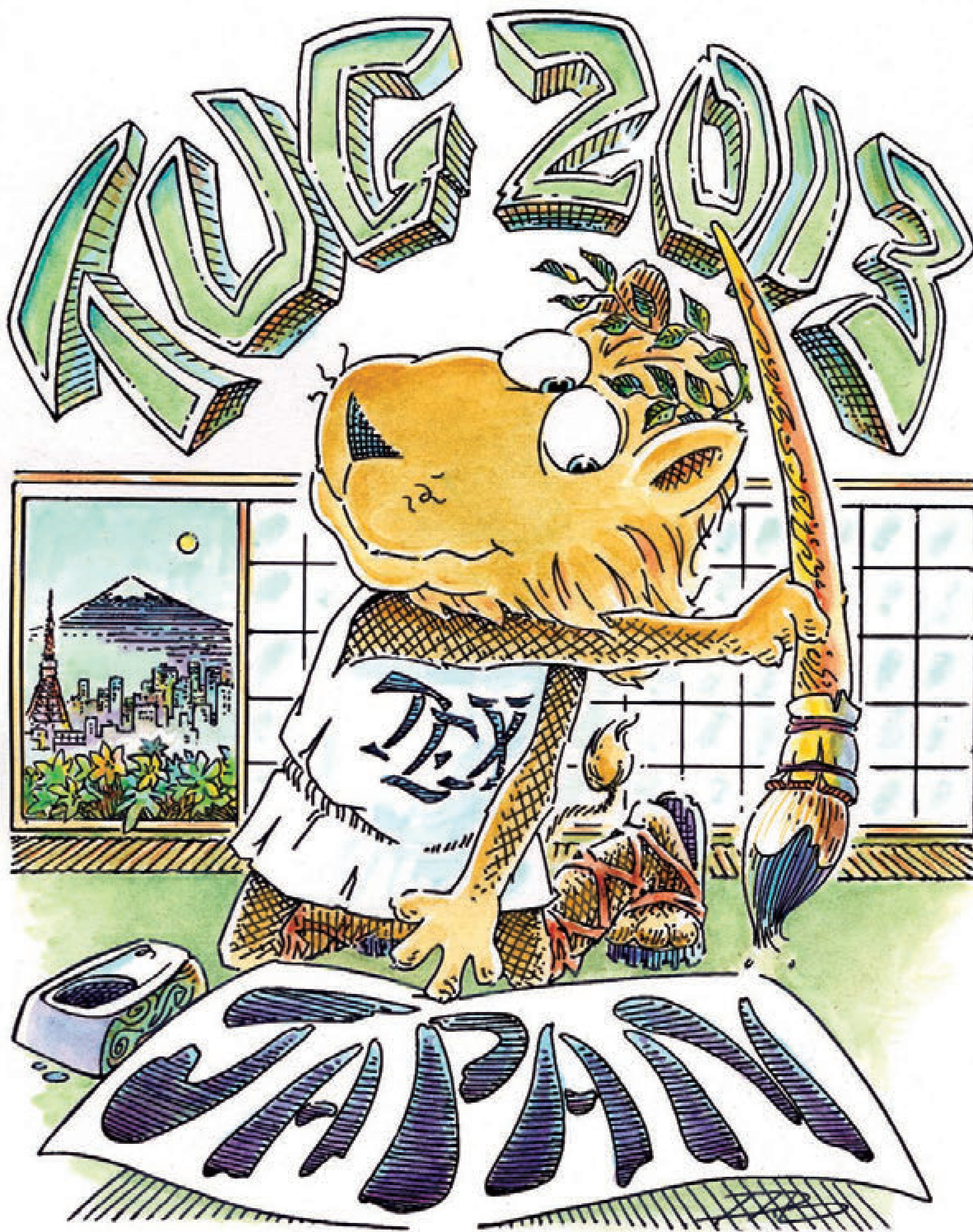


TUG 2013

Graduate School of Mathematical Sciences,
the University of Tokyo — Tokyo, Japan

October 23–26, 2013

TUG 2013
The 34th Annual Meeting of the TEX Users Group

October 23–26, 2013
Graduate School of Mathematical Sciences, the University of Tokyo
3-8-1 Komaba, Meguro-ku, Tokyo, Japan

Printed in Japan (Sanbi Printing Co., Ltd.)

TEX Users Group information

The TEX Users Group (TUG, `http://tug.org`) was founded in 1980 to provide an organization for people who are interested in typography and font design, and/or are users of the TEX typesetting system invented by Donald Knuth.

Some background: TUG is a not-for-profit organization by, for, and of its members, also representing the interests of TEX users worldwide. It is almost entirely member-supported, so if you use any TEX-related programs (TEX, LATEX, ConTEXt, Metafont, MetaPost, Texinfo, et al.), please consider joining TUG (`http://tug.org/join.html`), or another TEX user group (`http://tug.org/usergroups.html`).

Among TUG membership benefits are the journal *TUGboat* (available both in print and online, `http://tug.org/TUGboat`) and the TEX software collection (`http://tug.org/texcollection`, consisting of TEX Live, proTEXt, MacTEX, etc.). TUG also holds an annual TEX conference (`http://tug.org/meetings`), such as the present one in Japan!

What TUG is, what TUG isn't

In essence, TUG is a clearinghouse organization for TEX activities of all kinds. It takes in revenue, mostly as membership dues and donations, with some sales of DVDs and other products (`http://tug.org/store`). Then it disburses that money back out, mostly funding the printing and mailing of *TUGboat* and the production of the software DVDs, and also making some relatively small development grants (`http://tug.org/tc/devfund`) and other projects. It doesn't have the funds to employ programmers as employees; all technical work is done by volunteers. TUG is formally a not-for-profit charitable organization in the USA, and has run essentially at break-even over the years.

Organizationally, TUG is ultimately run by a volunteer board of directors. All members are eligible to run, and directors are elected by membership vote every two years (`http://tug.org/election`).

TUG's past, present, future

In the early years of TEX, TUG was naturally focused on development and porting of the core programs to different systems, as perusing the online *TUGboat* archives shows. The conferences also played a major role in disseminating knowledge and basic information on getting TEX running, and then actually using it.

Nowadays, of course that is all well established, and the focus is on usability and extensions into new areas. As you know, TEX, LATEX, and related programs continue to develop. It is a testament to the flexibility and foresightedness of Knuth's design that TEX is still viable, indeed widely used, today — the web, PDF, OpenType, and much more were not dreamed of when TEX was conceived, and yet it has adapted to everything that has come along, with no signs of demise.

You may have used the TeX executables on a new computer — TUG supported the development and distribution of those programs. You've hopefully found style files or documentation online that helped you get your job done — TUG supported the CTAN archive that is a common collection point for those. You may have have used new fonts already set up for TeX, or an editor or utilities that make writing your papers go more smoothly, and by now you won't be surprised that it may well be that TUG helped those to happen.

TUG exists as part of the TeX community, helping it to remain the vibrant place it has always been.

Summary of links

- `http://tug.org` — TUG home page.
- `http://tug.org/join.html` — join (or renew with) TUG.
- `http://tug.org/texlive` — TeX Live software.
- `http://tug.org/TUGboat` — *TUGboat*.
- `http://tug.org/aims_ben.html` — aims of the organization & benefits of joining.

Karl Berry
September 27, 2013

Book fair and special discount for participants

45% discount for all Manning publications!

Manning Publications Co.

We are grateful to Manning Publications Co. for their generous gift to the participants of TUG '13: a 45% discount for all Manning publications. Just use the discount code ___________ when ordering from their site, http://www.manning.com/. The code is valid from one week before the conference till the end of the conference.

$9,99! The LaTeX Companion 2nd Ed.

Frank Mittelbach

I'm pleased to announce that *The LaTeX Companion* second Edition is finally available in eBook format in addition to the printed book.

Pearson is offering the eBook as a bundle: PDF, epub, and mobi (Kindle) format without DRM restriction (only watermarked). The list price will be $29,99. The eBook may be available from other resellers, but typically you will then only get one format and some resellers might apply DRM.

As a special promotion we will make the eBook bundle available to any interested particiant of the TUG conference 2013 at a price of $9,99.

To use this offer, follow the book link at http://www.latex-project.org/guides/books.html that directs you to the book page on the publisher site. As part of the checkout process you will be given the opportunity to enter a discount code. The code for the TUG conference participants is _____________ and it is valid until October 31 and will reduce the price to $9,99.

For the general public there is longer running promotion at price of $14,99 and code for this is **LATEXT2013** valid until the end of the year. You are invited to pass this on to anybody interested in the eBook.

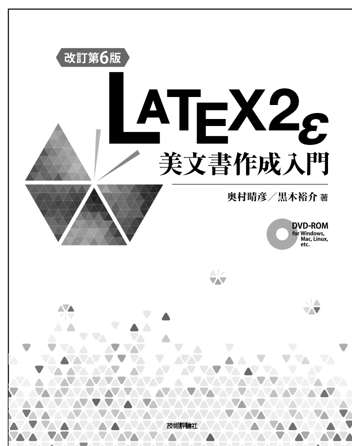
Japanese TeX Book Fair

A **Japanese TeX Book Fair** will be set up (courtesy of University of Tokyo CO-OP) just outside the lecture room, during the following hours:

23rd October, 15:35–15:55
24th October, 14:50–15:10
26th October, 10:50–11:10, 12:20–12:40

A great chance to shop from the widest range of TeX-related Japanese books in the market. Please note that only Japanese yen in cash will be accepted.

We are also pleased to announce an exclusive discount offer of the sixth edition of "LaTeX 2$_\varepsilon$ *Bibunsho Sakusei Nyumon* (A Guide to creating beautiful documents with LaTeX 2$_\varepsilon$)" by Haruhiko Okumura and Yusuke Kuroki. This book, due to be released on 23rd October, will be available to TUG 2013 participants at a special discount price of 3,000 yen (tax included). On 26th October, a book signing event will be held. Anyone who bought this book can have their book autographed by the authors. Don't miss this special offer!



The sixth edition of "LaTeX 2$_\varepsilon$ *Bibunsho Sakusei Nyumon*" is the latest update of the most widely-read introduction to LaTeX available in Japanese. First issued in 1991 as "LaTeX *Bibunsho Sakusei Nyumon*" (by Haruhiko Okumura) this book has always been a must for LaTeX users in Japan. Covering all the major information about LaTeX from basics to moderate, it is the perfect navigator to create beautiful Japanese documents with LaTeX. This sixth edition, now with Yusuke Kuroki as co-author, comes with a DVD including TeX Live 2013 Windows/Mac installers especially designed for Japanese users. Special Thanks to: Norbert Preining and the TeX Live Team, Noriyuki Abe (for Windows installer), Yusuke Terada and Munehiro Yamamoto (for Mac installer and binaries).

Special exhibition of used and out-of-print books

Along with the Japanese TeX Book Fair, the organizer is also holding a **special exhibition of used and out-of-print books**, featuring 36 or more items that are now hard to find in the market. Those who are interested can purchase the books on the 26th. Please note that only Japanese yen in cash will be accepted.

TEX
USERS
GROUP

TUG 2013 Program

Note: The asterisk mark (∗) after a name means that the indicated person will be presenting.

Tuesday, 22 October

6:00pm–8:00pm

Registration and Reception

Wednesday, 23 October

9:00am–9:15am

Opening Message

Steve Peter and Haruhiko Okumura

9:15am–9:50am

TiCL: The prototype

Didier Verna

9:50am–10:05am

LISP on TEX: A LISP interpreter written using TEX macros

Shizuya Hakuta

10:05am–10:40am

A gentle introduction to PythonTEX

Andrew Mertz* and William Slough

10:40am–11:00am

Break

11:00am–11:10am

TUTORIAL **Introduction to Tutorials**

KUROKI Yusuke

11:10am–0:40pm

TUTORIAL **An Introduction to the Structure of the Japanese Writing System**

YADA Tsutomu

0:40pm–1:40pm

Lunch

1:40pm–2:15pm

The incredible tale of the author who didn't want to do the publisher's job

Didier Verna

2:15pm–2:50pm

How we try to make working with TEX comfortable

Hans Hagen

2:50pm–3:35pm

TUTORIAL **Indexing Makes Your Book Perfect**

SHIKANO Keiichiro

3:35pm–3:55pm

Break

3:55pm–4:30pm

How I use LaTeX to make a product catalogue that doesn't look like a dissertation

Jason Lewis

4:30pm–5:05pm

TeX in educational institutes

Yasuhide Minoda

5:05pm–5:40pm

Online Publishing via pdf2htmlEX

Lu Wang and Wanmin Liu*

5:40pm–6:25pm

The stony route to complex page layout

Frank Mittelbach

After a short break, we will have another session (not in the official program).

6:35pm–7:10pm++

What is ConTeXt? A short introduction

Hans Hagen

Thursday, 24 October

9:00am–9:30am

Making math textbooks and materials with TeX+KETpic+hyperlink

Yoshifumi Maeda and Masataka Kaneko

9:30am–10:05am

Wind roses for TeX documents

Alan Wetmore

10:05am–10:40am

Plots in LaTeX: Gnuplot, Octave, make

Boris Veytsman

10:40am–11:00am

Break

11:00am–0:30pm

TUTORIAL **Japanese Typeface Design—Similarities and Differences from Western Typeface Design**

TAKATA Yumi

0:30pm–1:30pm

Lunch

1:30pm–2:05pm

LaTeX and graphics

Aleksandra Hankus and Zofia Walczak

2:05pm–2:50pm

LaTeX3: Using the layers

Frank Mittelbach

2:50pm–3:10pm

Break

3:10pm–4:40pm

TUTORIAL **Japanese Text Layout—Basic Issues**

YABE Masafumi

4:40pm–4:55pm

TUTORIAL **Some notes on TEXt processing**

KUROKI Yusuke

4:55pm–5:15pm

Break

5:15pm–5:50pm

Development of TeXShop—the past and the future

Yusuke Terada

5:50pm–6:25pm

TEX Live for Android: Development and usage

Clerk Ma

6:25pm–6:50pm

TEX Live Manager's rare gems: User mode and multiple repository support

Norbert Preining

6:50pm–7:00pm

Guidance for the excursion

Organizing Committee

Friday, 25 October

ALL DAY

Excursion to the Printing Museum, Tokyo, etc.

See pp. 15–19.

Saturday, 26 October

9:00am–9:35am

Tsukurimashou: a Japanese-language font meta-family

Matthew Skala

9:35am–10:10am

upTEX—Unicode version of pTEX with CJK extensions

Takuji Tanaka

10:10am–10:45am

A short history of TEX in China

Jie Su* and Clerk Ma

10:45am–10:50am

Group Photo

10:50am–11:10am

Break

11:10am–11:45am

A case study: Typesetting old documents of Japan

Ken Nakano* and Hajime Kobayashi

11:45am–0:20pm

A case study on TeX's superior power: Giving different colors to building blocks of Korean syllables

