article}
\usepackage{pictexwd}
\begin{document}
\beginpicture
\setdashes
%% paper{units(mm),xrange(0,70),yrange(0,60),axes(LBT*R*),ticks(10,10)}
\setcoordinatesystem units < 1mm, 1mm>
\setplotarea x from  0 to  70, y from  0 to  60
\axis left ticks numbered from  0 to  60 by  10  /
\axis right /
\axis top /
\axis bottom ticks numbered from  0 to  70 by  10  /
\setsolid
%% point(A){10,10}  ( 10 , 10 )  %% anchor point
%% point(B){A,polar(50,50deg)}  ( 42.13938 , 48.30222 )
%% point(C){A,polar(50,0deg)}  ( 60 , 10 )
%% point(J){pointonline(AB,30)}  ( 29.28363 , 32.98133 )
%% point(K){perpendicular(J,AC)}  ( 29.28363 , 10 )
%% drawRightangle(JKC,3)
\plot   32.28363  10     32.28363  13   /
\plot   29.28363  13     32.28363  13   /
%% drawLines(AB,AC,JK)
\plot   10  10     42.13938  48.30222 /  %% AB
\putrule from   10  10 to  60  10  %% AC
\putrule from   29.28363  32.98133 to  29.28363  10  %% JK
%% drawIncircle(AJK)
%% Incircle centre =  23.15115 , 16.13248 ;  Radius =  6.132483
\circulararc 360 degrees from  29.28363  16.13248 center at  23.15115  16.13248
%% drawExcircle(AJK,JK)
%% Excircle centre =  46.13249 , 26.84886 ;  Radius =  16.84886
\circulararc 360 degrees from  62.98135  26.84886 center at  46.13249  26.84886
\setplotsymbol({\large .})
\setdots
%% drawCircumcircle(AJK)
```

```
%% circumcircle centre =  19.64182 , 21.49067 ;  Radius =  15
\circulararc 360 degrees from  34.64181  21.49067 center at  19.64182  21.49067
%% point(I){IncircleCenter(AJK)}[$\odot$]  ( 23.15115 , 16.13248 )
%% point(E){ExcircleCenter(AJK,JK)}[$\odot$,1.2]  ( 46.13249 , 26.84886 )
%% point(P1){perpendicular(E,AC)}  ( 46.13249 , 10 )
%% var r = EP1  ( 16.84886 )  %% radius of excircle
%% var d = 360/5  ( 72 )  %% angle of pentagon (deg)
%% var a1=-90, a2=a1+d, a3=a2+d, a4=a3+d, a5=a4+d
%% a1 = -90
%% a2 = -18
%% a3 =  54
%% a4 =  126
%% a5 =  198
%% point(P2){E,polar(r,a2)}  ( 62.1567 , 21.64227 )
%% point(P3){E,polar(r,a3)}  ( 56.036 , 40.47987 )
%% point(P4){E,polar(r,a4)}  ( 36.22898 , 40.47987 )
%% point(P5){E,polar(r,a5)}  ( 30.10827 , 21.64227 )
%% drawPoints(ABCJKIEP1P2P3P4P5)
\put {$\bullet$} at  10  10  %% A
\put {$\bullet$} at  42.13938  48.30222  %% B
\put {$\bullet$} at  60  10  %% C
\put {$\bullet$} at  29.28363  32.98133  %% J
\put {$\bullet$} at  29.28363  10  %% K
\put {$\odot$} at  23.15115  16.13248  %% I
\put {$\odot$} at  46.13249  26.84886  %% E
\put {$\bullet$} at  46.13249  10  %% P1
\put {$\bullet$} at  62.1567  21.64227  %% P2
\put {$\bullet$} at  56.036  40.47987  %% P3
\put {$\bullet$} at  36.22898  40.47987  %% P4
\put {$\bullet$} at  30.10827  21.64227  %% P5
\setplotsymbol({\tiny .})
\setdashes
%% drawline(P1P2P3P4P5P1,EP1,EP2)
\plot   46.13249  10     62.1567  21.64227 /  %% P1P2
\plot   62.1567  21.64227     56.036  40.47987 /  %% P2P3
\putrule from   56.036  40.47987 to  36.22898  40.47987  %% P3P4
\plot   36.22898  40.47987     30.10827  21.64227 /  %% P4P5
\plot   30.10827  21.64227     46.13249  10 /  %% P5P1
\putrule from   46.13249  25.64886 to  46.13249  10  %% EP1
\plot   47.27376  26.47803     62.1567  21.64227 /  %% EP2
\setsolid
%% drawAnglearc{angle(P2EP1),radius(9),internal,clockwise}
\circulararc -72 degrees from  54.692  24.0677 center at  46.13249  26.84886
\newcommand{\figtitle}{%
    \begin{minipage}{30mm}%
    Triangle, pentagon and three circles%
    \end{minipage}%
    }%
%% text(\fbox{\figtitle}){20,52}
\put {\fbox{\figtitle}} at  20  52
%% variable(s){5}  ( 5 )
%% text($A$){A,polar(s,230deg)}
\put {$A$} at  6.786062  6.169777
%% text($B$){B,polar(s,50deg)}
```

```
\put {$B$} at  45.35332  52.13245
%% text($C$){C,polar(s,0deg)}
\put {$C$} at  65  10
%% text($J$){J,polar(s,90deg)}
\put {$J$} at  29.28363  37.98133
%% text($K$){K,polar(s,270deg)}
\put {$K$} at  29.28363  5
%% text($E$){E,polar(s,0deg)}
\put {$E$} at  51.13249  26.84886
%% text($72$){E,polar(5.5,-54deg)}
\put {$72$} at  49.36531  22.39926
%% text($I$){I,shift(3,0)}
\put {$I$} at  26.15115  16.13248
%% text($P_1$){P1,polar(s,a1)}
\put {$P_1$} at  46.13249  5
%% text($P_2$){P2,polar(s,a2)}
\put {$P_2$} at  66.91198  20.09719
%% text($P_3$){P3,polar(s,a3)}
\put {$P_3$} at  58.97492  44.52495
\endpicture
\end{document}
```



Richard W.D. Nickalls and Apostolos Syropoulos

Chess macros for chess games and puzzles

Marina Yu. Nikulina
St. Petersburg Technical University
St. Petersburg, Russia
marishka@ptslab.ioffe.rssi.ru

Alexander S. Berdnikov
Institute of Analytical Instrumentation
St. Petersburg, Russia
berd@ianin.spb.su

Abstract

The macro package UCHESS.STY described here generalizes the well known chess macros CHESS.STY by Piet Tutelaers [1], BDFCHESS.STY by Frank Hassel [2] and the less known CHESS.STY by Tomasz Przechlewski [9]. It adapts international (European) chess notation for other languages (primarily for Russian) as it is required by modern LaTeX. It also allows the User to annotate non-classical chess games — hexagonal chesses, Ω-chess, etc. Special care is taken to allow the User to represent various chess puzzles with non-standard chess rules.