Jin-Hwan Cho

0:20pm–1:20pm

Lunch

1:20pm–1:55pm

The multibibliography package

Michael Cohen*, Yannis Haralambous, and Boris Veytsman*

1:55pm–2:30pm

TANSU: A workflow for cabinet layout

Pavneet Arora

2:30pm–3:05pm

Typesetting and Layout in Multiple Directions

John Plaice

3:05pm–3:25pm

Break

3:25pm–4:00pm

Making mathematical content accessible using Tagged PDF and LaTeX

Ross Moore

4:00pm–4:35pm

How we move(d) on with math

Hans Hagen

4:35pm–5:10pm

The XyMTeX system for publishing interdisciplinary chemistry/mathematics books

Shinsaku Fujita

5:10pm–5:45pm

Distributing TeX and friends: methods, pitfalls, advice

Norbert Preining

5:45pm–5:50pm

Closing Message

Norbert Preining

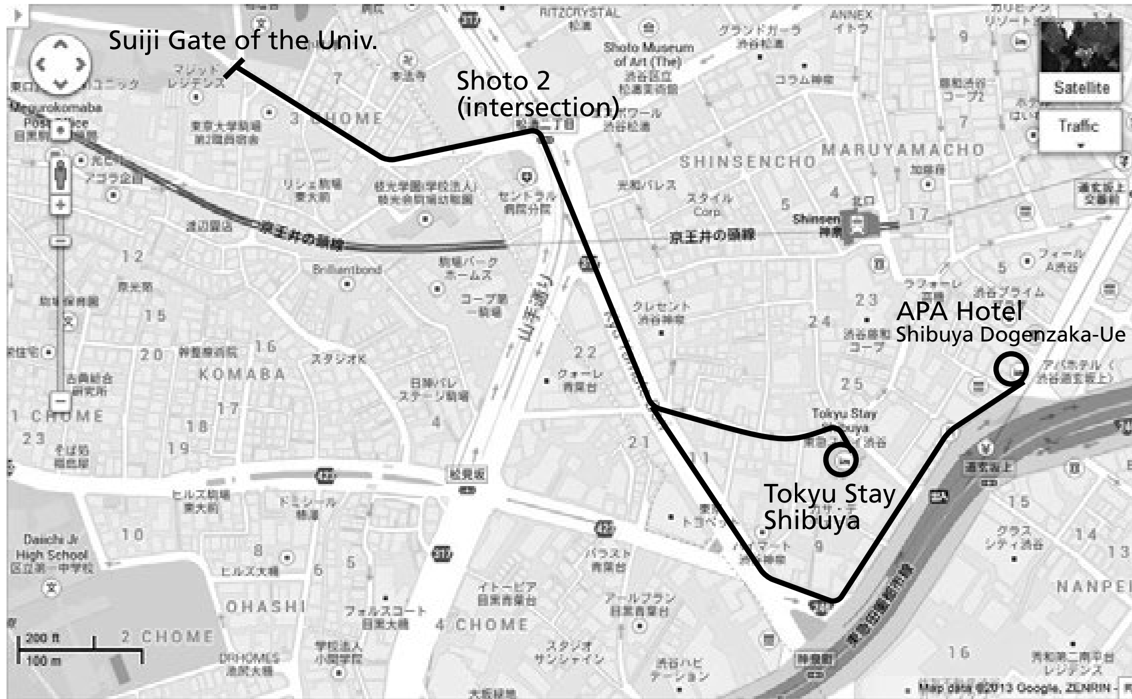
6:00pm–8:30pm

Banquet

Access Map

From Tokyu Stay Shibuya/APA Hotel to Conference Venue

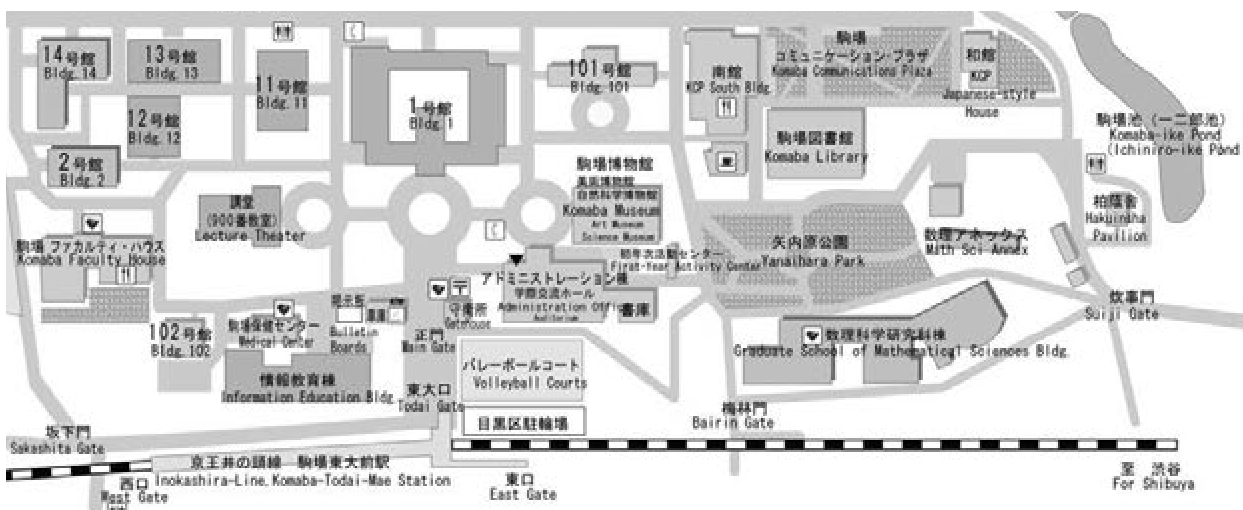
If you are staying at Tokyu Stay Shibuya or APA Hotel Shibuya-Dogenzaka-Ue, the best way to get to the conference venue is on foot; it takes 10 to 15 minutes (less than 1 km). Please see the map below.



From Shibuya Excel Hotel Tokyu or Shibuya Station to Conference Venue

If you are staying at Shibuya Excel Hotel Tokyu, the best way to get to the conference venue is to board the Keio Inokashira Line local train at Shibuya Station. The conference venue is very close to Komaba-Todaimae Station, second stop from Shibuya. Take the East Exit, turn left and go out the station (Todai Gate), and you will see the Main Gate of the University ahead. Then, turn right to a narrow path, —you don't have to go through the Main Gate—go along the path, seeing roses on your left and volleyball courts on your right. This path leads to the West entrance of the Graduate School of Mathematical Sciences Building, the conference venue.

Campus Map: Komaba I Campus, the University of Tokyo



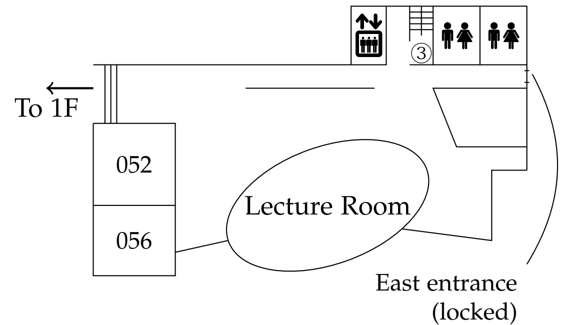
- **NO SMOKING** on campus except in smoking areas
- Banquet: Italian Tomato Cafe Jr. (next to KCP South Building)

Graduate School of Mathematical Sciences Building

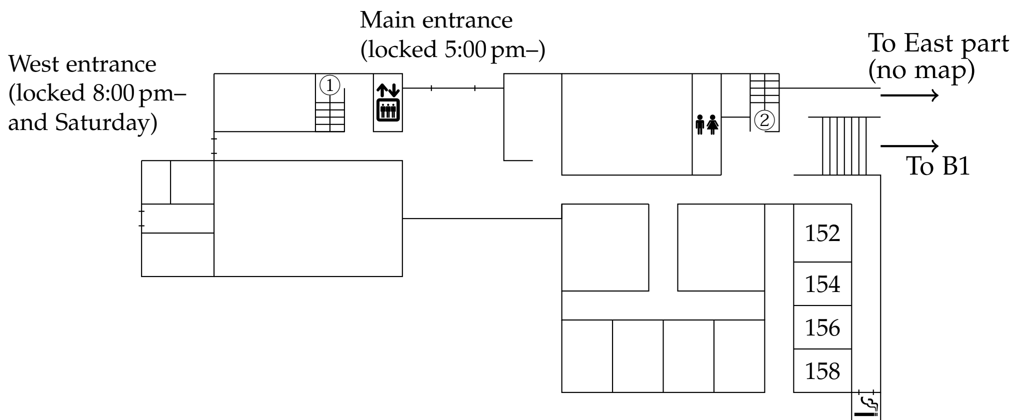
: elevator : toilet : smoking area

- **NO SMOKING** inside the building (except in smoking area)
- Lectures: Lecture room (B1)
- Rooms for lunch: 211 (2F), 152, 154, 156, 158 (1F), Common Room (2F, Saturday only)
- Lunch handout: 211 (2F)
- Reception party: Common Room (2F)
- Book fair: 056 (B1)
- Room for discussions: 052 (B1)

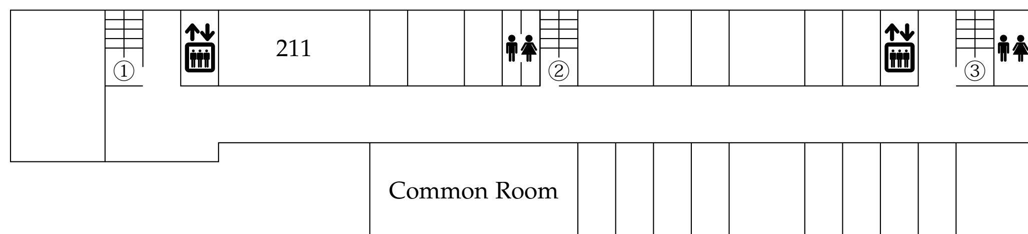
B1



1F (West part, ground floor)



2F



Lunch

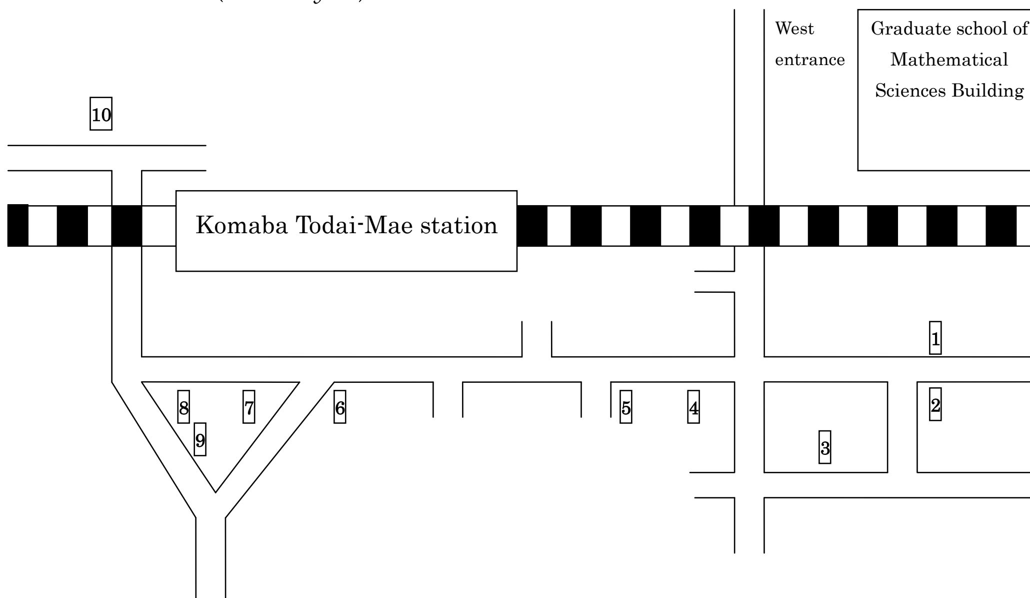
If you have registered full-term (including speakers and students) or one-day (with lunch, in the Japanese form), the organizer offers you a lunch box. It will be handed out in Room 211 when lunch time starts. Please do not miss your lunch box; halfway through lunch time, the remainders will be given away.
For those who have chosen a no-lunch option, there are a quite a few restaurants in and outside the campus:

Restaurants in the campus (see Campus Map)

- University cafeteria (KCP South Building)
- Italian Tomato Cafe Jr. (casual Italian, next to KCP South Building)
- Lever son verre (French, at Komaba Faculty House)

Restaurants outside the campus

1 ムスカン Muskan (Indian curry)
2 英香 Eika (seafood)
3 中華井上 Chuka Inoue (casual Chinese)
4 菱田屋 Hishitaya
5 キッチン南海 Kitchen Nankai
6 ルーシー Lucy (curry)
7 つけめん 駒鉄 Tsukemen Komatetsu (*ramen*)
8 McDonald’s
9 苗場 Naeba (casual Chinese)
10 (1F) 満留賀 Maruka (*soba*, *udon*, *donburi*)
10 (2F) 楓 Kaede (*okonomiyaki*)



Excursion

On 25th October, a full day excursion is organized as part of the official program. It consists of three parts: a museum tour, a letterpress printing workshop, and a calligraphy workshop. Note: *Please wear casual clothing that you would not mind staining, since the workshops will use waterproof ink.*

Schedule

8:20 am Bus pick-up at Conference Venue
8:35 am Bus pick-up at Dogenzaka Ue, Shibuya

9:45 am–10:00 am Registration at the Printing Museum, Tokyo

Group A	
10:00 am–0:00 pm	Letterpress Printing Workshop
0:00 pm–0:55 pm	Lunch
1:00 pm–2:00 pm	Museum Tour (1)
2:05 pm–3:30 pm	Bus
2:35 pm–4:20 pm	Calligraphy Workshop
4:25 pm–4:35 pm	Bus
4:40 pm–6:00 pm	Museum Tour (2)

Return by bus or disband on site

Group B	
10:00 am–11:30 am	Museum Tour (1)
11:30 am–0:15 pm	Lunch
0:20 pm–0:45 pm	Bus
0:50 pm–2:35 pm	Calligraphy Workshop
2:40 pm–2:50 pm	Bus
2:55 pm–4:00 pm	Museum Tour (2)
4:00 pm–6:00 pm	Letterpress Printing Workshop

Return by bus or disband on site

Group C	
10:00 am–11:45 am	Museum Tour (1)
11:45 am–0:55 pm	Lunch
1:00 pm–3:00 pm	Letterpress Printing Workshop
3:00 pm–3:45 pm	Museum Tour (2)
3:50 pm–4:15 pm	Bus
4:20 pm–6:05 pm	Calligraphy Workshop

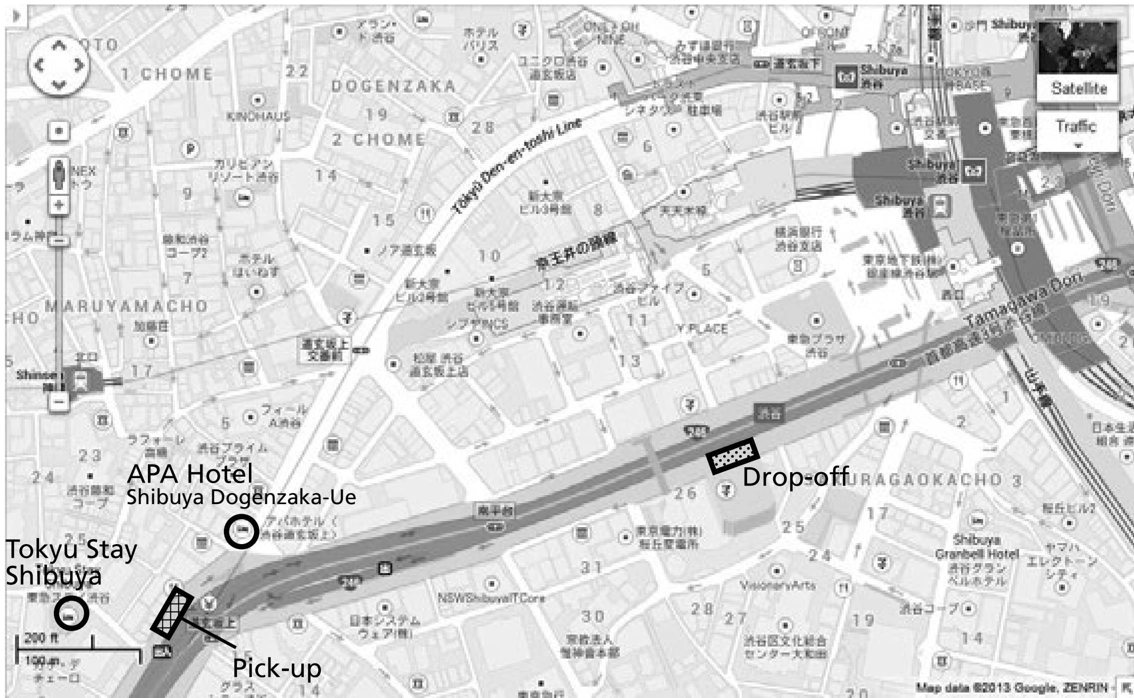
Return by bus or disband on site

7:05 pm Bus drop-off at Shibuya
7:20 pm Bus drop-off at Conference Venue

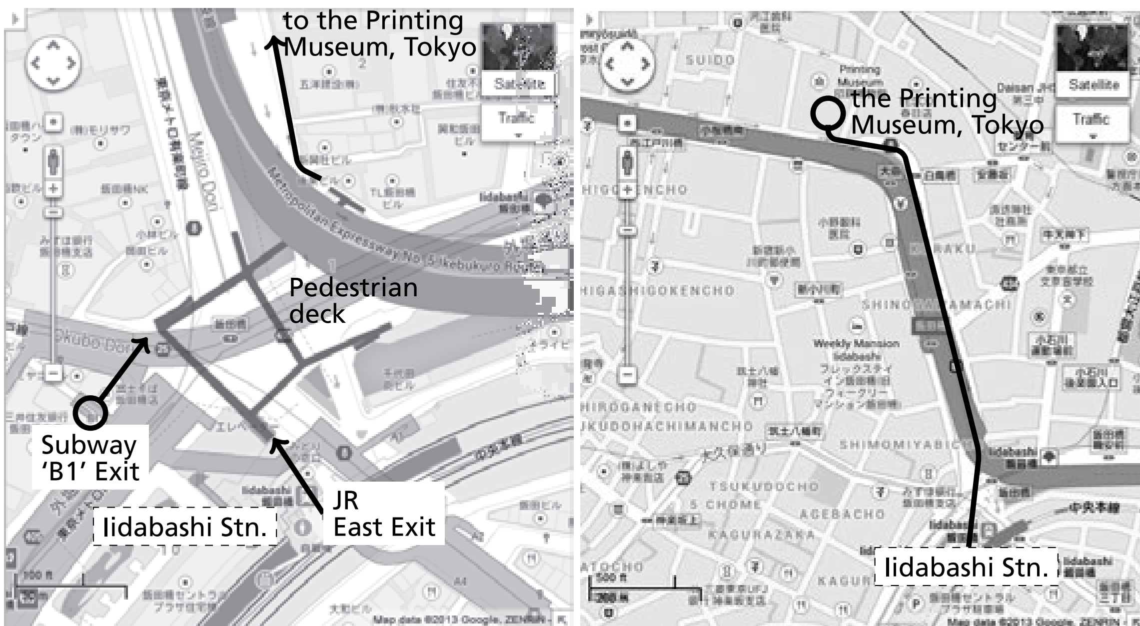
Pickup for the Excursion

The excursion will be held mainly in the Printing Museum, Tokyo. You have a number of options on how to get to the Printing Museum, which you can choose from according to your hotel location and/or your familiarity with Tokyo:

- A charter bus will pick you up at the Main entrance of the conference venue at 8:20 am;
- This bus will also pick you up at Dogenzaka Ue, Shibuya at 8:35 am (the organizer will be waiting at the entrance of Tokyu Stay Shibuya/APA Hotel at 8:30 am);



- Come on your own to the Printing Museum by 9:45 am (15-minute walk from Iidabashi Station).



(Zoom in on Iidabashi Station) (Panoramic view)

Letterpress Printing Workshop

at the *Printing House*, the Printing Museum, Tokyo

Abstract This workshop allows you to typeset and print your own name using a letterpress. We will work in groups of three, with one or more Japanese-speaking person in each group to help with *katakana*. The procedure is as follows: 1) Select metal types corresponding to your name, set it in the composing stick, and position your name in the center. 2) Bray ink, have the typeset mounted onto the press, and make an impression on paper. 3) Separate the printed sheet to make a bookmark, place each bookmark between absorbing sheets to let it dry. Each attendee can take home several bookmarks with your name in *katakana*.