Introduction

The well known and available on CTAN chess macros CHESS.STY (by Piet Tutelaers) and BDFCHESS.STY (by Frank Hassel) are excellent but do not work well with the Russian language without manual correction of the original macros. Moreover, if the BABEL package (versions 3.6 and 3.7) is used, neither CHESS.STY nor BDFCHESS.STY can recognize the presence of BABEL and hence cannot support multi-language chess notation as they should. The other reason to issue a new macro set is that CHESS.STY and BDFCHESS.STY only deal with the classical chess game — such things as non-standard figures and non-standard chess boards are not supported.

The new macro package UCHESS.STY outlined here should correct these gaps. It follows the scheme suggested in CHESS.STY and BDFCHESS.STY and is more an extension of the ideas suggested in these packages than an original macro package. It includes support of multilingual chess notation in a more robust way (including support of the Russian notation). It supports multiple chess boards and positions simultaneously and contains special macros which simplify the presentation of chess problems. It also has special commands which help to represent non-standard chess puzzles with unusual boards and rules. It is based on an extended chess font which permits representation of chess and checkers on non-standard boards and with non-standard figures (like the hexagonal chess [10] and Ω-chess [7] games), and will draw the arrows showing moves, etc.

Problems with foreign language support

One of the most powerful features of CHESS.STY by Piet Tutelaers is the support of chess notation for foreign languages. But since that part of LaTeX changed greatly since CHESS.STY was released, the feature has steadily transformed itself to a weak point of the package. Let us consider this effect in more detail.

First, the support of foreign languages in CHESS.STY is entirely based on BABEL. But it recognizes the presence of BABEL by the existence of the macro `\babel@core@loaded` — which is obsolete since BABEL 3.6a (1996/11/02), as explained in the BABEL manual. As a result CHESS.STY loads `english.sty` even if BABEL is already loaded. (And if by chance it reads the obsolete version of `english.sty` included among other files with the bundle with CHESS.STY, the result may be rather strange.)

Second, the CHESS.STY support of foreign languages analyzes the current BABEL language each time you switch to chess mode and unconditionally redefines the chess notation as specified by this language. So it is impossible without extensive language switching to have chess notation in English while the main text is in German, French, Dutch or Russian. Moreover, if the corresponding language is

not supported by CHESS.STY at all (the current version 1.2 recognizes only `english`, `french`, `german` and `dutch` as valid languages), you'll see that your chess notation disappears without any warning or error message — so you must switch language to some known one before each chess fragment. (This defect is partially corrected in BDFCHESS.STY — namely, you can specify one language for chess notation and another language for the main text. But if the language selected for chess notation is not supported by CHESS.STY, you'll still find all your chess figures dissappearing from your document without any warning.)

You can correct this defect by adding the support of the desired language in CHESS.STY directly. But if you would like to do this, you'll need to correct the original file or to redefine its internal macro `\select@pieces` — both operations are not too difficult for TEX experts (especially taking into account the that the procedure is described in the documentation supplementary to CHESS.STY). But for the ordinary user it is not so clear how to do it, and there is no ready-to-use intrinsic mechanism for adding a new language in CHESS.STY with modest effort and in a standard way. (And the strange behaviour of CHESS.STY is quite annoying for an ordinary user because he/she cannot understand what happens and what is wrong with the document.)

Finally, CHESS.STY supports foreign language notation in the assumption that in *any* language one-letter notation is used. This is true for English, French, German, Dutch, Italian, etc., but in Russian it is not so: *two-letter* notation is used to distinguish *King* ('Kp') from *kNight* ('K'). Taking into account that for the Russian language there is no unified input encoding (different platforms uses different encodings for Russian letters) the problem how to support Russian chess notation in a unified way becomes even more difficult.

Foreign language support in UCHESS.STY

The support of foreign language notation is the main component which is different in UCHESS.STY and CHESS.STY (to be honest, correct support of Russian chess notation was the main reason to substitute CHESS.STY by UCHESS.STY). You may control the language used for chess notation independently of the language used for the main text by the command `\chesslanguage`. And although your chess language is synchronized with the main text if you specify the chess language as `babel`, synchronization is not obligatory.

The procedure for adding new languages into UCHESS.STY is modified as well. First, when a special language is required for chess notation, the package checks that a characteristic macro command based on the language name already exists. If the command does not exist, a warning message is issued and the chess notation is switched to English (the default notation built-in for UCHESS.STY). Second, if you need to add a new language for UCHESS.STY, you just define the necessary macro command — this is enough to extend the set of languages supported by UCHESS.STY — you do not need to change anything thing inside the original UCHESS.STY commands. Finally, when you load the package UCHESS.STY by the LATEX 2_ε command `\usepackage`, you can define the list of supported languages as its options — for each unknown option *name* the file *name*.cld is loaded (.cld stands for *chess language definition*). (UCHESS.STY also checks that the macro with the desired name is really defined in this file. For all main languages the corresponding .cld-files are included with the package UCHESS.STY.)

But the problem with the Russian language and its numerous encodings still remains. To solve it we added an additional (optional) parameter which helps the language switching macro to select the appropriate input encoding. Additionally, to simplify the user's work on defining the language switching macro there are standard commands which help to define one-letter and two-letter abbreviations for chess figures. And if by chance the language switching mechanism misbehaves (as it does for CHESS.STY where the chess figures just disappear from your text if there are no standard chess abbreviations for your language), the user can always use macro commands with fixed names.

The UCHESS.STY support of other languages also allows the user to switch flexibly between alphanumeric notation and the 'figurine' notation in the document without major changes to the input file (where the alphanumeric notation is in agreement with your language). Special precautions are performed to adapt other special symbols used by chess notation properly when the language is switched as well (for example, the symbols used for international *Chess Informant* notation and the notation traditionally used in Russia, Poland, etc., differ widely as concerns the symbols used for check, checkmate, stalemate, etc.).

Non-standard chess games

Chess board for 4 players and its rules It is nice that there is the support of the classical chess game in LATEX, but today the non-standard chess games are becoming more and more popular [8, 7].

While it is necessary to keep chess figures of *four* colors for the chess game for 4 players [8], it is possible to overcome the restriction of typical black-and-white typography by rotating the corresponding chess figures by 90°, 180° and 270°. The fonts described in [3] enable us to generate the rotated fonts without any problems but it is questionable whether that it is necessary to support such an exotic chess game inside UCHESS.STY. Maybe it will be done someday — if somebody really needs it.

Ω-chess board and its rules While the chess game for 4 players [8] seems to be just a private joke for home entertainment, Ω-chess [7] is a more serious game, and its support by LaTeX is a desirable feature. Support of the Ω-chess board requires two additional figures (*wizard* and *champion*), a non-standard chess-board with 104 fields (10×10 plus four separate corner squares) and support of non-usual moves and field notation in the way that is already done for the classical chess game in CHESS.STY. Additional figures are included in our new chess font [3], and the extended chess notation is supported when you mark your chess board as an Ω-game by a special option.

Hexagonal chess boards and their rules In addition to the Ω-chess board there are (at least) two hexagonal chess boards [10] with chess rules of their own. These boards are composed from hexagonal fields colored in three colors (white, gray and black), and although *standard* chess figures are used, the initial positions and the rules how to move the figures over the board are far from being standard. As for Ω-chess the support of hexagonal chess games is included in UCHESS.STY although serious modifications of the internal representations used for chess boards in CHESS.STY were necessary.

Chess problems The classical chess game is divided into two relatively independent branches. There is the chess game itself where two people fight for victory, and there is the solution of specially prepared chess problems. (Actually there is a third branch as well — the computer chess game and computer chess analysis — but these are not too different from the other two branches as concerns chess typesetting.)

While the support of an ordinary chess game is done in CHESS.STY in a user-friendly manner, and while BDFCHESS.STY supports playing chess games by post in the same way, the support of chess typesetting for chess problems is not so flexible. To assist the user working in this specific field UCHESS.STY contains the support of several boards and positions simultaneously, storing and copying the board content, transformation of boards, adding/removing individual chess pieces, etc. There are also special chess environments used to typeset the main stream of the solution, its side streams and the alternative variants — see section 'Additional environments' for more details.

Puzzles, fairy chess games, checkers, etc. Apart from classical chess games and chess problems there are so-called unusual (fairy) chess games and problems — i.e., with non-standard rules, boards, figures, etc. While the typesetting of chess diagrams with non-standard rules is more or less covered by macros designed for ordinary chess diagrams, this is not the case when we deal with non-standard (for example, non-rectangular) chess boards and unusual chess figures.

Macros from UCHESS.STY extend the flexibility of defining non-standard rectangular and hexagonal chess boards. In CHESS.STY each field may be black or white, occupied by a chess figure or not, but in UCHESS.STY two more variants are included:

- empty field which is outside the board (drawn as an empty square),
- empty field which is a hole inside the board (drawn as a square filled by solid black color).