(Preparation)

1 Please leave your baggage in the Gutenberg Room.
2 Make a group of three. Make sure each group has at least one Japanese-speaking person.
3 Put on the apron, which is on the table.
✎ Note: Don't touch the black disks on the table—the surface is soaked with ink!

3

(Typesetting)

4 If you are not familiar with *katakana* expression, use your name plate as sample. Have the Japanese-speaking person in your group help you with *katakana* selection.
5 Hold the composing stick in your left hand.
6 Pick the metal types that correspond with your name from the case and set it inside the stick one by one, with the nick upwards.
7 Place a 1 em quad between your first and last name. Typesetting is now complete.
8 *Centering.* Place quadrat metal pieces evenly on either side so that your name is positioned in the center. In order to bind the lines tightly together, you should first focus on placing wider quads on the edges, then insert narrower ones in a position closer to the text. When inserting the last (and the narrowest) quad, take out the widest quad, insert the narrowest quad, then place the widest quad in position again.
9 Set an interline lead strip above the type line. The forme will be mounted onto the press by the instructor.

5

(Printing)

10

10 *Trial run*. Position a sheet of paper on the printer. Pull the bar halfway down and up again twice, so that the ink transfers from the disk to the roller. Then, pull the bar all the way down; allow some time to make an impression on the paper before pulling back the bar in its original position.

11 *Confirm settings*. Ensure that your name is as shown on your name plate. If not, have the instructor fix the type. Leave the printed sheet on the stand to let it dry.

12 *Production run*. When you have confirmed the settings, perform the production run. Each person has two sheets of paper for this.

(Cutting)

13

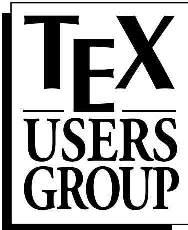
13 Separate the printed sheet into three pieces. Fold along the precut lines, and tear off slowly.

✎ **Note**: Since it takes more than a day for the ink to dry completely, be careful *not to touch the printed part*. Should the print be blurred or stained, please notify the instructor so that you can perform another run.

14 Place the bookmarks, back to back, between absorbing sheets. Let it dry for more than 24 hours.

(End)

15 Please return your apron on the hook on the side of the table before you leave.


TEX
USERS
GROUP

Tutorials

As a TEX user, you might have noticed that this city is filled with so many different characters. You might be wondering, how does the Japanese language function with so many kinds of symbols? How are they arranged and classified? How many glyphs does a Japanese computer system need? What are the rules of Japanese typesetting?

We have prepared a series of tutorials on a range of topics including the basics of the Japanese writing system. It consists of four invited presentations and one supplement:

23 October, 11:10 am–0:40 pm
YADA Tsutomu, *An Introduction to the Structure of the Japanese Writing System*

23 October, 2:50 pm–3:35 pm
SHIKANO Keiichiro, *Indexing Makes Your Book Perfect*

24 October, 11:00 am–0:30 pm
TAKATA Yumi, *Japanese Typeface Design — Similarities and Differences from Western Typeface Design*

24 October, 3:10 pm–4:40 pm
YABE Masafumi, *Japanese Text Layout — Basic Issues*

24 October, 4:40 pm–4:55 pm
KUROKI Yusuke, *Some notes on Japanese TEXt processing*

This tutorial should also serve as an introduction for the excursion on 25th October.

We hope these sessions help you get an idea of the Japanese writing system, the norms and practices underlying it, and how it is different from other writing systems of the world, especially the alphabet.

—KUROKI Yusuke

Indexing Makes Your Book Perfect

SHIKANO Keiichiro

Most of you already know how to make books using LaTeX. And some of you might know how to make back-of-the-book indexes with LaTeX. However, are you ready to worry about how the index of your book should be? Or, if you have already gone through a trouble of writing or editing books, have you actually taken advantage of indexing in your work?

The index, which would be inserted at the back of your book, is not just a reference list of words appearing in your book. Picking out keywords or chunks of text from your manuscript, then arranging them in another way—usually in alphabetical order, often complements your book. In other words, you can exploit indexing to make your book better!

Through this tutorial, you will find what is required for good indexes, how indexing helps you and your readers, and some techniques in making back-of-the-book indexes with LaTeX. On top of that, in non-alphabetical languages, you cannot just make use of makeindex or xindy, mainly because these languages don't have any concept of alphabetical order. So, I will also go over actual cases of making back-of-the-book indexes in non-alphabetical languages.

Japanese Typeface Design
—Similarities and Differences from Western Typeface Design

TAKATA Yumi

What is Japanese typeface design about? As a Japanese type designer for nearly 30 years, I will explain what it is to design a Japanese typeface, what it does and does not have in common with designing a Western typeface.

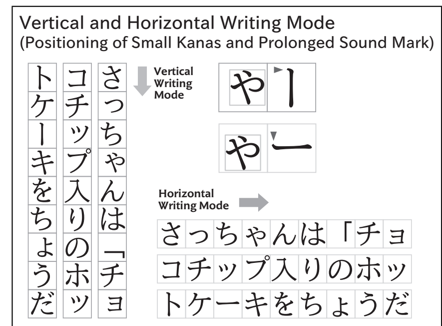
First, we will take a quick look at the history of Japanese characters, in particular how the shapes of the characters have evolved through the times.

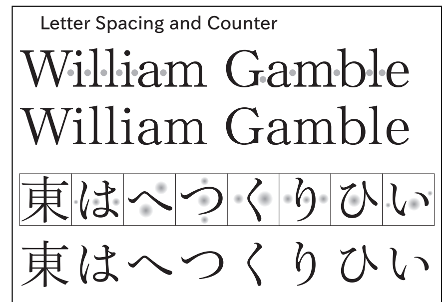
Then I will illustrate the process of Japanese typeface design in detail. Japanese typeface designers face the challenge of dealing with more than 9,000 characters and multiple constituent scripts. Some examples will be given of the various accommodation techniques we use to create readable and visually appealing typefaces, including adjusting for optical illusion.

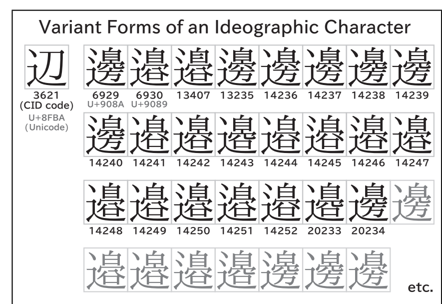
Another big challenge we face is the vertical and horizontal writing modes. I will show how we fine-tune the glyph design of each character, one by one, for both vertical and horizontal writing modes.

Finally, the complications related to Japanese coded character sets will be briefly explained.

I hope my presentation gives you a grasp of the Japanese typeface design and leads to further discussion.







Japanese Text Layout—Basic Issues

YABE Masafumi

This part of tutorial presents basic issues concerning page formats and typesetting methods applied to the main text of a Japanese book with reference to the typographic characteristics of Japanese writing system. The issues to be discussed are threefold. The first section focuses on the text direction, vertical or horizontal writing mode, which depends on an editorial decision and commands in many ways the page layout as well as the printed forms of a Japanese text. The second section concerns the typesetting methods applied to the basic Japanese text as a sequence of characters without spaces between words, and illustrates relevant typographic building blocks in line composition rules with emphasis on the functional importance of punctuation marks and their surrounding spaces for line and paragraph adjustments. The last section addresses several issues about the methods of mixed composition of Japanese and Western texts, presenting major technical problems relating to differentiation and harmonization of typographically heterogeneous elements in sequential texts: Western text in the context of main horizontal or vertical Japanese text as well as Japanese text in the context of main Western text.

Abstracts

Note: The asterisk mark (∗) after a name means that the indicated person will be presenting.

TiCL: The prototype

Didier Verna

9:15am–9:50am, 23 October

Last year, I presented some ideas about using one of the oldest programming language (Lisp), in order to modernize one of the oldest typesetting systems (TEX). This talk was mostly focused on justifying the technical fitness of Lisp for this task. This time, I would like to take the opposite view and demonstrate a prototype, from the user's perspective. This will involve showing what a TiCL document could look like, the implications in terms of typesetting vs. programmatic features, and also in terms of extensibility (relating this to package authoring).

LISP on TEX: A LISP interpreter written using TEX macros

Shizuya Hakuta

9:50am–10:05am, 23 October

Although TEX macros are useful, it is difficult to write macros for novice users. To make easier to use, there are some researches that combine TEX and another programming language. There are a couple of approaches: calling an outer interpreter, or embedding additional language in a family of the TEX engine. Nevertheless, we have taken another approach because TEX is a Turing machine: implementing a language processor with TEX macros. The production, called 'LISP on TEX', allows us to embed LISP scripts in a LATEX document. The interpretor is written only with TEX macros and it has been archived in CTAN (macros/latex/contrib/lisp-on-tex). In this talk, we would like to illustrate how to use it and contrast with LuaTEX, PerlTEX, or related approaches.

A gentle introduction to PythonTEX

Andrew Mertz and William Slough*

10:05am–10:40am, 23 October

The PythonTEX package allows authors to combine computational and typesetting tasks by embedding Python code in TEX documents. This package allows access to many powerful Python modules, providing support for such things as symbolic mathematics, plotting, arbitrary precision numerical calculations, and networking. Python's intuitive syntax, popularity, and extensibility together with TEX's formatting strengths make them a logical system for programming documents. By examining a variety of examples, we will provide an overview of the capabilities and possibilities of PythonTEX.

The incredible tale of the author who didn't want to do the publisher's job

Didier Verna

1:40pm–2:15pm, 23 October

In this talk, I will relate a recent experience of mine: writing a book chapter for a publisher who doesn't have a clue about typesetting. I will confess my futile attempt at using TEX for writing the chapter in question. I will then describe the hell that descended upon me for daring to do that. I will however admit that the hell in question would have been even greater, had I not done so. I will give this talk both crying and laughing, and I will seek your comfort.

How we try to make working with TEX comfortable

Hans Hagen

2:15pm–2:50pm, 23 October

Just as book and music production is under pressure, so is the way we produce documents. We're accustomed to instant rendering in browsers and even if WYSIWYG is not that important when most of the time is spent on writing instead of messing with the look and feel, there is the comfort factor to keep in mind. The last few years I have spent quite some time on a comfortable edit-proofing cycle: from advanced syntax highlighting to fast rendering. Do such things matter and is it worth the effort?

How I use LaTeX to make a product catalogue that doesn't look like a dissertation

Jason Lewis

3:55pm–4:30pm, 23 October

I run a small wholesale and distribution business in Australia. I created a program that uses LaTeX to produce PDF for a full colour printed catalogue. The catalogue is about 70 pages, contains 8–10 full page colour adverts, has a table of contents and a product index, and usually has around 800–1000 products listed. We produce a new catalogue every six months.

The program takes data from our accounting system about the price of products, combines it with some other data and produces a LaTeX file that can be compiled to a PDF. The PDF is then ready for sending to the printer. The program greatly changed where the work load was for producing a catalogue. It used to be a lot of copying and pasting of data into Excel or InDesign, but now the work is mostly in data entry and ensuring the data is correct and ready for export to produce the catalogue.

The program is written in Perl and Perl's Template Toolkit; it also integrates MS Access and MS SQL server.

I faced numerous difficulties in building this system.

1. LaTeX documents don't natively look very much like a glossy catalogue.
2. The system had to be usable by non-technical people to build new catalogues.
3. Windows file paths don't map well in LaTeX.
4. Many characters that users want to use break LaTeX.
5. Making a templating system to produce the LaTeX from a database of products.
6. Integration of MS Access, SQL, Template Toolkit and Perl to produce LaTeX.

In my presentation I will outline how I worked around these difficulties to make a system for users to build new product catalogues when they need one.

TEX in educational institutes

Yasuhide Minoda

4:30pm–5:05pm, 23 October

Tokyo Educational Institute (Tetsuryokukai) is a preparatory school specializing in the entrance exam for Tokyo University. We use TEX for our texts, workbooks, other handouts, and even for internal documents and memorandums.

We used to use other software, but we swiched to TEX and made our original documents (over 100,000 pages) into TEX files over the last few years.

In Tetsuryokukai, we now have over 200 teachers, with various levels of computer skill, so in order to introduce TEX we:

- developed related software (automatic installer, TEX2img and so on),
- prepared various style files,
- educate and motivate teachers.

In this presentation, I would like to present what we have been doing in our company. Our company can be an interesting and helpful example of introducing TEX throughout an institution, especially in the field of education.

Online Publishing via pdf2htmlEX

*Lu Wang and Wanmin Liu**

5:05pm–5:40pm, 23 October

The Web has long become an essential part of our lives. While web technologies have been actively developed for years, there is still a large gap between web and traditional paper publishing. For example, the PDF format, the de facto standard for publishing, is not supported in the HTML standard; and the most powerful typesetting system TEX cannot be integrated perfectly.

Despite of the long history of people trying to convert PDF or TEX into HTML, some are focused on only a small fraction of features, e.g. text, formulas or images; some are too old to support latest features in the HTML standard such as embedded fonts or linear transformations (e.g. rotation); some display everything in images at the cost of larger sizes.

In this article, a new approach is attempted to attack this issue. We introduce an open source software, called pdf2htmlEX, which is a general PDF to HTML converter with high fidelity. It tries to present PDF elements with corresponding native HTML elements, in order to achieve high accuracy and small size. The flexible design also makes it useful for different use cases in online publishing. Obviously TEX users can be immediately benefited with zero learning cost, just like 'dvipdf' while people were still using DVI.

More information are available at the home page: `https://github.com/coolwanglu/pdf2htmlex`

Making math textbooks and materials with TeX+KETpic+hyperlink

Yoshifumi Maeda and Masataka Kaneko

9:00am–9:30am, 24 October

Because of its precision and simplicity, the graphics capability originally present in TeX should have great potential in mathematics education. However, it seems to be burdensome for usual TeX users to fully utilize such capability. Although including the graphical images generated by using computer algebra systems (CAS) is a typical alternate approach, the resulting documents tend to become inefficient for actual use in classroom.

The CAS macro package named KETpic is one of the most hopeful candidates for realizing handy and efficient use of TeX graphics. For instance, it enables us to edit high-quality math textbooks and materials containing

- 2D-graphics which are precise in shape and length as in Sample 1, and
- 3D-graphics which are readily understandable as in Sample 2.

In this talk, it is emphasized that the programmability of KETpic (associated with CAS) and TeX could make the use of TeX more flexible. For example, in Sample 3,

- many documents with graphics can be readily generated by using both for-loop programming and "meta commands" of KETpic, and
- those documents can be readily linked also by using "hyperref"' package of TeX(connected to "hypertextlink" function).

Such unified use of TeX graphics and TeX programming through KETpic might be applicable to many other situations in math classrooms, and should enhance the possibility of TeX use in education.

Wind roses for TeX documents

Alan Wetmore

9:30am–10:05am, 24 October

In recent years a great many systems for including plots and graphics in TeX documents have been developed. Many varieties of scientific plots are directly supported by these packages. One style of plot which has not been available is a wind rose: describing the probability of wind speed and direction with a stylized polar plot. This report will describe a set of macros for Ti*k*Z for preparing wind rose plots.

Plots in LaTeX: Gnuplot, Octave, make

Boris Veytsman

10:05am–10:40am, 24 October

Making scientific and engineering documents with complex plots may be difficult and time consuming. This is especially true if data updates require rebuilding of plots and documents. In this report a workflow based on integration of AllTeX, Gnuplot and Octave using Makefiles in a Unix-based environment is proposed and discussed in detail.

LaTeX and graphics

Aleksandra Hankus and Zofia Walczak

1:30pm–2:05pm, 24 October

There are an number of distinct ways of producing graphics each with advantages and disadvantages in terms of flexibility, device independence and ability to include arbitrary TeX text. We will present a short history of creating graphics starting with the *picture* environment provided by L. Lamport with the LaTeX 2.09 format.

We will show examples of diagrams, charts, chemical formulas, musical notation and more complicated three-dimensional graphics. We also show how complex graphics can be produced with Ti*k*Z.

LaTeX3: Using the layers

Frank Mittelbach

2:05pm–2:50pm, 24 October

In this talk we will briey present the architecture of LaTeX3 with its four conceptual layers

- Document Interface Layer
- Document Design Layer
- Typesetting Element Layer
- Programming Layer

We will then look in some detail at `xparse` — a LaTeX 2_ε-like user interface, as an example of the LaTeX3 Document Interface Layer, that can already now be used to provide extended functionality for existing LaTeX 2_ε documents and packages.

We conclude with a brief detour of `expl3` the foundation layer for LaTeX3 that provides the basis for all higher-level modules of LATEX3 but can also be usefully deployed to develop packages for LaTeX 2_ε.

The `expl3` language is by now in a stable state and gets more and more traction outside the LaTeX3 development work, which can be seen, for example, by its use in a growing number of answers on the question and answer portal `http://tex.stackexchange.com` and in the appearance of LaTeX 2_ε packages that use it for programming.

Development of TeXShop—the past and the future

Yusuke Terada

5:15pm–5:50pm, 24 October

TeXShop is a widely-used open source TeX editor and previewer for MacOS X. TeXShop is developed by Richard Koch, emeritus professor of mathematics at the University of Oregon, and many other worldwide contributors including me. Now it is localized for as many as 10 languages. While it has already sufficient functions for editing TeX documents, TeXShop is still being updated. In this presentation, I will give an outline of the design concept of TeXShop and some new features that have been added recently, especially for editing Japanese documents. In addition, I will show a vision of TeXShop for the future.

TeX Live for Android: Development and usage

Clerk Ma

5:50pm–6:25pm, 24 October

The Android system is one of the most popular embeded systems for mobile devices nowdays. The word *Android* was coined by Andy Rubin, the father of Android. The original functions of the Android were aimed at photography. But, with the growth of mobile phones, this has changed. Now, with Google's support, Android can runs anywhere.

The kernel of Android is based on Linux. So, any binary which compiled by ARM toolchains can theoretically run on the Android. But, Android's directory structure do not obey the FHS (Filesystem Hierarchy Standard). And, common users cannot access root permissions.

Programs for Android must packed into a simple file, such as `foo.apk`, and run under the Dalvik VM. The main programming language is Java, and the tools are provided by an SDK. But Google provides an NDK (Native Development Kit) for complex developing.

My project uses both the SDK and NDK. The Java part acts like a native Android app, and the rest of my project's task is porting TeX Live via the NDK.

Portability of TeX Live. The engines in TeX Live can split into three parts. Some engines are written primarily in Pascal, like TeX82, Aleph, pTeX and e(u)pTeX. Two engines, pdfTeX and XeTeX, are written in Pascal with significant additions in C or C++. One engine, LuaTeX, is fully written in C/C++.

TeX Live provided a set of tools to translate Pascal to C, so any engines' code can be translated to C when building the binaries of TeX Live. Thus, TeX Live runs on essentially any normal Unix or Windows platform.. Building TeX Live on Linux is very simple, based on the GNU autotools. However, porting TeX Live to Android is not an easy task because Android's C library is `bionic`, not a standard `libc`. So, I needed to change some of the source.

Another cause of problems is the RTTI and exception support of the NDK's GCC.

Native app as installer and terminal emulator. In 2012, when I start my project, I only provided binary files to the users. They needed to "root" their Android devices to run these program. This is not convenient for many users.

In 2013, I have developed a native app based on Android Terminal Emulator. This app is intended to eventually have three groups of functions: terminal emulator (done), TeX Live package manager (work in progress), and an editor for TeX (not implemented).

Showcase: XeTeX and font-caching; copy any text to get PDF output; managing binary file and packages.

TeX Live Manager's rare gems: User mode and multiple repository support

Norbert Preining

6:25pm–6:50pm, 24 October

The TeX Live Manager (`tlmgr`) is responsible for the management of a TeX Live installation. It can be used to search for packages, do the usual package management (install, update, remove, backup). In the last couple of releases, two more features are available that have been requested for a long time: *user mode* and *multiple repository support*.

In user mode, instead of managing the system tree's and installation, `tlmgr` can be used to manage an arbitrary `texmf` tree, for example the user's `TEXMFHOME` (which is also the default). Although full functionality is not possible in user mode, the basic operations of package installation, removal, and updates can be handled by any user without requiring write permission to any system trees.

Multiple repository support was introduced to allow for easy handling of additional repositories of TeX Live packages. A few of them have come into existence (`tlcontrib`, `tlcritical`, `tlptexlive`, Korean support, ...), but before now one had to update packages one by one from these repositories. With multiple repositories this has been made a bit more convenient.

We will give detailed explanation of the workings of these two features, including live demonstrations, and finish with some words of caution.

Tsukurimashou: a Japanese-language font meta-family

Matthew Skala

9:00am–9:35am, 26 October

METAFONT-based font projects for the Chinese, Japanese, and Korean (CJK) languages have been announced every few years since the early 1980s, even predating the current form of the METAFONT language. Except for a few non-parameterized conversions of fonts that originated in other formats, in 30 years every METAFONT CJK font has been abandoned at or before the 8-bit barrier of 256 kanji, nowhere near the thousands required for practical typesetting. In this presentation I describe the first project to break that barrier: Tsukurimashou (`http://tsukurimashou.sourceforge.jp/`), currently at over 1400 kanji (as well as kana, Latin, and Korean hangul) and steadily growing. I discuss technical and human challenges facing this kind of project, how to solve them, and spin-off technologies such as the `IDSgrep` kanji structural query system.

upTEX—Unicode version of pTEX with CJK extensions

Takuji Tanaka

9:35am–10:10am, 26 October

upTEX is a Unicode extension of ASCII's pTEX (a Japanese-localized TEX). It not only improves Japanese support, but also treats Chinese and Korean characters i.e., Hanzi, Kanji, Hanja/Kana/CJK symbols, and Hangul with Unicode. Moreover, it can process multilingual typesetting of original LATEX with `\inputenc{utf8}` and Babel (Latin, Cyrillic, Greek, etc.) by switching its `\kcatcode` tables. In this presentation, the main features of upTEX will be described.

A short history of TEX in China

Jie Su and Clerk Ma*

10:10am–10:45am, 26 October

The original TEX only supported an 8-bit input encoding, and only supported a limited number of characters in PK fonts. When using TEX to get kanji output in DVI files, we need to handle four issues: (1) encoding processing, (2) kanji font processing, (3) glue processing between kanji and other characters, (4) kinsoku and mojikumi processing.

First, encoding of CJK languages must contain more than thousands of kanjis. In mainland China, people use the GB/T encodings; in Taiwan, people use the Big5 encoding.

A font is the general mechanism for mapping code points to glyphs. During the 1990s, in mainland China, many kanji fonts for computers were designed without using PostScript technology. Some of these fonts are bitmap fonts. Two preprocessors, CCT and TY, can produce DVI by converting these Chinese bitmap fonts to PK font.

The glue processing is the first step to get breakable kanji text.

Kinsoku and mojikumi processing is the second step to get correct output of kanji text with punctuations in proper position.

1. Poor Man's TEX

The earliest way to process CJK languages was Poor Man's TEX. Several years later, the CJK package has inherited Poor Man's TEX's mechanism, and wraps it with a simple interface.

The first step of Poor Man's TEX is to switch the catcode of kanjis to 13 (active character). And then to define these kanjis as macros which contain a `\char` command and a font command to produce a glyph in kanji fonts.

Without any extension to METAFONT, Poor Man's TEX only split a big kanji font into several small fonts to avoid the limitations of PK format.

2. CCT and TY: Two preprocessors

These preprocessors appeared in the 1990s. At that time, computers in China often ran a DOS system. These preprocessors' mechanism is similar to that of Poor Man's TEX, but modified for the tradition of Chinese typography.

In China, CCT has been accepted by the Chinese Mathematical Society (CMS). Most magzines are produced by CCT.

3. cwTEX and chiTEX: Another two prepocessors in Taiwan

These preprocessors are used in Taiwan, also using the Poor Man's TEX mechanism. The cwTEX developers have provided some patches for ConTEXt MkII to get kanji output.

4. PUTEX: An extension of TEX3

The PUTEX project was started in 1997 and stopped in 2004. The project was developed by Chey Woei Tsay. The PU in PUTEX is the abbreviation of Providence University in Taiwan.

Compared with the preprocessors above, PUTEX provided an intelligent way to processing kanjis.

PUTEX can handle both GB/T and Big5 encodings natively. On processing a font, PUTEX have patched one kanji font to an alphabetic font defined by `\font`. Put another way, PUTEX provided a fallback font mechanism.

Regarding kinsoku and mojikumi processing, PUTEX provided lots of primitives. In pTEX, some processing of mojikumi is defined by a JFM file. The different method of PUTEX's mojikumi processing is caused by the complex punctuation system in China.

The PUTEX project have also provided a patched version of makeindex: `puidx`. Traditionally, index sorting in China is more complex than in alphabetic languages. The sorting method can be: (1) by phonetic order, (2) by Bopomofo order, (3) by stroke count order, (4) by radical/stroke order. The `puidx` program can do sorting by phonetic order and Bopomofo order.

The PUTEX project also provided a patched version of `dvipdfmx`: `cdi2pdf`.

The general design of PUTEX is comparable to that of pTEX. Although these two engines have some differences in detail, they share some common concepts.

A case study: Typesetting old documents of Japan

Ken Nakano and Hajime Kobayashi*

11:10am–11:45am, 26 October

Shiryo Hensan-jo (the Historiographical Institute, HI), the University of Tokyo, is a major center of Japanese historical research. The HI makes historical sources available through its library, publications and databases.

Livretech helped them develop the typesetting system and databases for the old documents of Japan. These documents have more difficulties than ordinary Japanese books:

- many varieties of characters such as *seiji* (proper/correct characters) and sanskrit;
- various editorial notes such as headnote, one-liner note, inline note, line spacing note before line and after line;
- page-breakable family tree.

In this presentation, I will talk about how we approached these topics.

A case study on TEX's superior power: Giving different colors to building blocks of Korean syllables

Jin-Hwan Cho

11:45am–0:20pm, 26 October

In 2007 Dave Walden, the instigator and primary interviewer of TUG's Interview Corner, tossed a tricky question at me. "One of the concerns of many people in the TEX world is that TEX is relatively unknown in the larger worlds of typesetting and word processing compared with commercial programs such as Adobe's InDesign and Microsoft Word. How do you see the future of TEX when it comes to Asian languages?" Since then, it has been my mission to find a wonderful answer, that is, a TEX product which other programs cannot reproduce.

Unicode contains 11,172 modern Korean syllables all of which are composed by only 24 building blocks. In this talk, I will show an interesting TEX example containing a large number of Korean syllables each of which is grouped by building blocks of different colors. Nobody, of course, will try to reproduce the example with other commercial programs.

The multibibliography package

Michael Cohen, Yannis Haralambous, and Boris Veytsman**

1:20pm–1:55pm, 26 October

Conventional standards for bibliography styles entail a forced choice between index and name–year citations and corresponding references. We reject this false dichotomy, and describe a multibibliography, comprising alphabetic, sequenced, and also chronological orderings of references. An extended inline citation format is presented which integrates such heterogeneous styles, and is useful even without separate bibliographies. Richly hyperlinked for electronic browsing, the citations are articulated to select particular bibliographies, and the bibliographies are cross-referenced through their labels, linking among them.

Typesetting and Layout in Multiple Directions

John Plaice

2:30pm–3:05pm, 26 October

I propose a new, general way of looking at typesetting and layout in multiple directions. It subsumes the left-to-right and right-to-left horizontal writing used in most of the world, as well as the vertical writing used in East Asia. The generality allows the development of layout schemes for situations when several writing directions appear on the same page.

The key to the approach is that managing multidirectional text requires a separation of writing style from box direction. It turns out that there are only three different kinds of writing style, and eight kinds of directional box, and that simple rules can be used to define how these different writing styles may appear in different kinds of box.

Making mathematical content accessible using Tagged PDF and LaTeX

Ross Moore

3:25pm–4:00pm, 26 October

'Tagged PDF' (more specifically PDF/UA) is the method developed by Adobe to allow the Web Content Accessibility Guidelines (WCAG 1.0 and WCAG 2.0) to be satisfied within PDF documents. In this talk I will show the latest developments on using an extended version of pdfTeX to allow Tagged PDF documents to be produced, satisfying both PDF/A (archivability) and PDF/UA (Universal Accessibility).

I'll show examples which include quite complicated mathematical expressions, fully tagged with MathML, which can be 'Read Aloud' in Adobe's Acrobat and free Reader software. These will include 'real-world' documents containing such features as top-matter, nested list environments, logos, watermarks and other pagination artifacts, tabular material within mathematics, and some support of colour and text-styling. A special math-indexing feature has been developed, which allows the result of processing by external programs to be identified and reused in successive LaTeX runs. This indexing feature leads to significant time savings when developing a full document over many processing runs.

How we move(d) on with math

Hans Hagen

4:00pm–4:35pm, 26 October

Given the amount of time I spend on LuaTeX and ConTeXt I occasionally ask myself if it really makes sense to do that. The answer to that question is determined by several factors. Probably the most important factor is the userbase: what are their demands, how do they like to code, what control do they want, and therefore, where can these tools be of help? Another factor is relevance: can this combinations do certain things better than other tools? One area that has always drawn users is math typesetting. So, how up to date is TeX in that respect? Can we still claim victory there? Did we evolve well? Can we survive?

The XΥMTEX system for publishing interdisciplinary chemistry/mathematics books

Shinsaku Fujita
4:35pm–5:10pm, 26 October

The present version of the XΥMTEX system for drawing chemical structural formulas supports a AllTEX-compatible mode (based on the LATEX picture environment and the `epic` package), a PostScript-compatible mode (based on the PSTricks package), as well as a PDF-compatible mode (based on the PGF/TikZ package).

XΥMTEX is useful to generate directly-printable PDF manuscripts for publishing interdisciplinary books between chemistry and mathematics. Thereby, I have published several books by combining the chemical capabilities of XΥMTEX with mathematical ones of AllTEX, e.g., S. Fujita, *Organic Chemistry of Photography*, Springer-Verlag (2004); S. Fujita, *Diagrammatical Approach to Molecular Symmetry and Enumeration of Stereoisomers*, Univ. Kragujevac (2007); S. Fujita, *Combinatorial Enumeration of Graphs, Three-Dimensional Structures, and Chemical Compounds*, Univ. Kragujevac (2013, in press).

I will show how the XΥMTEXsystem has changed my style of writing manuscripts by referring to my monograph before the development of the XΥMTEX system: S. Fujita, *Symmetry and Combinatorial Enumeration in Chemistry*, Springer-Verlag (1991).

Distributing TEX and friends: methods, pitfalls, advice

Norbert Preining
5:10pm–5:45pm, 26 October

The TEX environment has grown slowly but steadily to a huge collection of programs, fonts, macro packages, support packages. TEX Live currently ships about 2 GB in more than 2000 different TEX Live "packages". As teTEX stopped being developed and supported several years ago, TEX Live has become the main TEX distribution on Unix, including MacOS X (MacTEX is exactly TEX Live plus a few Mac-specific additions); it is also gaining on Windows (where MiKTEX is still strong).

Integrating TEX Live into any full operating system distribution is a nontrivial task due to the large number of post installation tasks that have to be performed. Although over the last years the quality of packages has improved, the TEX Live development list still often gets bug reports that stem from incorrect packaging.

This talk gives an overview of the structure of TEX Live and a list of important and special configuration files. Furthermore, based on the experience of packaging TEX Live over many years, we will give some advice and examples on best practices. The talk is not targeted specifically for Debian, but at any distribution that redistributes TEX Live in one way or another.

Preprints

Math new style: are you better off?
Hans Hagen

Typesetting and Layout in Multiple Directions — Outline
John Plaice

Tsukurimashou: a Japanese-language font meta-family
Matthew Skala

The XyMTEX System for Publishing Interdisciplinary Chemistry/Mathematics Books
Shinsaku Fujita

The Multibibliography Package
Michael Cohen, Yannis Haralambous, and Boris Veytsman

1 Math new style: are we better off?

1.1 Introduction

In this article I will summarize the state of upgrading math support in ConTEXt per mid 2013 in the perspective of demand, usability, font development and LuaTEX. There will be some examples, but don't consider this a manual: there are enough articles in the `mkiv`, `hybrid` and `about` series about specific topics; after all, we started with this many years ago. Where possible I will draw some conclusions with respect to the engine. Some comments might sound like criticism, but you should keep in mind that I wouldn't spend so much time on TEX if I would not like it that much. It's just that the environment wherein TEX is and can be used is not always as perfect as one likes it to be, i.e. bad habits and decisions once made can be pretty persistent and haunt us forever. I'm not referring to TEX the language and program here, but more to its use in scientific publishing: in an early stage standards were set and habits were nurtured which meant that to some extent the coding resembles the early days of computing and the look and feel got frozen in time, in spite of developments in coding and evolving typographic needs. I think that the community has missed some opportunities to influence and improve matters which means that we're stuck with suboptimal situations and, although they are an improvement, UniCode math and OpenType math have their flaws.

This is not a manual. Some aspects will be explained with examples, others are just mentioned. I've written down enough details in the documents that describe the history of LuaTEX and MkIV and dedicated manuals and repeating myself makes not much sense. Even if you think that I talk nonsense, some of the examples might set you thinking. This article was written for the TUG 2013 conference in Japan. Many thanks to Barbara Beeton for proofreading and providing feedback.

1.2 Some basic questions

Is there still a need for a program like TEX? Those who typeset math will argue that there is. After all, one of the reasons why TEX showed up is typesetting math. In this perspective we should ask ourselves a few questions:

- Is TEX still the most adequate tool?
- Does it make sense to invest in better machinery?
- Have we learned from the past and improved matters?
- What drives development and choices to be made?

The first question is not that easy to answer, unless you see proof in the fact that TEX is still popular for typesetting a wide range of complex content (with critical editions being among the most complex). Indeed the program still attracts new users and developers. But we need to be realistic. First of all, there is some bias involved: if you have used a tool for many years, it becomes the one and only and best tool. But that doesn't necessarily make it the best tool for everyone.

In this internet world finding a few thousand fellow users gives the impression that there is a wide audience but there can be of course thousandfold more users of other systems that don't fall into your scope. This is fine: I always wonder why there is not more diversity; for instance, we have only a few operating systems to choose from, and in communities around computer languages there is a tendency to evangelize (sometimes quite extreme). We should also take into account that a small audience can have a large impact so size doesn't matter much.

As TEX is still popular among mathematicians, we can assume that it hasn't lost its charm yet and often it is their only option. We have a somewhat curious situation that scientific publishers still want to receive TEX documents —a demand that is not much different from organizations demanding MS Word documents— but at the same time don't care too much about TEX at all. Their involvement in user groups has started degrading long ago, compared to their profits; they don't invest in development; they are mostly profit driven, i.e. those who submit their articles don't even own their sources any more, etc.

On the other hand, we have users who make their own books (self-publishing) and who go, certainly in coding and style, beyond what publishers do: they want to use all kinds of fonts (and mixtures), color, nicely integrated graphics, more interesting layouts, experiment with alternative presentations. But especially for documents that contain math that also brings a price: you have to spend more time on thinking about presenting the content and coding of the source. This all means that if we look at the user side, alternative input is an option, especially if they want to publish on different media. I know that there are ConTEXt users who make documents (or articles) with ConTEXt, using whatever coding suits best, and do some conversion when it has to be submitted to a journal. Personally I think that the lack of interest of (commercial) publishers, and their rather minimal role in development, no longer qualifies them to come up with requirements for the input, if only because in the end all gets redone anyway (in Far Far Away).