While neither field can be occupied by a figure, they are drawn in a different style for a clear reason: it is not reasonable to fill with black the entire region outside the non-rectangular board, and it is difficult to distinguish the hole inside non-rectangular board from the white field if it is just empty without an additional marker.

The other aspect essential for fairy chesses is the existence of non-standard figures. While there are no strict rules how to represent these figures graphically (the Ω-chess game is an exception [7]), it is more or less clear that they should be different from the ordinary chess figures. To satisfy the needs of this group of users, UCHESS.STY supports additional pictograms (crosses, bullets) to represent unusual chess figures, and a big bold circle which can modify the meaning of an ordinary chess figure if it is necessary. And in addition to these chess-like figures it is necessary to note, that *checkers* and *checker-like* board games can be considered as fairy chess games as well — so it is natural to include support of checkers in UCHESS.STY also.

Finally, it is necessary sometimes to draw markers, borders, arrows, etc., over chess boards (particularly for child-level chess textbooks). A separate font enables us to draw arrows, etc., over the chess board in the same way as in the picture environment, and it is possible to add plain borders and to make

bold boundaries between the square and hexagonal cells of the board.

Extended chess encoding

As a result of our attempt to cover all these contradictory requirements, a special chess font (or, more strictly, chess encoding) was developed. It includes all classical chess figures (black and white) and some special-purpose figures (wizard, champion, checker, double-checker, crosses and bullets) are also added. Non-symmetrical figures exist in two variants, straight and mirrored (hopefully there are just two figures of that kind, *knight* and *wizard*) — because, for example, it is recommended to use the horse with the head rotated to the left when it represents the knight on chess board, and to the right when it represents the knight in chess text [2].

Editor's note: Surely the figures don't exist as *a* text character, but rather as some sort of character code... We've already been told that in Russian notation there is a two-letter representations.

Each figure exists as a text character, as the figure placed over a black or over a white square field, and as the figure placed over a white, black or gray hexagonal field. (In fact, there is *no* separate figure to be placed over a white hexagonal field — it is enough to have the figure placed over a white square and a simple TEX macro. But the difference between 'text figures' and figures placed over white square fields is essential — see [3] for more details.) In addition there are the empty square and hexagonal fields of corresponding color, the fields filled by a solid black color, the lines used to surround the square field or the hexagonal field by a frame (or to make the solid frame around the whole board), and the solid circle used to modify the meaning of a chess figure.

The final chess encoding and the details of new chess fonts are described in [3].

Chess Informant notation

Aside from chess figures there are many special symbols used to annotate real chess games in an abbreviated manner. The most famous is the international *Chess Informant* system, but there are also other symbols locally accepted in national chess publications. Since these symbols can be used in text only, there is no reason to put them in the same font as ordinary chess figures. Moreover, these symbols should be visually compatible with the ordinary Computer Modern text and, hence, change their style as the style of the main text is changed. LATEX 2_ε has a sophisticated but user-friendly NFSS-system which controls such behaviour nearly automatically. All what we need to do is to fix the list of necessary symbols and the font encoding, to create the fonts for all necessary variants (size/shape/series/family) and, finally, to prepare corresponding .def- and .fd-files and define macros.

Since this part of UCHESS.STY is under preparation now, it does not contain symbol fonts which are compatible with Computer Modern and European Modern and the representation of the notation of *Chess Informant* to a full extent, but there are definite plans to extend the support of the international chess notation to the level where *Chess Informant* notation and national notations are fully supported as well. (It is worth noting that sometimes *Chess Informant* notation and national notations are different — as a result the language-switching commands should control this difference as well.)

Arrows, borders, etc.

Chess figures and chess symbols described in sections 'Extended chess encoding' and '*Chess Informant* notation' are enough for almost all professional chess typesetting. But what about textbooks? Quite often it is necessary to draw the arrows showing the moves of particular figures, to emphasize some fields by special markers, to draw a special border around a chess diagram, to mark explicitly the boundary between two chess fields, etc. — and there are no such elements in our fonts nor macro commands at our disposal!

But this just means that such elements and such macros should be added. Hopefully, there is a good predecessor for this — namely, the LATEX `picture` environment based on restricted LR-mode which enables construction of arbitrary two-dimensional mosaics from discrete elements by explicit specification of individual two-dimensional coordinates. Our environment allows the user to annotate by arrows, markers, borders, bounding lines, etc., for textbook chess diagrams that are organized in the same manner.

Although just now there is no special font with arrows, etc., available to UCHESS.STY for that purpose (and, consequently, there are no macros to draw arrows over the chess board), we have plans to add the necessary extensions.

Multiple chess boards

An essential extension of CHESS.STY is the support of multiple chess boards inside the same document. The board is identified by its name, and by default the board `current` is used. The board contains the description of the current position, the flags showing the state of the game (white or black side is to make

the move, is it possible to castle, etc.) and a special flag identifying the rules of the game. The board can be of non-standard size (square or rectangular) and, besides ordinary fields, can contain special fields — namely, *empty* fields which are outside the board and are displayed as empty white squares (hexagons), and *null* fields which are displayed as black squares (hexagons) and represent the holes inside the standard board. It is possible to make a copy of the current content of a board with a new name, to correct the content of the board in a silent mode without actually typesetting the moves, to change a board's state manually, etc. The boards obey standard TEX rules about block nesting which enables the user to play the game variants and to investigate side game branches without any problems and special precautions.

Additional environments

Is it true that now we have *everything* we need for accurate chess typesetting? Not at all — we need specialized *environments* to mark up chess-oriented text fragments and to mark up logically different chess-oriented material as well. A very good example can be found in the documentation of CHESS.STY — namely, the case where the main stream of the annotated chess game is typed in bold while the alternative variants of the chess game are typed as ordinary text. While these environments for logical mark-up are outside the main style created by Piet Tutelaers, they are an intrinsic component of our package UCHESS.STY.

Acknowledgements

We express our warmest thanks to our colleagues from other countries — their careful attention supports our activity in TEX even in such unfavourable conditions as they are in modern Russia.

Special thanks are to Bogusław Jackowski and Tomasz Przechlewski for valuable consultations, literature references and ready-to-use chess fonts and chess macros used when we worked on UCHESS.STY.

Alexander Berdnikov would like to thank separately Dr. A. Compagner from Delft University of Technology for his long-term friendship and patient attention as a teacher — not necessarily relating to some TEX or LATEX joint works.

References

[1] Piet Tutelaers. A font and style for typesetting chess using LATEX or TEX. Machine-readable document (supplement to CHESS.STY), 1992.

[2] Frank Hassel. A LATEX style for management of correspondence chess games. Machine-readable document (supplement to BDFCHESS.STY), 1993.

[3] Marina Nikulina, Alexander Berdnikov. Chess fonts for chess games and puzzles. — in these Proceedings.

Editor's note: This paper was not in fact presented, and will not appear in the proceedings. Please supply an alternative description.

[4] Zalman Rubinstein. Chess printing via METAFONT and TEX. *TUGboat*, **10**, pp. 170–172 (July 1989).

[5] David Tofsted. An improved chess font. *TUGboat*, **11**, pp. 542–544 (November 1990).

[6] Jan Eric Larson. A chess font for TEX. *TUGboat*, **10**, pp. 351–352 (November 1989).

[7] Ω-Chess Home page: `www.omegachess.com`

[8] Chess game for 4 players: `www.4playerchess.com`

[9] Tomasz Przechlewski, CHESS.STY, private communication.

[10] Wojciech Pijanowski. Gry w które grałem. "Pomorze", Bydgoszcz, 1989.

[11] Е. Я. Гик. Занимательные математические игры. "Знание", Москва, 1987.

Abstract: Omega version 2

John Plaice
School of Computer Science
UNSW
Sydney NSW 2052
Australia
plaice@cse.unsw.edu.au

Yannis Haralambous
187, rue Nationale
F-59800 Lille
Freance
yannis@fluxus-virus.com

Abstract

We present the latest developments of the Omega system: multidirectional typesetting, a new accentuation algorithm, the use of external binary translation processes, enhanced XML/MathML generation, etc. We will also give an overview of developments by various Omega users around the world, for different languages and scripts.

John Plaice

The $\mathcal{NTS}$ project: from conception to birth*

Philip Taylor
RHBNC, University of London, United Kingdom
p.taylor@exch1.rhbnc.ac.uk

Jiří Zlatuška
Masaryk University, Brno, Czech Republic
zlatuska@muni.cz

Introduction

It is an enormous pleasure and privilege to be able to present this paper on $\mathcal{NTS}$ at a TUG conference in Oxford. For almost ten years, $\mathcal{NTS}$ has slowly been evolving from a concept to a reality, and I am delighted to be able to report that $\mathcal{NTS}$ is virtually complete. The fact that we have reached this point is due almost entirely to the efforts of one man: my co-presenter, Karel Skoupý. Karel has work tirelessly on this project, and without his efforts I have no hesitation in saying that we would not be presenting $\mathcal{NTS}$ as a success story today.

Let me start by presenting an overview of today's talk and presentation; We will attempt to cover seven separate areas, including (of course) the mandatory questions and answers at the end. The seven areas to be covered are:

- A brief history of $\mathcal{NTS}$
- TEX, ε-TEX & $\mathcal{NTS}$ compared
- The choice of Java as the language of implementation
- An overview of the classes, object and methods of $\mathcal{NTS}$
- A summary of the *status quo*
- A demonstration of $\mathcal{NTS}$, and comparison with TEX
- Questions & answers

and you will soon realize that my expertise lies very much in the earlier areas; the *implementation details* of $\mathcal{NTS}$ are very much Karel's area, and I will defer to him whenever any explanation of a detailed implementation issue is called for.

Phil Taylor

* The present authors would like to record their grateful thanks to all members of the $\mathcal{NTS}$ and ε-TEX teams, past and present, without whom neither this paper nor $\mathcal{NTS}$ itself could have ever come to fruition. A further debt of gratitude is owed to GUT, in whose *Cahiers GUTenberg* an earlier version of this article first appeared. And, most importantly of all, we would like to thank the sponsors of the $\mathcal{NTS}$ project. both private and corporate, without whose financial support the project could never have succeeded.

Calendar

2000

Nov 13 – Jan 6 — Gutenberg exhibit, including working replica of his original printing press, Louisville Free Public Library, Louisville, Kentucky.

2001

Feb 28 – Mar 3 — DANTE 2001, 24th meeting, Fachhochschule Rosenheim, Germany. For information, visit `http://www.dante.de/dante2001/`.

Mar 26 – 28 — XML World Europe, Amsterdam, Netherlands. For information, visit `http://www.xmlworld.org/`.