It means that, as long as TEX is feasible, we are relatively free to move on and experiment with alternative input. Therefore the other two questions become relevant. The TEX engines are adapted to new font technology and a couple

of math fonts are being developed (funded by the user groups). Although the TEX community didn't take the lead in math font technology we are catching up. At the same time we're investing much time in new tools, but given the fact that much math is produced for publishers it doesn't get much exposure. Scientific publishing is quite traditional and like other publishing lags behind and eventually will disappear in its current form. It could happen that one morning we find out that all that 'publishers want it this or that way' gets replaced by ways of publishing where authors do all themselves. A publisher (or his supplier) can keep using a 20-year old TEX ecosystem without problems and no one will notice, but users can go on and come up with more modern designs and output formats and in that perspective the availability of modern engines and fonts is good. I've said it before: for CONTEXT user demand drives development.

In the next sections I will focus on different aspects of math and how we went from MKII to MKIV. I will also discuss some (pending) issues. For each aspect I will try to answer the third question: did matters improve and if not, and how do we cope with it (in CONTEXT).

1.3 The math script

All math starts with symbols and/or characters that have some symbolic meaning and in TEX speak this can be entered in a rather natural way:

```
$ y = 2x + b $
```

In order to let TEX know it's math (the equivalent of) two dollar signs are used as triggers. The output of this input is: $y = 2x + b$. But not all is that simple, for instance if we want to square the x, we need to use a superscript signal:

```
$ y = x^2 + ax + b $
```

The ^ symbol results in a smaller 2 raised after the x as in $y = x^2 + ax + b$. Ok, this ^ and its cousin _ are well known conventions so we stick to this kind of input.

A next level of complexity introduces special commands, for instance a command that will wrap its argument in a square root symbol: $y = \sqrt{x^2 + ax + b}$.

```
$ y = \sqrt { x^2 + ax + b } $
```

It is no big deal to avoid the backslash and use this kind of coding:

```
\asciimath { y = sqrt ( x^2 + ax + b ) }
```

In fact, we have been supporting scientific calculator input for over a decade in projects where relatively simple math had to be typeset. In one of our longest-running math related projects the input went from TEX, to content MATHML to OPENMATH and via presentation MATHML ended up as a combination of some kind of encoding that web browsers can deal with. This brings us to reality: it's web technology that drives (and will drive math) coding. Unfortunately content driven coding (like content MATHML) does not seem to be the winner here, even if it renders easier and is more robust.

Later I will discuss fences, like parentheses. Take this dummy formula:

```
$ (x + 1) / a = (x - 1) / b $
```

In a sequential (inline) rendering this will come out okay. A more display mode friendly variant can be:

```
$ \frac{x + 1}{a} = \frac{x - 1}{b} $
```

which in pure TEX would have been:

```
$ {x + 1} \over {a} = {x - 1} \over {b} $
```

The main difference between these two ways of coding is that in the second (plain) variant the parser doesn't know in advance what it is dealing with. There are a few cases in TEX where this kind of parsing is needed and it complicates not only the parser but also is not too handy at the macro level. This is why the `\frac` macro is often used instead. In LUATEX we didn't dare to get rid of `\over` and friends, even if we're sure they are not used that often by users.

In inline or in more complex display math, the use of fences is quite normal.

```
$ ( \frac{x + 1}{a}  + 1 )^2  = \frac{x - 1}{b} $
```

Here we have a problem. The parentheses don't come out well.

$(\frac{x+1}{a} + 1)^2 = \frac{x-1}{b}$

We have to do this:

```
$ \left( \frac{x + 1}{a}  + 1 \right)^2  = \frac{x - 1}{b} $
```

in order to get:

3

$\left(\frac{x+1}{a}+1\right)^2=\frac{x-1}{b}$

Doing that \left-\right trick automatically is hard, although in MathML, where we have to interpret operators anyway it is somewhat easier. The biggest issue here is that these two directives need to be paired. In ε-TeX a \middle primitive was added to provide a way to have bars adapt their height to the surroundings. Interesting is that where at the character level a (has a math property open and) has close. The bar, as we will see later, can also act as separator but this property does not exist. Because properties (classes in TeX speak) determine spacing we have a problem here. So far we didn't extend the repertoire of properties in LuaTeX to suit our needs (although in ConTeXt we do have more properties).

If you are a TeX user typesetting math, you can without doubt come up with more cases of source coding that have the potential of introducing complexities. But you will also have noticed that in most cases TeX does a pretty good job on rendering math out of the box. And macro packages can provide additional constructs that help to hide the details of fine tuning (because there is a lot that *can* be fine tuned).

In TeX there are a couple of special cases that we can reconsider in the perspective of (for instance) faster machines. Normally a macro cannot have a \par in one of its arguments. By defining them as \long this limitation goes away. This default limitation was handy in times when a run was relatively slow and grabbing a whole document source as argument due to a missing brace had a price. Nowadays this is no real issue which is why in LuaTeX we can disable \long which indeed we do in ConTeXt. On the agenda is to also permit \par in a math formula, as currently TeX complains loudly. Permitting a bit more spacy formula definitions (by using empty lines) would be a good thing.

Another catch is that in traditional TeX math characters cannot be used outside math. That restriction has been lifted. Of course users need to be aware of the fact that a mix of math and text symbols can be visually incompatible.

In the examples we used ^ and _ and in math mode these have special meanings. Traditionally in text mode they trigger an error message. In ConTeXt MkIV we have made these characters regular characters but in math mode they still behave as expected.[1] In a similar fashion the & is an ampersand and when you enable \asciimode the dollar and percent signs also become regular.[2] In LuaTeX we have introduced primitives for all characters (or more precisely: catcodes) that TeX uses for special purposes like opening and closing math mode, scripts, table alignment, etc.

In projects that involve xml we use MathML. In TeX many characters can be inserted using commands that are tuned for some purpose. The same character can be associated with several commands. In MathML entities and UniCode characters are used instead of commands. Interesting is that whenever we get math coded that way, there is a good chance that the coding is inconsistent. Of course there are ways in MathML to make sure that a character gets interpreted in the right way. For instance, the mfenced element drives the process of (matching) parenthesis, brackets, etc. and a renderer can use this property to make sure these symbols stretch vertically when needed. However, using mo in an mrow for a fence is also an option, but that demands some more (fuzzy) analysis. I will not go into details here, but some of the more obscure options and flags in ConTeXt relate to overcoming issues with such cases.

I have no experience with how MS Word handles math input, apart from seeing some demos. But I know that there is some input parsing involved that is a mixture between TeX and analysis. Just as word processing has driven math font technology it might be that at some point users expect more clever processing of input. To a large extent TeX users already expect that. Where till now TeX could inspire the way word processers do math, word processors can inspire TeXies way of inputting text.

So, we have MathML, which, in spite of being structured, is still providing users a lot of freedom. Then there are word processors, where mouse clicks and interpretation does the job. And of course we have TeX, with its familiar backslashes. Let us consider math, when seen in print, as a script to express the math language. And indeed, in OpenType, math is one of the official scripts although one where a rather specific kind of machinery is needed in order to get output.

I could show more complex math formulas but no matter what notation is used, coding will always be somewhat cumbersome and handywork. Math formula coding and typesetting remains a craft in itself and TeX notation will keep its place for a while. So, with that aspect settled we can continue to discuss rendering.

1.4 Alphabets

I have written about math alphabets before so let's keep it simple here. I think we can safely say that most math support mechanisms in macro packages are inspired by plain TeX. In traditional TeX we have fonts with a limited number of glyphs and an eight-bit engine, so in order to get the thousands of possible characters mapped onto glyphs the right one has to be picked from some font. In addition to characters that you find in UniCode, there are also variants, additional sizes and bits and pieces that are used in constructing large characters, so in practice a math font is quite large. But it is unlikely that we will ever run into a situation where fonts pose limits.

1 In an intermediate version \nonknuthmode and \donknuthmode controlled this.

2 Double percent signs act as comments then which is comparable to comments in some programming languages.

The easiest way is of course a direct mapping: an ‘a’ entered in math mode becomes an ‘a’ simply because the current font at that time has an italic shape in the slot referenced by the character. If we want a bold shape instead, we can switch to another font and still input an ‘a’. The 16 families available are normally enough for the alphabets that we need. Because symbols can be collected in any font, they are normally accessed by name, like `\oplus` or ⊕.

In UniCode math the math italic ‘a’ has slot `U+1D44E` and directly entering this character in a UniCode aware TeX engine also has to give that ‘a’. In fact, it is the only official way to get that character and the fact that we can enter the traditional ascii characters and get an italic shape is a side effect of the macro package, for instance the way it defines math fonts and families.[3]

Before we move on, let’s stress a limitation in UniCode with respect to math alphabets. It has always been a principle of UniCode committees to never duplicate entries. So, thanks to the availability of some characters in traditional (font) encodings, we ended up with some symbols that are used for math in the older regions of UniCode. As a consequence some alphabets have gaps. The only real reason I can come up with for accepting these gaps is that old documents using these symbols would be not compatible with gapfull UniCode math but I could argue that a document that uses those old codepoints uses commands (and needs some special fonts) to get the other symbols anyway, so it’s unlikely to be a real math document. On the other hand, once we start using UniCode math we could benefit from gapless alphabets simply because otherwise each application would have to deal with the exceptions. One can come up with arguments like “just use this or that library” but that assumes persistence, and also forces everyone to use the same approach. In fact, if we hide behind a library we could as well have hidden the vectors (alphabets) as well. But as they are exposed, the gaps stand out as an anomaly. Let’s illustrate this with an example. Say that we load the TeXGyre Pagella math font and call up a few characters:

```
\definefont[mathdemo][file:texgyrepagellamath*mathematics]
\mathdemo \char"0211C \char"1D507 \char"1D515
```

The UniCode fraktur math alphabet is continuous but the ‘MATHEMATICAL FRAKTUR CAPITAL R’ is missing as we already have the `BLACK-LETTER CAPITAL R` instead. So, this is why we only see two characters show up. It means that in the input we cannot have a `U+1D515`.

ℜ𝔇

Of course we can cheat and fill in the gap:

```
\definefontfeature
  [mymathematics]
  [mathematics]
  [mathgaps=yes]
```

This feature will help us cheat:

```
\definefont[mathdemo][file:texgyrepagellamath*mymathematics]
\mathdemo \char"0211C \char"1D507 \char"1D515
```

This time we can use the character. I wonder what would happen if the TeX community would simply state that slot `U+1D515` is valid. I bet that math related applications would support it, as they also support more obscure properties of TeX input encoding.

ℜ𝔇ℜ

If you still wonder why I bother about this, here is a practical example. The SciTE editor that I use is rather flexible and permits me to implement advanced lexers for ConTeXt (and especially hybrid usage). It also permits to hook in Lua code and that way the editor can (within bounds) be extended. As an example I’ve added some button bars that permit entering math alphabets. Of course the appearance depends on the font used but operating systems tend to consult multiple fonts when the core font of the editor doesn’t provide a glyph.

Here I show a small portion of the stripe with buttons that inject the shown characters. What happens in the rendering is that first the used font is consulted and that one has a couple of ‘BLACK LETTER CAPITAL’s so they get used. The

[3] Our experience is that even when for instance MathML permits coding of math in xml, copy editors have no problem with abusing regular italic font switches to simulate math. This can result is a weird mix of math rendering.

others are ‘MATHEMATICAL FRAKTUR CAPITAL’s and since the font is not a math font the renderer takes them from (in this case) Cambria Math, which is why they look so different, especially in proportion. Of course we could start out with Cambria but it has no monospace (which I want for editing) and is a less complete text font, so we have a chicken–egg problem here. It is one reason why as part of the math font project we extend the Dejavu Sans Mono with proper (consistent) math symbols. Anyhow, it illustrates why gaps are kind of evil from the application point of view.

gap	**char**	**meant**	**unicode**	**used**
U+1D455	ℎ	MATHEMATICAL ITALIC SMALL H	U+0210E	PLANCK CONSTANT
U+1D49D	ℬ	MATHEMATICAL SCRIPT CAPITAL B	U+0212C	SCRIPT CAPITAL B
U+1D4A0	ℰ	MATHEMATICAL SCRIPT CAPITAL E	U+02130	SCRIPT CAPITAL E
U+1D4A1	ℱ	MATHEMATICAL SCRIPT CAPITAL F	U+02131	SCRIPT CAPITAL F
U+1D4A3	ℋ	MATHEMATICAL SCRIPT CAPITAL H	U+0210B	SCRIPT CAPITAL H
U+1D4A4	ℐ	MATHEMATICAL SCRIPT CAPITAL I	U+02110	SCRIPT CAPITAL I
U+1D4A7	ℒ	MATHEMATICAL SCRIPT CAPITAL L	U+02112	SCRIPT CAPITAL L
U+1D4A8	ℳ	MATHEMATICAL SCRIPT CAPITAL M	U+02133	SCRIPT CAPITAL M
U+1D4AD	ℛ	MATHEMATICAL SCRIPT CAPITAL R	U+0211B	SCRIPT CAPITAL R
U+1D4BA	ℯ	MATHEMATICAL SCRIPT SMALL E	U+0212F	SCRIPT SMALL E
U+1D4BC	ℊ	MATHEMATICAL SCRIPT SMALL G	U+0210A	SCRIPT SMALL G
U+1D4C4	ℴ	MATHEMATICAL SCRIPT SMALL O	U+02134	SCRIPT SMALL O
U+1D506	ℭ	MATHEMATICAL FRAKTUR CAPITAL C	U+0212D	BLACK-LETTER CAPITAL C
U+1D50B	ℌ	MATHEMATICAL FRAKTUR CAPITAL H	U+0210C	BLACK-LETTER CAPITAL H
U+1D50C	ℑ	MATHEMATICAL FRAKTUR CAPITAL I	U+02111	BLACK-LETTER CAPITAL I
U+1D515	ℜ	MATHEMATICAL FRAKTUR CAPITAL R	U+0211C	BLACK-LETTER CAPITAL R
U+1D51D	ℨ	MATHEMATICAL FRAKTUR CAPITAL Z	U+02128	BLACK-LETTER CAPITAL Z
U+1D53A	ℂ	MATHEMATICAL DOUBLE-STRUCK CAPITAL C	U+02102	DOUBLE-STRUCK CAPITAL C
U+1D53F	ℍ	MATHEMATICAL DOUBLE-STRUCK CAPITAL H	U+0210D	DOUBLE-STRUCK CAPITAL H
U+1D545	ℕ	MATHEMATICAL DOUBLE-STRUCK CAPITAL N	U+02115	DOUBLE-STRUCK CAPITAL N
U+1D547	ℙ	MATHEMATICAL DOUBLE-STRUCK CAPITAL P	U+02119	DOUBLE-STRUCK CAPITAL P
U+1D548	ℚ	MATHEMATICAL DOUBLE-STRUCK CAPITAL Q	U+0211A	DOUBLE-STRUCK CAPITAL Q
U+1D549	ℝ	MATHEMATICAL DOUBLE-STRUCK CAPITAL R	U+0211D	DOUBLE-STRUCK CAPITAL R
U+1D551	ℤ	MATHEMATICAL DOUBLE-STRUCK CAPITAL Z	U+02124	DOUBLE-STRUCK CAPITAL Z

Barbara Beeton told me that, although it took some convincing arguments in the discussions about math in UNICODE, we have at least one hole less than to be expected: slot `U+1D4C1` has not been seen as already covered by `U+02113`. So is there really this distinction between a `MATHEMATICAL SCRIPT SMALL L` and `SCRIPT SMALL L` (usually `\ell` in macro packages? Indeed there is, although at the time of this writing interestingly Latin Modern fonts lacked the mathematical one (which in ConTeXt math mode normally results in an upright drop–in). Such details become important when math is edited by someone not familiar with the distinction between a variable (or whatever) represented by a script shape and the length operator. There seems not to be agreement by font designers about the shapes being upright or italic, so some confusion will remain, although this does not matter as long as within the font they differ.

font	**U+1D4C1**	**U+02113**
latin modern		ℓ
stix/xits	𝓁	ℓ
bonum	𝓁	ℓ
termes	𝓁	ℓ
pagella	𝓁	ℓ
lucida	𝓁	ℓ

As math uses greek and because greek was already present in UNICODE when math was recognized as script and got its entries, you can imagine that there are some issues there too, but let us move on to using alphabets.

In addition to a one–to–one mapping from a font slot onto a glyph, you can assign properties to characters that map them onto a slot in some family (which itself relates to a font). This means that in a traditional approach you can choose among two methods:

- You define several fonts (or instances of the same font) where the positions of regular characters point to the relevant shape. So, when an italic family is active the related font maps character `U+61` as well as `U+1D44E` to the same italic shape ‘*a*’. A switch from italic to bold italic is then a switch in family and in that family the `U+61` as well as `U+1D482` become bold italic ‘***a***’.
- You define just one font. The alphabet (uppercase, lowercase and sometimes digits and a few symbols) gets codes that point to the right shape. When we switch from italic to bold italic, these codes get reassigned.

The first method has some additional overhead in defining fonts (you can use copies but need to make sure that the regular ASCII slots are overloaded) but the switch from italic to bold italic is fast, while in the second variant there is less overhead in fonts but reassigning the codes with a style switch has some overhead (although in practice this overhead is can be neglected because not that many alphabet switches take place). In fact, many TEX users will probably stick to traditional approaches where verbose names are used and these can directly point to the right shape.