Apr 1 – Jun 15 — The Best of the Best: A traveling juried exhibition of books by members of the Guild of Book Workers. Ohio State University Library, Athens, Ohio. Sites and dates are listed at `http://palimpsest.stanford.edu/byorg/gbw`.

Apr 9 – 13 — Seybold Boston, Boston, Massachusetts. For information, visit `http://www.key3media.com/seyboldseminars/boston2001/`.

Apr 29 – May 2 — BachoTEX 2001, 9th annual meeting of the Polish TEX Users' Group (GUST), "Contemporary publishing TEXnology", Bachotek, Brodnica Lake District, Poland. For information, visit `http://www.gust.org.pl/BachoTeX/`.

May 14 – 17 — Congrès GUTenberg 2001, "Le document au XXIe Siècle", Metz, France. For information, visit `http://www.gutenberg.eu.org/manif/gut2001/`.

Jun 4 – 8 — Rare Book School Summer Session, University of Virginia, Charlottesville, Virginia. A series of one-week courses on topics concerning rare books, manuscripts, the history of books and printing, and special collections. For information, visit `http://www.virginia.edu/oldbooks`.

Jun 6 – 8 — Society for Scholarly Publishing, 23rd annual meeting, San Francisco, California. For information, visit `http://www.sspnet.org`.

Jun 13 – 17 — ACH/ALLC 2001: Joint International Conference of the Association for Computers and the Humanities, and Association for Literary and Linguistic Computing, New York University, New York. For information, visit `http://www.nyu.edu/its/humanities/ach_allc2001/`.

Jul 6 – Aug 18 — The Best of the Best: A traveling juried exhibition of books by members of the Guild of Book Workers. Columbia College Chicago Center for Book and Paper Arts, Chicago, Illinois. Sites and dates are listed at `http://palimpsest.stanford.edu/byorg/gbw`.

Jul 13 – 15 — TypeCon 2001, Rochester, New York. For information, visit `http://www.typecon2001.com`.

Jul 16 – Aug 10 — Rare Book School Summer Session, University of Virginia, Charlottesville, Virginia. A series of one-week courses on topics concerning rare books, manuscripts, the history of books and printing, and special collections. For information, visit `http://www.virginia.edu/oldbooks`.

TUG 2001

University of Delaware, Newark, Delaware.
For information, visit `http://www.tug.org/tug2001/`.

Aug 6 – 10 — Intermediate/Advanced LATEX training class.

Aug 12 – 16 — The 22nd annual meeting of the TEX Users Group, "2001: A TEX Odyssey".

Aug 12 – 17 — Extreme Markup Languages 2001: "There's Nothing so Practical as a Good Theory", Montréal, Canada. For information, visit `http://www.gca.org`.

Status as of 1 December 2000

For additional information on TUG-sponsored events listed above, contact the TUG office (+1 503 223-9994, fax: +1 503 223-3960, e-mail: `office@tug.org`). For events sponsored by other organizations, please use the contact address provided.

Additional type-related events and news items are listed in the Sans Serif Web pages, at `http://www.quixote.com/serif/sans`.

Aug 12 – 17	SIGGRAPH 2001, Los Angeles, California. For information, visit `http://www.siggraph.org/s2001/`.
Sep 8	WDA'2001: First International Workshop on Web Document Analysis, Seattle, Washington. For information, visit `http://www.csc.liv.ac.uk/~wda2001`.
Sep 10 – Oct 26	The Best of the Best: A traveling juried exhibition of books by members of the Guild of Book Workers. Dartmouth College, Hanover, New Hampshire. Sites and dates are listed at `http://palimpsest.stanford.edu/byorg/gbw`.
Sep 17 – 20	XML World 2001, San Francisco, California. For information, visit `http://www.xmlworld.org/`.
Sep 23 – 27	EuroTEX 2001, "TEX and Meta: the Good, the Bad and the Ugly Bits", Kerkrade, Netherlands. For information, visit `http://www.ntg.nl/eurotex/`.
Sep 29	29th Annual General Meeting of the Danish TEX Users Group (DK-TUG), Århus, Denmark. For information, visit `http://sunsite.dk/dk-tug/`.
Oct 24 – 26	4th International Conference on The Electronic Document, Toulouse, France. For information, visit `http://www.irit.fr/CIDE2001/`.
Nov 9 – 10	ASM Symposium on Document Engineering, Atlanta, Georgia. For information, visit `http://www.documentengineering.org`.

Institutional Members

American Mathematical Society, *Providence*, *Rhode Island*

Center for Computing Services, *Bowie*, *Maryland*

CNRS - IDRIS, *Orsay*, *France*

College of William & Mary, Department of Computer Science, *Williamsburg*, *Virginia*

CSTUG, *Praha*, *Czech Republic*

Florida State University, Supercomputer Computations Research, *Tallahassee*, *Florida*

Hong Kong University of Science and Technology, Department of Computer Science, *Hong Kong*, *China*

IBM Corporation, T J Watson Research Center, *Yorktown*, *New York*

ICC Corporation, *Portland*, *Oregon*

Institute for Advanced Study, *Princeton*, *New Jersey*

Institute for Defense Analyses, Center for Communications Research, *Princeton*, *New Jersey*

Iowa State University, Computation Center, *Ames*, *Iowa*

Kluwer Academic Publishers, *Dordrecht*, *The Netherlands*

KTH Royal Institute of Technology, *Stockholm*, *Sweden*

Marquette University, Department of Mathematics, Statistics and Computer Science, *Milwaukee*, *Wisconsin*

Masaryk University, Faculty of Informatics, *Brno*, *Czechoslovakia*

Max Planck Institut für Mathematik, *Bonn*, *Germany*

National Institute for Child & Human Development, *Bethesda*, *Maryland*

New York University, Academic Computing Facility, *New York*, *New York*

Princeton University, Department of Mathematics, *Princeton*, *New Jersey*

Space Telescope Science Institute, *Baltimore*, *Maryland*

Springer-Verlag Heidelberg, *Heidelberg*, *Germany*

Springer-Verlag New York, Inc., *New York*, *New York*

Stanford Linear Accelerator Center (SLAC), *Stanford*, *California*

Stanford University, Computer Science Department, *Stanford*, *California*

Stockholm University, Department of Mathematics, *Stockholm*, *Sweden*

University of Canterbury, Computer Services Centre, *Christchurch*, *New Zealand*

University College, Cork, Computer Centre, *Cork*, *Ireland*

University of Delaware, Computing and Network Services, *Newark*, *Delaware*

Universität Koblenz–Landau, Fachbereich Informatik, *Koblenz*, *Germany*

University of Oslo, Institute of Informatics, *Blindern*, *Oslo*, *Norway*

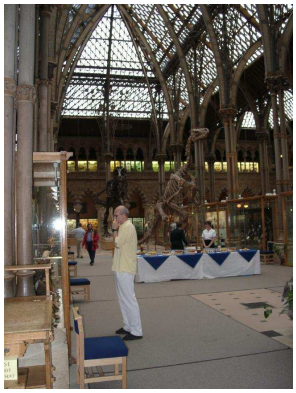
Università degli Studi di Trieste, *Trieste*, *Italy*

Vanderbilt University, *Nashville*, *Tennessee*

Vrije Universiteit, *Amsterdam*, *The Netherlands*

Miscellaneous Photos*

Robin Fairbairns



University Museum

Start of the Banquet

Tea Break with Kim Roberts centre stage

A view down Parks Road

TUG Board members: (L–R) Phil Taylor, Don DeLand, Robin Laakso (TUG Business Manager, seated), Susan DeMeritt, Judy Johnson, Barbara Beeton, and Kaja Christiansen.

* These photos and others scattered throughout the issue are only a few of the photos kindly donated by Alan Wetmore; more may be viewed on the TUG web site at `http://www.tug.org/tug2000/Photos/index.html`.

Participants at the 21st Annual TUG Meeting August 12–16, 2000, Wadham College, Oxford, UK

Benjamin Bayart
(France)

Kaveh Bazargan
Focal Image Ltd (UK)

Nelson Beebe
University of Utah (USA)

Barbara Beeton
American Mathematical Society (USA)

Duncan Bennett
MRC Biostatistics Unit (UK)

Alexander Berdnikov
Institute of Analytical Instrumentation (Russia)

Berend de Boer
(Netherlands)

Thierry Bouche
Université de Grenoble I (France)

Johannes Braams
TEXniek (Netherlands)

Klaus Braune
Universität Karlsruhe (Germany)

Włodzimierz Bzyl
University of Gdansk (Poland)

David Carlisle
NAG/LATEX3 (UK)

Lance Carnes
Personal TEX, Inc. (USA)

Ed Cashin
University of Georgia (USA)

Kaja Christiansen
Aarhus University (Denmark)

Steve Daniels
The Open University (UK)

Donald DeLand
Integre Technical Publishing (USA)

Susan DeMeritt
IDA/CCR La Jolla (USA)

Christine Detig
Net & Publication Consultance (Germany)

Michael Downes
American Mathematical Society (USA)

Gontran Ervynck
K.U. Leuven Campus Kortryk (Belgium)

Robin Fairbairns
University of Cambridge Computer Laboratory (UK)

Emilio Faro Rivas
Escuela de Telecomunicaciones (Spain)

Alexey V. Filippov
Institute of Fine Mechanics and Optics (Russia)

Jonathan Fine
(UK)

Peter Flynn
University College, Cork (Ireland)

Andrew Ford
Ford & Mason Ltd (UK)

James Foster
University of Sussex (UK)

Erik Frambach
(The Netherlands)

Walter Gander
Comlab, University of Oxford (UK)

Jeremy Gibbons
Comlab, University of Oxford (UK)

Gurpreet Gill
Lindsay Ross International Limited (UK)

Peter Giunta
Massachusetts Institute of Technology (USA)

Rosa Maria Gomez Flores
STAR AG (Switzerland)

Michel Goossens
CERN (Switzerland)

Steve Grathwohl