In ConTEXt, when we started with MkIV, we immediately decided to follow another approach. We only have one family and we assume UniCode math input. Ok, we do have a few more families, but these relate to a full bold math switch and right–to–left math. We cannot expect users to enter UniCode math, if only because support in editors is not that advanced, so we need to support the ASCII input method as well.

We have one family and don't redefine character codes, but set properties instead. We don't switch fonts, but properties. These properties (often a combination) translates into the remapping of a specific character in the input onto a UniCode math code point that then directly maps onto a shape. This approach is quite clean and efficient at the TEX end but carries quite a lot of overhead at the Lua end. So far users never complained about it, maybe because ConTEXt math support is rather optimized. Also, dealing with characters is only part of math typesetting and we have subsystems that use far more processing power.

Because math characters are organized in classes, we need to set them up. Because for several reasons we collect character properties in a database we also define these character properties in Lua. This means that the `math-*` files are relatively small. So we have much less code at the TEX end, but quite a lot at the Lua end. This assumes a well managed Lua subsystem because as soon as users start plugging in their code, we have to make sure that the core system still functions well. The amount of code involved in virtual math fonts is also relatively large but most of that is becoming sort of obsolete.

Relatively new in ConTEXt is the possibility in some mathematical constructs to configure the math style (text, script, etc.) and in some cases math classes can be influenced. Control over styles is somewhat more convenient in LuaTEX, because we can consult the current style in some cases. I expect more of this kind of control in ConTEXt, although most users probably never need it. These kinds of features are meant for users like Aditya Mahajan, who likes to explore such features and also takes advantage of the freedom to experiment with the look and feel of math.

The font code that relates to math is not the easiest to understand but this is because it has to deal with bold as well as bidirectional math in efficient ways. Because in ConTEXt we have additional sizes (`x`, `xx`, `a`, `b`, `c`, `d`, ...) we also have some delayed additional defining going on. This all might sound slower to set up but in the end we win some back by the fact that we have fewer fonts to load. The price that a ConTEXt user pays in terms of runtime is more influenced by the by now large sequence of math list manipulators than by loading a font.

An unfortunate shortcoming of UniCode math is that some alphabets have gaps. This is because characters can only end up once in the standard. Given the number of weird characters showing up in recent versions, I think this condition is somewhat over the top. It forces applications that deal with UniCode math to implement exceptions over and over again. In ConTEXt we assume no gaps and compensate for that.

There are several ways that characters can become glyphs. An 'a' can become an italic, bold, bold italic but also end up sans serif or monospace. Because there are several artistic interpretations possible, some fonts provide a so-called alternate. In the case of for instance greek we can also distinguish upright or slanted (italic). A less well known transformation is variants driven by UniCode modified directives. If we forget about bidirectional math and full bold (heavy) math we can (currently) identify 6 axes:

axis	**use**	**choices**
1	type	digits, lowercase & uppercase latin & greek, symbols
2	alphabet	regular, sans serif, monospace, blackboard, fraktur, script
3	style	upright, italic, bold, bolditalic
4	variant	alternative rendering provided by font
5	shape	unchanged, upright, italic
6	UniCode	alternative rendering driven by UniCode modifier

Apart from the last one, this is not new, but it is somewhat easier to support this consistently. It's one of the areas where UniCode shines, although the gaps in vectors are a bad thing. One thing that I decided early in the MkIV math development is that all should fit into the same model: it makes no sense to cripple a whole system because of a few exceptions.

Users expect their digits to be rendered upright and letters to be rendered with italic shapes, but use regular ASCII input. This means that we need to relocate the letters to the relevant alphabet in UniCode. In ConTEXt this happens as part of several analysis steps that more or less are the same as the axis mentioned. In addition there is collapsing, remapping, italic correction, boldening, checking, intercepting of special input, and more going on. Currently there are (depending on what gets enabled) some 10 to 15 manipulation passes over the list and there will be more.

7

So how does the situation compare to the old one? I think we can safely say that we're better off now and that LuaTEX behaves quite okay. There is not much that can be improved, apart from more complete fonts (especially bold). A nice bonus of LuaTEX is that math characters can be used in text mode as well (given that the current font provides them).

It will be clear that by following this route we moved far away from the MkII approach and the dependency on Lua has become rather large in this case. The benefit is that we have rather clean code with hardly any exceptions. It came at the price of lots of experiments and (re)coding but I think it pays off for users.

1.5 Bold

Bold is sort of special. There are bold symbols and some bold alphabets and that *is* basically what bold math is: just a different rendering. In a proper OpenType math fonts these bold characters are covered.

Section titles or captions are often typeset bolder and when they contain math all of it needs to be bolder too. So, a regular italic shape becomes a bold italic shape but a bold shape becomes heavy. This means that we need a full blown bold font for that purpose. And although some are on the agenda of the font team, often we need to fake it. This is seldom an issue as (at least in the documents that I deal with) section titles are not that loaded with math.

A proper implementation of such a mechanism involves two aspects: first there needs to be a complete bold math font with heavy bold included, and second the macro package must switch to bold math in a bold context. When no real bold font is available, some automatic mapping can take place, but that might give interpretation issues if bold is used in a formula. For the average highschool math that we render this is not an issue. Currently there are no full bold math fonts that have enough coverage. (The Xits font, derived from Stix, has a bold companion that does provide for instance bold radicals but lacks many bolder alphabets and symbols.)

```
\startimath
       \sqrt{x^2\over 4x}  \qquad
  {\bf \sqrt{x^2\over 4x}} \qquad
  {\mb \sqrt{x^2\over 4x}} \qquad
       \sqrt{x^2 + 4x}     \qquad
  {\bf \sqrt{x^2 + 4x}}    \qquad
  {\mb \sqrt{x^2 + 4x}}
\stopimath
```

This gives:

$\sqrt{\frac{x^2}{4x}} \qquad \sqrt{\frac{\mathbf{x^2}}{\mathbf{4x}}} \qquad \boldsymbol{\sqrt{\frac{x^2}{4x}}} \qquad \sqrt{x^2 + 4x} \qquad \sqrt{\mathbf{x^2 + 4x}} \qquad \boldsymbol{\sqrt{x^2 + 4x}}$

Here it is always a bit of a guess if bold extensibles are (already) supported so it's dangerous to go wild with full bold/heavy combinations unless you check carefully what results you get. Another aspect you need to be aware of is that there is an extensive fallback mechanism present. When possible a proper alphabet will be used, but when one is not present there is a fallback on another. This ensures that we get at least something.

There is not much that an engine can do about it, apart from providing enough families to implement it. In a Type1 universe indeed we need lots of families already so the traditional 16-family pool is drained soon. In LuaTEX we can have 256 families which means that additional Type1 bases family sets are no issue any longer. But as in MkIV we no longer follow that route, bold math can be set up relatively easy, given that we have a bold font. If we don't have such a font, we have an intermediate mode where a bold font is simulated. Keep in mind that this always will need checking, at least as long as don't have complete enough bold fonts with heavy bold included.

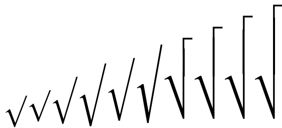
1.6 Radicals

In most cases a TEX user is not that aware of what happens in order to get a nicely wrapped up root on paper. In traditional TEX this is an interplay between rather special font properties and macros. In LuaTEX it has become a bit more simple because we introduced a primitive for it. Also, in OpenType fonts, the radical is provided in a somewhat more convenient way. In an OpenType math font there are some variables that control the rendering:

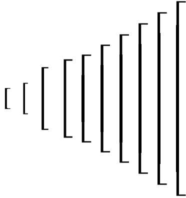
```
RadicalExtraAscender
RadicalRuleThickness
RadicalVerticalGap
RadicalDisplayStyleVerticalGap
```

The engine will use these to construct the symbol. The root symbols can grow in two dimensions: the left bit grows vertically but due to the fact that there is a slope involved it happens in steps using different symbols.

8

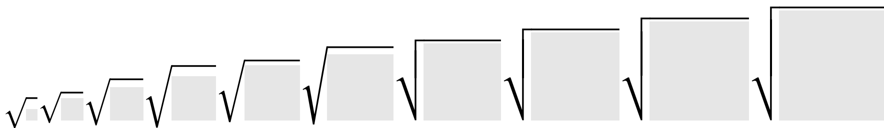


Compare this to for instance how a bracket grows:



The bracket is a so-called vertical extensible character. It grows in steps using different glyphs and when we run out of variants a last resort kicks in: a symbol gets constructed from three pieces, a top and bottom piece and in between a repeated middle segment. The root symbol is also vertically extensible but there the change to the stretched variant is visually rather distinct. This has a reason: the specification cannot deal with slopes. So, in order to stretch the last resort, as with the bracket, goes vertical and provides a middle segment.

The root can also grow horizontally; just watch this:



The font specification can handle vertical as well as horizontal extensibles but surprise: it cannot handle a combination. Maybe the reason is that there is only one such symbol: the radical. So, instead of expecting a symmetrical engine, an exception is made that is controlled by the mentioned variables. So, while we go upwards with a proper middle glyph, we go horizontal using a rule.

One can argue that the traditional TEX machinery is complex because it uses special font properties and macros, but once you start looking into the modern variant it becomes clear that although we can have a somewhat cleaner implementation, it still is a kludge. And, because rendering on paper no longer drives development it is not to be expected that this will change. The TEX community didn't come up with a better approach and there is no reason to believe that it will in the future.

One of the reasons for users to use TEX is control over the output: instead of some quick and dirty job authors can spend time on making their documents look the way they want. Even in these internet times with dynamic rendering, there is still a place for a more frozen rendering, explicitly driven by the author. But, that only makes sense when the author can influence the rendering, maybe even without bounds.

So, because in ConTEXt I really want to provide control, as one of the last components, math radicals were made configurable too. In fact, the code involved is not that complex because most was already in place. What is interesting is that when I rewrapped radicals once again I realized that instead of delegating something to the engine and font one could as well forget about it and do all in dedicated code. After all, what is a root symbol more that a variation of a framed bit of text. Here are some examples.

```
$
  y = \sqrt   { x^2 + ax + b } \quad
  y = \sqrt[2]{ x^2 + ax + b } \quad
  y = \sqrt[3]{ \frac{x^2 + ax + b }{c} }
$
```

By default this gets rendered as follows:

$y = \sqrt{x^2 + ax + b} \quad y = \sqrt[2]{x^2 + ax + b} \quad y = \sqrt[3]{\frac{x^2+ax+b}{c}}$

We can change the rendering alternative to one that permits some additional properties (like color):

```
\setupmathradical[sqrt][alternative=normal,color=darkblue]
```

This looks more or less the same:

$y = \sqrt{x^2 + ax + b} \quad y = \sqrt[2]{x^2 + ax + b} \quad y = \sqrt[3]{\frac{x^2+ax+b}{c}}$

We can go a step further and instead of a font use a symbol that adapts itself:

```
\setupmathradical
```

```
  [sqrt]
  [alternative=mp,
   color=darkgreen]
```

Now we get this:

$y = \sqrt{x^2 + ax + b} \quad y = \sqrt[2]{x^2 + ax + b} \quad y = \sqrt[3]{\frac{x^2+ax+b}{c}}$

Such a variant can be more subtle, as we not only can adapt the slope dynamically, but also add a nice finishing touch to the end of the horizontal line. Take this variant:

```
\startuniqueMPgraphic{math:radical:extra}
  draw
    math_radical_simple(OverlayWidth,OverlayHeight,OverlayDepth,OverlayOffset)
    withpen pencircle
      xscaled (2OverlayLineWidth)
      yscaled (3OverlayLineWidth/4)
      rotated 30
    dashed evenly
    withcolor OverlayLineColor ;
\stopuniqueMPgraphic
```

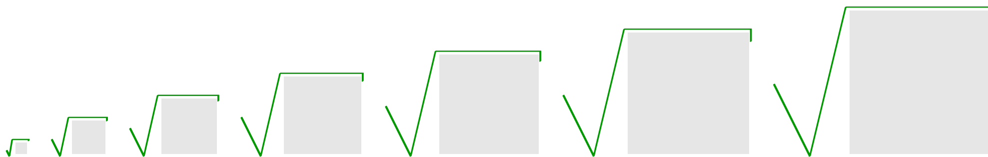
We hook this graphic into the macro:

```
\setupmathradical
  [sqrt]
  [alternative=mp,
   mp=math:radical:extra,
   color=darkred]
```

And this time we see a dashed line:

$y = \sqrt{x^2 + ax + b} \quad y = \sqrt[2]{x^2 + ax + b} \quad y = \sqrt[3]{\frac{x^2+ax+b}{c}}$

Of course one can argue about esthetics but let's face it: much ends up in print, also by publishers, that doesn't look pretty at all, so I tend to provide the author the freedom to make what he or she likes most. If someone is willing to spend time on typesetting (using TEX), let's at least make it a pleasant experience.



Here we see the symbol adapt. We can think of alternative symbols, for instance the first part becomes wider dependent on the height, but this can be made less prominent. Depending on user input I will provide some more variants as it's relatively easy to implement.

Before I wrap up, let's see what exactly we have in stock deep down. Traditionally TEX provides a `\surd` command which is just the root symbol. Then there is a macro `\root..\of..` that wraps the last argument in a root and typesets a degree as well (of given). In ConTEXt we now provide this:

```
$\surd x \quad \surdradical x  \quad \rootradical{3}{x} \quad \sqrt[3]{x}$
```

I don't remember ever having used the `\surd` command, but this is what it renders:

$\surd x \quad \sqrt{x} \quad \sqrt[3]{x} \quad \sqrt[3]{x}$

Only the last command, `\sqrt` is a macro defined in one of the math modules, the others are automatically defined from the database:

```
[0x221A] = { -- there are a few more properties set
  unicodeslot = 0x221A,
  description = "SQUARE ROOT",
  adobename   = "radical",
  category    = "sm",
  mathspec    = {
    { class = "root",     name = "rootradical" },
    { class = "radical",  name = "surdradical" },
```

```
    { class = "ordinary", name = "surd"        },
  },
}
```

So we get the following definitions:

command	meaning	usage
`\surd`	`\Umathchar"0"00"00221A`	`\surd`
`\surdradical`	`\protected macro:->\Uradical "0 "221A`	`\surdradical{body}`
`\rootradical`	`\protected macro:->\Uroot "0 "221A`	`\rootradical{degree} {body}`

So, are we better off? Given that a font sticks to how Cambria does it, we only need a minimal amount of code to implement roots. This is definitely an improvement at the engine level. However, in the font there are no fundamental differences between the traditional and more modern approach, but we've lost the opportunity to make a proper two–dimensional extensible. Eventually the user won't care as long as the macro package wraps it all up in useable macros.

1.7 Primes

Another rather disturbing issue is with primes. A prime is an accent-like symbol that as a kind of superscript is attached to a variable or function. In good old TEX tradition this is entered as follows:

```
$ f'(x) $ and $ f''(x) $
```

which produces: $f'(x)$ and $f''(x)$. The upright quote symbols are never used for anything else than primes and magically get remapped onto a prime symbol. This might look trivial, but there are several aspects to deal with, especially when using traditional fonts. In the eight-bit `lmsy10` math symbol font, which is derived from the original `cmsy10` the prime symbol looks like this:

′

The bounding box is rather tight and the reason for this becomes clear when we put it alongside another character:

$x′$

The prime is not only pretty large, it also sits on the baseline. It means that in order to make it a real prime (basically an operator pointing back to the preceding symbol), we need to raise it. Of course we can define a `\prime` command that takes care of this, and indeed that is what happens in plain TEX and derived formats. The more direct `'` input is supported by making that character an active character in math mode. Active characters behave like commands and in this case the `\prime` command.

In the OPENTYPE latin modern fonts the prime (`U+2032`) looks like this:

x'

So here we have an already raised and also smaller prime symbol. And, because we also have double (`U+2033`) and triple primes (`U+2034`) a few more characters are available

x' x'' x'''

In the traditional approach these second and third order primes are built from the first order primes. And this introduces, in addition to the raising, another complexity: the `\prime` command has to look ahead and intercept future primes. And as there can also be a following raised symbol (or number) it needs to take a superscript trigger into account as well. So, let's look at some possible input:

`$f'(x)$`	$f'(x)$
`$f''(x)$`	$f''(x)$
`$f'''(x)$`	$f'''(x)$
`$f\prime ^2$`	f'^2
`$f\prime \prime ^2$`	f''^2
`$f\prime \prime \prime ^2$`	f'''^2
`$f'\prime '^2$`	f'''^2
`$f^'(x)$`	$f^{\prime}(x)$
`$f'^2$`	f'^2
`$f{\prime }^2$`	f'^2

Now imagine that you have this big prime character sitting on the baseline and you need to turn `'''` into a a triple prime, but don't want `^'` to be double raised, while on the other hand `^2` should be. This is of course doable with some macro juggling but how about supporting traditional fonts in combination with OPENTYPE, where the primes are already raised.

When we started with LUATEX and CONTEXT MKIV, one of the first decisions I made was to go UNICODE math and drop eight-bit. In order to compensate for the lack of fonts, a mechanism was provided to construct virtual UNICODE math

fonts, as a prelude to the lm/gyre OpenType math fonts. In the meantime we have these fonts and the virtual variants are only kept as historic reference and for further experiments.

As a starter I wrote a variant of the traditional ConTeXt `\prime` command that could recognize somehow if it was dealing with a Type1 or OpenType font. As a consequence it also had the traditional raise and look ahead mess on board. However, there was also some delegation to the Lua enhanced math support code, so the macro was not that complex. When the real OpenType math fonts showed up the macro was dropped and the virtual fonts were adapted to the raised-by-default situation, which in itself was somewhat complicated by the fact that a smaller symbol had to be used, i.e. some more information about the current set of defined math sizes has to be passed around.[4]

Anyhow, the current implementation is rather clean and supports collapsing of combinations rather well. There are four prime symbols but only three reverse prime symbols. If needed I can provide a virtual `REVERSED TRIPLE PRIME` if needed, but I guess it's not needed.

```
U+2032  PRIME                   ′
U+2033  DOUBLE PRIME            ″
U+2034  TRIPLE PRIME            ‴
U+2057  QUADRUPLE PRIME         ⁗
U+2035  REVERSED PRIME          ‵
U+2036  REVERSED DOUBLE PRIME   ‶
U+2037  REVERSED TRIPLE PRIME   ‷
```

Of course no one will use this ligature approach but I've learned to be prepared as it wouldn't be the first time when we encounter input that is cut and paste from someplace or clicked-till-it-looks-okay.