Duke University Press (USA)

Hans Hagen
PRAGMA Advanced Document Engineering (Netherlands)

Jay Hammond
QMW, University of London (England)

Hán Thế Thánh
Masaryk University (Czech Republic)

Ahmed Hindawi
Hindawi Publishing Corporation (Egypt)

Rick Hobbs
Institute of Public Administration (Saudi Arabia)

Alan Hoenig
Knowledge Equity, Inc. (USA)

Mimi Jett
IBM (USA)

Judy Johnson
Personal TEX, Inc. (USA)

Tom Kacvinsky
American Mathematical Society (USA)

Hirotsugu Kakugawa
(Japan)

NG Kalivas
Westminster School (UK)

Anders Källström
Uppsala University (Sweden)

Jonathan Kew
SIL Intl (UK)

Thomas Koch
Dante e.V. (Germany)

Reinhard Kotucha
(Germany)

Johannes Kuester
typoma (Germany)

Robin Laakso
TEX Users Group (USA)

Klaus Lagally
(Germany)

Olga Lapko
Mir Publishers (Russia)

Michel Lavaud
CNRS (France)

Jenny Levine
Duke University Press (USA)

Bernice Lipkin
(USA)

Irina Makhovaia
Mir Publishers (Russia)

Wendy McKay
California Institute of Technology (USA)

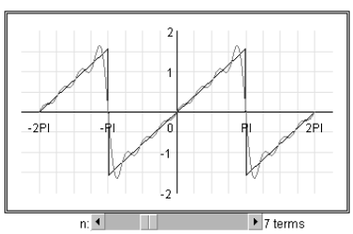
Lothar Meyer-Lerbs
(Germany)

Bruce Miller
NIST (USA)

www.software.ibm.com/techexplorer

techexplorer Hypermedia Browser v3

Live Web Math with Interactivity



$$f(x) = \sum_{k=1}^{n} (-1)^{k+1} \left(\frac{1}{k}\right) \sin(kx), \quad n = 7$$

$$= \sin(x) - \frac{1}{2}\sin(2x) + \frac{1}{3}\sin(3x) - \frac{1}{4}\sin(4x) + \cdots + \frac{1}{7}\sin(7x)$$

techexplorer is a plug-in for Web browsers that allows users to have live, interactive mathematics on an HTML or XML page. The Intro Edition is a free LaTeX or MathML reader. The Pro Edition includes advanced features for truly interactive mathematical collaboration and publishing.

IBM Research announces techexplorer 3, released in conjunction with the MathML International Conference 2000. Special new features include: Interoperability with *Mathematica* 4.1 from Wolfram Research, Inc.; A Macintosh version for both the Professional and Introductory Editions; ActiveX allows techexplorer to interactively work within many Windows applications, including Microsoft IE, Word, and Powerpoint; Support for W3C DOM and MathML standards allows interactivity with applets, multi-media resources, animations, and many other applications.

Write... create... collaborate... techexplorer makes your browser a scientific publishing environment. View LaTeX and MathML files; create interactive documents with Java applets, video or audio clips, animations, and graphics.

NEW FEATURES:
- Macintosh platform
- *Mathematica*™ connectivity
- ActiveX control
- DOM API
- LaTeX-2-techexplorer filter

KEY FEATURES:
- MathML rendering and conversion
- LaTeX viewing and editing
- ActiveX control
- DOM API functionality and support
- Create pop-up and pull-down menus
- In-line video and auto-play audio
- AMS symbol fonts
- Expression editor
- Interact with many tools and applications
- Java and JavaScript

BROWSERS:
- Netscape, MS Internet Explorer, Opera

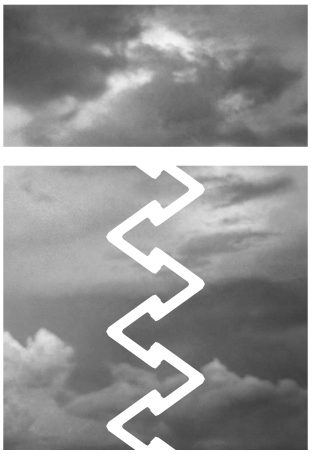
PLATFORMS:
- Pro: Windows 95/98/NT, Macintosh OS 8.6/9
- Intro: Windows, Mac, Linux, AIX, Solaris

$29.95 from ibm.com

introducing

TEXTURES® 2.0

WITH SYNCHRONICITY



AGAIN THE MACINTOSH DELIVERS A NEW TEX WITH A REVOLUTION IN HUMAN INTERFACE.

As computer power has advanced, the Macintosh has consistently been the leader in the human and humane connection to technology, and Textures has consistently led in bringing ease of use to TEX users. First with Textures 1.0, the first truly integrated TEX system. Then with Lightning Textures, the first truly interactive TEX system. Now, with Textures 2.0 and Synchronicity, Blue Sky Research again delivers a striking advance in TEX interactivity and productivity.

With Synchronicity, your TEX input documents are reliably and automatically cross-linked, correlated, or "synchronized" with the finished TEX typeset pages. Every piece of the finished product is tied directly to the source code from which it was generated, and vice-versa. To go from TEX input directly and exactly to the corresponding typeset characters, just click. It's that simple: just click, and Textures will take you instantly and precisely to the corresponding location. And it goes both ways: just click on any typeset character, and Textures will take you directly to the TEX code that produced it. No matter how many input files you have, no matter what macros you use, Synchronicity will take you there, instantly and dependably.

Improve YOUR *performance:*

GET SYNCHRONICITY

BLUE SKY RESEARCH
317 SW ALDER STREET
PORTLAND, OR 97204 USA

800 622 8398
503 222 9571
WWW.BLUESKY.COM