There is one big complication and that is that where in TeX there is only one big prime that gets raised and repeated in case of multiple primes, in OpenType the primes are already raised. They are in fact not supposed to be superscripted, as they are already. In plain TeX the prime is entered using an upright single quote and that one is made active: it is in fact a macro. That macro looks ahead and intercepts following primes as well as subscripts. In the end, a superscript (the prime) and optional subscripts are attached to the preceding symbol. If we want to benefit from the UniCode primes as well as support collapsing, such a macro quickly becomes messy. Therefore, in MkIV the optional subscript is handled in the collapser. We cheat a bit by relocating super- and subscripts and at the same time remap the primes to virtual characters that are smashed to a smaller height, lowered to the baseline, and eventually superscripted. Indeed, it sounds somewhat complex and it is. In a next version I will also provide ways to influence the size as one might want larger of smaller primes to show up. This is one case where the traditional TeX fonts have a benefit as the primes are superscriptable characters, but we have to admit that the UniCode and OpenType approach is conceptually more correct. The only way out of this is to have a primitive operation for primes just as we have for radicals but that also has some drawbacks. Eventually I might come up with a cleaner solution for this dilemma.

Let us summarize the situation and solution used in MkIV now:

- When (still) using the virtual UniCode math fonts, we construct a virtual glyph that has properties similar to proper OpenType math fonts.
- We collapse a sequence of primes into proper double and triple primes.
- We unraise primes so that users who (for some reason) superscript them (maybe because they still assume big ones sitting on the baseline) get the desired outcome.
- We accept mixtures of `'` and `\prime`.

We can do this because in ConTeXt MkIV we don't care too much about exact visual compatibility as long as we can make users happy with clean mechanisms. So, this is one of the situations where the new situation is better, thanks to on the one hand the way primes are provided in fonts, and on the other hand the enhanced math machinery in MkIV.

1.8 Accents

There are a few special character types in math and accents are one of them. Personally I think that the term accent is somewhat debatable but as they are symbols drawn on top of or below something we can stick to that description for the moment. In addition to some regular fixed width variants, we have adaptive versions: `\hat` as well as `\widehat` and more.

I have no clue if wider variants are needed but such a partial coverage definitely looks weird. So, as an escape users can kick in their own code. After all, who says that a user cannot come up with a new kind of math. The following example demonstrates how this is done:

[4] The actual solution for this qualifies as a dirty trick so we are not freed from tricks yet.

```
\startMPextensions
    vardef math_ornament_hat(expr w,h,d,o,l) text t =
        image (
            fill
                (w/2,10l) -- (w + o/2,o/2) --
                (w/2, 7l) -- (  - o/2,o/2) --
                cycle shifted (0,h-o) t ;
            setbounds
                currentpicture
            to
                unitsquare xysized(w,h) enlarged (o/2,0)
        )
    enddef ;
\stopMPextensions
```

This defines a hat-like symbol. Once the sources of the math font project are published I can imagine that an ambitious user defines a whole set of proper shapes. Next we define an adaptive instance:

```
\startuniqueMPgraphic{math:ornament:hat}
    draw
        math_ornament_hat(
            OverlayWidth,
            OverlayHeight,
            OverlayDepth,
            OverlayOffset,
            OverlayLineWidth
        )
    withpen
        pencircle
            xscaled (2OverlayLineWidth)
            yscaled (3OverlayLineWidth/4)
            rotated 30
    withcolor
        OverlayLineColor ;
\stopuniqueMPgraphic
```

Last we define a symbol:

```
\definemathornament [mathhat] [mp=math:ornament:hat,color=darkred]
```

And use it as `\mathhat{...}`:

Of course this completely bypasses the accent handler and in fact even writing the normal stepwise one is not that hard to do in macros. But, there is a built–in mechanism that helps us for those cases and it can even deal with font based stretched alternatives of which there are a few: curly braces, brackets and parentheses. The reason that these can stretch is that they don't have slopes and therefore can be constructed out of pieces: in the case of a curly brace we have 4 snippets: begin, end, middle and repeated rules, and in the case of braces and brackets 3 snippets will do. But, if we really want we can use METAPOST code similar to the code shown above to get a nicer outcome.

There are in good TEX tradition four accents that can also stretch horizontally: bar, brace, parenthesis and bracket. When using fonts such an accent looks like this:

$\overbrace{a+b+c+d} \quad \underbrace{a+b+c+d} \quad \overbrace{\underbrace{a+b+c+d}}$

this is coded like:

```
$ \overbrace{a+b+c+d} \quad \underbrace{a+b+c+d} \quad \doublebrace{a+b+c+d} $
```

As with radicals, for more fancy math you can plug in METAPOST variants. Of course this kind of rendering should fit into the layout of the document but I can imagine that for schoolbooks this makes sense.

```
\useMPlibrary[mat]

\setupmathstackers
  [vfenced]
```

```
[color=darkred,
 alternative=mp]
```

Applied in an example we get:

$\overbracket{a+b+c+d}$ $\underbracket{a+b+c+d}$ $\overbracket{\underbracket{a+b+c+d}}$

$\overparent{a+b+c+d}$ $\underparent{a+b+c+d}$ $\overparent{\underparent{a+b+c+d}}$

$\overbrace{a+b+c+d}$ $\underbrace{a+b+c+d}$ $\overbrace{\underbrace{a+b+c+d}}$

$\overline{a+b+c+d}$ $\underline{a+b+c+d}$ $\overline{\underline{a+b+c+d}}$

This kind of magic is partly possible because in LuaTeX (and therefore MkIV) we can control matters a bit better. And of course the fact that we have METAPOST embedded means that the impact of using graphics is not that large.

We used the term 'stackers' in the setup command so although these are officially accents, in ConTeXt we implement them as instances of a more generic mechanism: things stacked on top of each other. We will discuss these in the next section.

1.9 Stackers

In plain TeX and derived work you will find lots of arrow builders. In most cases we're talking of a combination of one or more single or double arrow heads combined with a rule. In any case it is something that is not so much font driven but macro magic. Optionally there can be text before and/or after as well as text above and/or below them. The later is for instance the case in chemistry. This text is either math or upright properly kerned and spaced non–mathematical text so we're talking of some mixed math and text usage. The size is normally somewhat smaller.

Arrows can also go on top or below regular math so in the end we end up with several cases:

- Something stretchable on top of or centered around the baseline, optionally with text above or below.
- Something stretchable on top of a running (piece of) text or math.
- Something stretchable below a running (piece of) text or math.
- Something stretchable on top as well as below a running (piece of) text or math.

These have in common that the symbol gets stretched. In fact the last three cases are quite similar to accents but in traditional TeX and its fonts arrows and alike never made it to accents. One reason is probably that because a macro language was available and because fonts were limited, it was rather easy to use rules to extend an arrowhead.

In ConTeXt this kind of vertically stacked stretchable material is implemented as stackers. In the chapter `mathstackers` of `about.pdf` you can read more about the details so here I stick to a short summary to illustrate what we're dealing with. Say that you want an arrow that stretches over a given width.

```
\hbox to 4cm{\leftarrowfill}
```

In traditional TeX with traditional fonts the definition of this arrow looks as follows:

```
\def\leftarrowfill {$
  \mathsurround=0pt
  \mathord{\mathchar"2190}
  \mkern-7mu
  \cleaders
   \hbox {$
     \mkern-2mu
     \mathchoice
      {\setbox0\hbox{$\displaystyle     -$}\ht0=0pt\dp0=0pt\box0}
      {\setbox0\hbox{$\textstyle        -$}\ht0=0pt\dp0=0pt\box0}
      {\setbox0\hbox{$\scriptstyle      -$}\ht0=0pt\dp0=0pt\box0}
      {\setbox0\hbox{$\scriptscriptstyle-$}\ht0=0pt\dp0=0pt\box0}
     \mkern-2mu
    $}
   \hfill
  \mkern-7mu
  \mathchoice
   {\setbox0\hbox{$\displaystyle     -$}\ht0=0pt\dp0=0pt\box0}
   {\setbox0\hbox{$\textstyle        -$}\ht0=0pt\dp0=0pt\box0}
   {\setbox0\hbox{$\scriptstyle      -$}\ht0=0pt\dp0=0pt\box0}
```

```
   {\setbox0\hbox{$\scriptscriptstyle-$}\ht0=0pt\dp0=0pt\box0}
$}
```

When using TYPE1 fonts we don't use a `\mathchar` but more something like this:

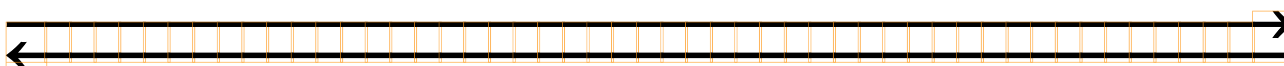
```
\leftarrow = \mathchardef\leftarrow="3220
```

What we see in this macro is a left arrow head at the start and as minus sign at the end. In between the `\cleaders` will take care of filling up the available hsize with more minus signs. The overlap is needed in order to avoid gaps due to rounding in the renderer and also obscures the rounded caps of the used minus sign.

The minus sign is used because it magically connects well to the arrow head. This is of course a property of the design but even then you can consider it a dirty trick. We don't specify a width here as this macro adapts itself to the current width due to the leader. But if we do know the width an easier approach becomes possible. Take this combination of a left and right arrow on top of each other:

```
\mathstylehbox{\Umathaccent\fam\zerocount"21C4{\hskip4cm}}
```

The `\mathstylehbox` macro is a CONTEXT helper. When we take a closer look at the result (scaled up a bit) we see again snippets being used:[5].



But this time the engine itself deals with the filling. Unfortunately for the accent approach to work we need to specify the width. Given how these arrows are used, this is no problem: because we often put text on top and/or below, we need to do some packaging and therefore know the dimensions, but a generic alternative would be nice. This is why for LUATEX we have on the low priority agenda:

```
\leaders"2190\hfill
```

or a similar primitive. This way we can let the engine do some work and keep macros simple. Normally `\leaders` delegate part of repeating to the backend but in the case of math it has to be part of constructing the formula because the extensible constructor has to be used.

If you've looked into the LUATEX manual you might have noticed that there is a new primitive that permits this:

```
\mathstylehbox{\Uoverdelimiter\fam"21C4{\hskip4cm}}
```

However, it is hardly useable for our purpose for several reasons. First of all, when the argument is narrower than the smallest possible delimiter both get left aligned, so the delimiter sticks out (this can be considered a bug). But also, the placement is influenced by a couple of parameters that we then need to force to zero values, which might interfere. Another property of this mechanism is that the style is influenced and so we need to mess more with that. These are enough reasons to ignore this extension for a while. Maybe at some point, when really needed, I will write a proper wrapper for this primitive.

When we started with MKIV we stuck with the leaders approach for a while if only because there was no real need to redefine the old macros. But after a while one starts wondering if this is still the way to go, especially when reimplementing the chemistry macros didn't lead to nicer looking code. Part of the problem was that putting two arrows on top of each other where each one goes into another direction gave issues due to the fact that we don't have the right snippets to do it nicely. A way out was to create virtual characters for combinations of begin and end snippets as well as middle pieces, construct a proper virtual extensible and use the LUATEX extensible constructor. Although we still have a character that gets built out of snippets, at least the begin and end snippet indicate that we have to do with one codepoint, contrary to two independent stacked arrows.

This was also the moment that I realized that it was somewhat weird that OPENTYPE math fonts didn't have that kind of support. After discussing this with Bogusław Jackowski of the math font project we decided that it made sense to add proper native extensibles to the upcoming math fonts. Of course I still had to support other math fonts but at least we had a conceptually clean example font now. So, from that moment on the implementation used extensibles when possible and falls back on the fake approach when needed.

In CONTEXT all these vertically stacked items are now handled by the math stacker subsystem, including a decent set of configuration options. As said, the symbols that need to stretch currently use the accent primitives which is okay but somewhat messy because that mechanism is hard to control (after all it wants to put stuff on top or below something). For (mostly) chemistry we can put text on top or below arrows and control offsets of the text as well as the axis of the arrows. We can use color and set the style. In addition there are constructs where there is text in the middle and arrows (or other symbols that need to adapt) on top or at the bottom.

---

[5] We cheat a bit here: as we use XITS in this document, and that font doesn't yet provide this magic we switch temporarily to the Pagella font

Many arrows come in sizes. For instance there are two sizes of right pointing arrows as well as stretched variants, and use as top and bottom accents.

`$\rightarrow \quad \char "2192$`	$\rightarrow \quad \rightarrow$
`$\longrightarrow \quad \char "27F6$`	$\longrightarrow \quad \longrightarrow$
`\hbox to 2cm{$\rightarrowfill $}`	————————→
`\hbox to 4cm{$\rightarrowfill $}`	————————————————→
`$\overrightarrow {a+b+c}$`	$\overrightarrow{a+b+c}$
`$\underrightarrow {a+b+c}$`	$\underrightarrow{a+b+c}$

The first two arrows are just characters. The boxed ones are extensibles using leaders that build the arrow from snippets (a hack till we have proper character leaders) and the last two are implemented by abusing the accent mechanism and thereby use the native extensibles of the first character.

The problem here is in names and standards. The first characters have a fixed size while the later are composed. The short ones have the extensibles and can therefore be used as accents (or when supported as character leader). However from the user's perspective, the distinction between the two UNICODE characters might be less clear, not so much when they are used as character, but when used on top of or below something. As a coincidence, while writing this section, a colleague dropped a snippet of MATHML on my desk:

```
<m:math>
  <m:mrow>
    <m:mover accent='true'>
      <m:mrow>
        <m:mi>A</m:mi>
        <m:mi>S</m:mi>
      </m:mrow>
      <m:mo stretchy='true'>→</m:mo>
    </m:mover>
  </m:mrow>
</m:math>
```

However, instead of <m:mo>→</m:mo> there was used `<m:mo>⟶</m:mo>` and that entity is the long arrow. As is often the case in MATHML the rendering is supposed to be quite tolerant and here both should stretch over the row. When a TEX user renders his or her source and sees something wrong, the search for what character or command should be used instead starts. A MATHML user probably just expects things to work. This means that in a system like CONTEXT there will always be hacks and kludges to deal with such matters. It is again one of these areas where optimally the TEX community could have influenced proper and systematic coding, but it didn't happen. So, no matter now good we make an engine or macro package, we always need to be prepared to adapt to what users expect. Let's face it: it's not that trivial to explain why one should favor one or the other arrow as accent: the more it has to cover, the longer it gets and the more we think of long arrows, but adding a whole bunch pf `\longrightarrow...` commands to CONTEXT makes no sense.

Nevertheless, we might eventually provide more MATHML compliant commands at the TEX end. Just consider the following MATHML snippets:[6]

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
    <m:mrow>
        <m:mi>a</m:mi>
        <m:mover>
            <m:mo>&xrarr;</m:mo>
            <m:ms>arrow + text</m:ms>
        </m:mover>
        <m:mi>b</m:mi>
        <m:mover>
            <m:ms>text + arrow</m:ms>
            <m:mo>&xrarr;</m:mo>
        </m:mover>
        <m:mi>c</m:mi>
    </m:mrow>
</m:math>
```

This renders as:

[6] These examples are variations on what we run into in Dutch school math (age 14–16).

$$a \xrightarrow{\text{“arrow + text”}} b \overrightarrow{\text{“text + arrow”}} c$$

Here the same construct is being used for two purposes: put an arrow on top of content that sits on the math axis or put text on an arrow that sits on the math axis. In TEX we have different commands for these:

```
$ a \overrightarrow{b+c} d $ and $ a \mrightarrow{b+c} d $
```

or

$a \overrightarrow{b+c}\ d$ and $a \xrightarrow{b+c} d$

The same is the case for:

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
    <m:mrow>
        <m:mi>a</m:mi>
        <m:munder>
            <m:mo>&xrarr;</m:mo>
            <m:ms>arrow + text</m:ms>
        </m:munder>
        <m:mi>b</m:mi>
        <m:munder>
            <m:ms>text + arrow</m:ms>
            <m:mo>&xrarr;</m:mo>
        </m:munder>
        <m:mi>c</m:mi>
    </m:mrow>
</m:math>
```

or:

$$a \xrightarrow[\text{“arrow + text”}]{} b \underrightarrow{\text{“text + arrow”}} c$$

When no arrow (or other stretchable character) is used, we still need to put one on top of the other, but in any case we need to recognize the two cases that need the special stretch treatment. There is also a combination of over and under:

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
    <m:mrow>
        <m:mi>a</m:mi>
        <m:munderover>
            <m:mo>&xrarr;</m:mo>
            <m:ms>text 1</m:ms>
            <m:ms>text 2</m:ms>
        </m:munderover>
        <m:mi>b</m:mi>
    </m:mrow>
</m:math>
```

$$a \xrightarrow[\text{“text 2”}]{\text{“text 1”}} b$$

And again we need to identify the special stretchable characters from anything otherwise.

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
    <m:mrow>
        <m:mi>a</m:mi>
        <m:munderover>
            <m:ms>text 1</m:ms>
            <m:ms>text 2</m:ms>
            <m:ms>text 3</m:ms>
        </m:munderover>
        <m:mi>b</m:mi>
    </m:mrow>
</m:math>
```

or:

17

$$a \underset{\text{“text 3”}}{\overset{\text{“text 2”}}{\text{“text 1”}}} b$$

And we even can have this:

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
    <m:mrow>
        <m:mi>a</m:mi>
        <m:munderover>
            <m:ms>text 1</m:ms>
            <m:mo>&xrarr;</m:mo>
            <m:ms>text 2</m:ms>
        </m:munderover>
        <m:mi>b</m:mi>
    </m:mrow>
</m:math>
```

$$a \underset{\text{“text 2”}}{\overset{\longrightarrow}{\text{“text 1”}}} b$$

We have been supporting MathML in ConTeXt for a long time and will continue doing it. I will probably reimplement the converter (given a good reason) using more recent subsystems. It doesn't change the fact that in order to support it, we need to have some robust analytical support macros (functions) to deal with situations as mentioned. The TeX engine is not made for that but in the meantime it has become more easy thanks to a combination of TeX, Lua and data tables. Consistent availability of extensibles (either or not virtual) helps too.

Among the conclusions we can draw is that quite a lot of development (font as well as engine) is driven by what we have had for many years. A generic multi–dimensional glyph handler could have covered all odd cases that used to be done with macros but for historic reasons we could still be stuck with several slightly different and overlapping mechanisms. Nevertheless we can help macro writers by providing for instance leaders that accept characters as well in which case in math mode extensibles can be used.

1.10 Fences

Fences are symbols that are put left and/or right of a formula. They adapt their height and depth to the content they surround, so they are vertical extensibles. Users tend to minimize their coding but this is probably not a good idea with fences as there is some magic involved. For instance, TeX always wants a matching left and right fence, even if one is a phantom. So you will normally have something like this:

```
\left\lparent x \right\rparent
```

and when you don't want one of them you use a period:

```
\left\lparent x \right.
```

The question is, can we make the users live easier by magically turning braces, brackets and parentheses etc. into growing ones. As with much in MkIV, it could be that Lua can be of help. However, look at the following cases:

```
\startformula (x) \stopformula
```

$$(x)$$

This internally becomes something like this:

```
open  noad : nucleus : mathchar : U+00028
ord   noad : nucleus : mathchar : U+00078
close noad : nucleus : mathchar : U+00029
```

We get a linked list of three so-called noads where each nucleus is a math character. In addition to a nucleus there can be super- and subscripts.

```
\startformula \mathinner { (x) } \stopformula
```

$$(x)$$

```
inner noad : nucleus : submlist :
  open  noad : nucleus : mathchar : U+00028
```

18

```
  ord   noad : nucleus : mathchar : U+00078
  close noad : nucleus : mathchar : U+00029
```

This is still simple, although the inner primitive results in three extra levels.

```
\startformula \left( x \right) \stopformula
```

$$(x)$$

Now it becomes more complex, although we can still quite well recognize the input. The question is: how easily can we translate the previous examples into this structure.

```
inner noad : nucleus : submlist :
  left  fence : delim   : U+00028
  ord   noad  : nucleus : mathchar U+00078
  right fence : delim   : U+00029
```

```
\startformula ||x|| \stopformula
```

$$\|x\|$$

Again, we can recognize the sequence in the input:

```
ord noad : nucleus : mathchar : U+0007C
ord noad : nucleus : mathchar : U+0007C
ord noad : nucleus : mathchar : U+00078
ord noad : nucleus : mathchar : U+0007C
ord noad : nucleus : mathchar : U+0007C
```

Here we would have to collapse the two bars into one. Now, say that we manage to do this, even if it will cost a lot of code to check all border cases, then how about this?

```
\startformula \left|| x \right|| \stopformula
```

$$\left|| x \right||$$

```
inner noad : nucleus : submlist noad :
  left  fence : delim : U+00028
  ord   noad  : nucleus : mathchar : U+0007C
  ord   noad  : nucleus : mathchar : U+00078
  right fence : delim : U+00029
ord noad : nucleus : mathchar : U+0007C
```

This time we have to look over the sublist and compare the last fence with the character following the sublist. If you keep in mind that there can be all kind of nodes in between, like glue, and that we can have multiple nested fences, it will be clear that this is a no-go. Maybe for simple cases it could work out but for a bit more complex math one ends up in constantly fighting asymmetrical input at the LUA end and occasionally fighting the heuristics at the TEX end.

It is for this reason that we provide a mechanism that users can use to avoid the primitives `\left` and `\right`.

```
\setupmathfences
  [color=red]

\definemathfence
  [fancybracket]
  [bracket]
  [command=yes,
   color=blue]

\startformula
  a \fenced[bar]      {\frac{1}{b}} c \qquad
  a \fenced[doublebar]{\frac{1}{b}} c \qquad
  a \fenced[triplebar]{\frac{1}{b}} c \qquad
  a \fenced[bracket]  {\frac{1}{b}} c \qquad
  a \fancybracket     {\frac{1}{b}} c
\stopformula
```

So, you can either use a generic instance of fences (`\fenced`) or you can define your own commands. There can be several classes of fences and they can inherit and be cloned.

19

$$a\left|\frac{1}{b}\right|c \quad a\left\|\frac{1}{b}\right\|c \quad a\left|||\frac{1}{b}\right|||c \quad a\left[\frac{1}{b}\right]c \quad a\left[\frac{1}{b}\right]c$$

As a bonus ConTEXt provides a few wrappers:

```
\startformula
\Lparent    \frac{1}{a} \Rparent    \quad
\Lbracket   \frac{1}{b} \Rbracket   \quad
\Lbrace     \frac{1}{c} \Rbrace     \quad
\Langle     \frac{1}{d} \Rangle     \quad
\Lbar       \frac{1}{e} \Rbar       \quad
\Ldoublebar \frac{1}{f} \Rdoublebar \quad
\Ltriplebar \frac{1}{f} \Rtriplebar \quad
\Lbracket   \frac{1}{g} \Rparent    \quad
\Langle     \frac{1}{h} \Rnothing
\stopformula
```

which gives:

$$\left(\frac{1}{a}\right) \quad \left[\frac{1}{b}\right] \quad \left\{\frac{1}{c}\right\} \quad \left\langle\frac{1}{d}\right\rangle \quad \left|\frac{1}{e}\right| \quad \left\|\frac{1}{f}\right\| \quad \left|||\frac{1}{f}\right||| \quad \left[\frac{1}{g}\right) \quad \left\langle\frac{1}{h}\right.$$

For bars, the same applies as for primes: we collapse them into proper UniCode characters when applicable:

```
U+007C  VERTICAL LINE                 |  ||
U+2016  DOUBLE VERTICAL LINE          ‖  |||  ||
U+2980  TRIPLE VERTICAL BAR DELIMITER ⦀  ||||  |||  |||  |||
```

The question is always: to what extent do users want to structure their input. For instance, you can define this:

```
\definemathfence [weirdrange] [left="0028,right="005D]
```

and use it as:

```
$ (a,b] = \fenced[weirdrange]{a,b}$
```

This gives $(a,b] = (a,b]$ and unless you want to apply color or use specific features there is nothing wrong with the direct way. Interesting is that the complications are seldom in regular TEX input, but MathML is a different story. There is an `mfenced` element but as users can also use the more direct route, a bit more checking is needed in order to make sure that we have matching open and close symbols. For reasons mentioned before we cannot delegate this to Lua but have to use special versions of the `\left` and `\right` commands.

One complication of making a nice mechanism for this is that we cannot use the direct characters. For instance curly braces are also used for grouping and the less and equal signs serve different purposes. So, no matter what we come up with, these cases remain special. However, in ConTEXt the following is valid:

```
\setupmathfences[color=darkgreen]
\setupmathfences[mirrored][color=darkred]

\startformula
\left { \frac{1}{a} \right } \quad
\left [ \frac{1}{b} \right ] \quad
\left ( \frac{1}{c} \right ) \quad
\left < \frac{1}{d} \right > \quad
\left ⟨ \frac{1}{d} \right ⟩ \quad
\left | \frac{1}{e} \right | \quad
\left  \frac{1}{e} \right  \quad
\left  \frac{1}{e} \right  \quad
\left [ \frac{1}{d} \right [ \quad
\left ] \frac{1}{d} \right [ \quad
\stopformula
```

In the background mapping onto the mentioned left and right commands happens so we do get color support as well. And, it doesn't look that bad in your document source either. Of course other combinations are also possible.

$$\left\{\frac{1}{a}\right\} \quad \left[\frac{1}{b}\right] \quad \left(\frac{1}{c}\right) \quad \left\langle\frac{1}{d}\right\rangle \quad \left\langle\frac{1}{d}\right\rangle \quad \left|\frac{1}{e}\right| \quad \left\langle\!\langle\frac{1}{e}\right\rangle\!\rangle \quad \left\rangle\!\rangle\frac{1}{e}\right\langle\!\langle \quad \left[\frac{1}{d}\right[\quad \left]\frac{1}{d}\right[$$

As there are many ways to get fences and users can come from other macro packages (or use them mixed) we support them all as well as possible.

```
\left (       \frac{1}{x} \right )       =
      (       \frac{1}{x}        )       =
\left\(       \frac{1}{x} \right\)       =
     \(       \frac{1}{x}       \)       =
\left\lparent \frac{1}{x} \right\rparent =
     \lparent \frac{1}{x}       \rparent =
     \Lparent \frac{1}{x}       \Rparent
```

$\left(\frac{1}{x}\right) = (\frac{1}{x}) = \left(\frac{1}{x}\right) = (\frac{1}{x}) = \left(\frac{1}{x}\right) = (\frac{1}{x}) = \left(\frac{1}{x}\right)$

Unfortunately UniCode math doesn't free us from some annoyances with respect to paired fences. On the one hand coding math is a symbolic, abstract matter: a left parenthesis opens something and a right one closes something. The same is true for brackets and braces. However, the bar is used for left and right fencing as well as separating pieces of a formula (e.g. in conditions). Because traditionally these left and right bars were purely vertical with no slope, or hooks, or other thingies attached, in UniCode there is only one slot for it. Where paired fences can play a role in analyzing content, bars are rather useless for that. It also means that when coding a formula one cannot rely on the bar symbol to determine a left or right property. Normally this is no problem as we can use symbolic names (that include the `\left` or `\right` directive) but for instance in rendering MathML it demands some fuzzy logic to be applied. It would have been nice to have code points for the three cases.

```
\ruledhbox{$\left|x\right|$}
\ruledhbox{$\left(x\middle|x\right)$}
\ruledhbox{$\startcheckedfences\left(x\leftorright|x\right)\stopcheckedfences$}
\ruledhbox{$\startcheckedfences\leftorright|x\leftorright|\stopcheckedfences$}
\ruledhbox{$\startcheckedfences\leftorright|x\stopcheckedfences$}
\ruledhbox{$\startcheckedfences\left(x\leftorright|\stopcheckedfences$}
```

Believe me: we run into any combination of these bars and parentheses. And we're no longer surprised to see code like this (generated from applications):

```
<math>
  <mrow>
    <mo>(</mo>
    <mi>y</mi>
    <mrow>
      <mo>|</mo>
    </mrow>
    <mi>y</mi>
    <mo>)</mo>
  </mrow>
</math>
```

Here the bar sits in its own group, so what is it? A lone left, right or middle symbol, meant to stretch with the surroundings or not?

To summarize: there is no real difference (or progress) with respect to fences in LuaTEX compared to traditional TEX. We still need matching `\left` and `\right` usage and catching mismatches automatically is hard. By adding some hooks at the TEX end we can easily check for a missing `\right` but a missing `\left` needs a two-pass approach. Maybe some day in ConTEXt we will end up with multipass math processing and then I'll look into this again.

1.11 Directions

The first time I saw right-to-left math was at a Dante and later at a TUG meeting hosted in Morocco where Azzeddine Lazrek again demonstrated right-to-left math. It was only after Khaled Hosny added some support to the Xits font that I came to supporting it in ConTEXt. Apart from some housekeeping nothing special is needed: the engine is ready for it. Of course it would be nice to extend the lm and gyre fonts as well but currently it's not on the agenda. I expect to add some more control and features in the future, if only because it is a nice visual experience. And writing code for such features is kind of fun.

As this is about as complex as it can gets, it makes a nice example of how we control math font definitions, so let's see how we can define a Xits use case. Because we have a bold (heavy) font too, we define that as well. First we define the two fonts.

```
\starttypescript [math] [xits,xitsbidi] [name]
  \loadfontgoodies [xits-math]
```

```
  \definefontsynonym
    [MathRoman]
    [file:xits-math.otf]
    [features=math\mathsizesuffix,goodies=xits-math]
  \definefontsynonym
    [MathRomanBold]
    [file:xits-mathbold.otf]
    [features=math\mathsizesuffix,goodies=xits-math]
\stoptypescript
```

Discussing font goodies is beyond this article so I stick to a simple explanation. We use so-called goodie files for setting special properties of fonts, but also for defining special treatment, for instance runtime patches. The current `xits-math` goodie file looks as follows:

```
return {
  name = "xits-math",
  version = "1.00",
  comment = "Goodies that complement xits (by Khaled Hosny).",
  author = "Hans Hagen",
  copyright = "ConTeXt development team",
  mathematics = {
    italics = {
      ["xits-math"] = {
         defaultfactor = 0.025,
         disableengine = true,
         corrections   = {
           [0x1D453] = -0.0375, -- f
        },
      },
    },
    alternates = {
      cal       = { feature = 'ss01', value = 1,
        comment = "Mathematical Calligraphic Alphabet" },
      greekssup = { feature = 'ss02', value = 1,
        comment = "Mathematical Greek Sans Serif Alphabet" },
      greekssit = { feature = 'ss03', value = 1,
        comment = "Mathematical Italic Sans Serif Digits" },
      monobfnum = { feature = 'ss04', value = 1,
        comment = "Mathematical Bold Monospace Digits" },
      mathbbbf  = { feature = 'ss05', value = 1,
        comment = "Mathematical Bold Double-Struck Alphabet" },
      mathbbit  = { feature = 'ss06', value = 1,
        comment = "Mathematical Italic Double-Struck Alphabet" },
      mathbbbi  = { feature = 'ss07', value = 1,
        comment = "Mathematical Bold Italic Double-Struck Alphabet" },
      upint     = { feature = 'ss08', value = 1,
        comment = "Upright Integrals" },
      vertnot   = { feature = 'ss09', value = 1,
        comment = "Negated Symbols With Vertical Stroke" },
    },
  }
}
```

There can be many more entries but here the most important one is the `alternates` table. It defines the additional styles available in the font. Alternaties are chosen using commands like

```
\mathalternate{cal}\cal
```

and of course shortcuts for this can be defined.

Of course there is more than math, so we define a serif collection too:

```
\starttypescript [serif] [xits] [name]
  \setups[font:fallback:serif]
  \definefontsynonym[Serif]          [xits-regular.otf]    [features=default]
```

```
  \definefontsynonym[SerifBold]      [xits-bold.otf]       [features=default]
  \definefontsynonym[SerifItalic]    [xits-italic.otf]     [features=default]
  \definefontsynonym[SerifBoldItalic][xits-bolditalic.otf] [features=default]
\stoptypescript
```

If needed you can redefine the `default` feature before this typescript is used. Once we have the fonts defined we can start building a typeface:

```
\starttypescript[xits]
  \definetypeface [xits] [rm] [serif] [xits]   [default]
  \definetypeface [xits] [ss] [sans]  [heros]  [default] [rscale=0.9]
  \definetypeface [xits] [tt] [mono]  [modern] [default] [rscale=1.05]
  \definetypeface [xits] [mm] [math]  [xits]   [default]
\stoptypescript
```

We can now switch to this typeface with:

```
\setupbodyfont[xits]
```

But, as we wanted bidirectional math, something more is needed. Instead of the two fonts we define six. We could have a more abstract reference to the XITS fonts but in cases like this we prefer file names because then at least we can be sure that we get what we ask for.

So, we use the same fonts several times but apply different features to them. This time the typeface definition explicitly turns on both directions. When we don't do that we get only left to right support, which is of course more efficient in terms of font usage.

We can now switch to the bidirectional typeface with:

```
\setupbodyfont[xitsbidi]
```

However, in order to get bidirectional math indeed, we need to turn it on.

```
\setupmathematics[align=r2l]
```

You might have wondered what this special way of defining the features using `\mathsizesuffix` means? The value of this macro is set at font definition time, and can be one of three values: `text`, `script` and `scriptscript`. At this moment the features are defined as follows:

```
\definefontfeature
  [mathematics]
  [mode=base,
   liga=yes,
   kern=yes,
   tlig=yes,
   trep=yes,
   mathalternates=yes,
   mathitalics=yes,
 % nomathitalics=yes, % don't pass to tex
   language=dflt,
   script=math]
```

From this we clone:

```
\definefontfeature
  [mathematics-l2r]
  [mathematics]
  []

\definefontfeature
  [mathematics-r2l]
  [mathematics]
  [language=ara,
   rtlm=yes,
   locl=yes]
```

Watch how we enable two specific features, where `rtlm` is a XITS-specific one. The eventually used features are defined as follows.

```
\definefontfeature[math-text]            [mathematics]    [ssty=no]
```

```
\definefontfeature[math-script]          [mathematics]    [ssty=1,mathsize=yes]
\definefontfeature[math-scriptscript]    [mathematics]    [ssty=2,mathsize=yes]

\definefontfeature[math-text-l2r]        [mathematics-l2r][ssty=no]
\definefontfeature[math-script-l2r]      [mathematics-l2r][ssty=1,mathsize=yes]
\definefontfeature[math-scriptscript-l2r][mathematics-l2r][ssty=2,mathsize=yes]

\definefontfeature[math-text-r2l]        [mathematics-r2l][ssty=no]
\definefontfeature[math-script-r2l]      [mathematics-r2l][ssty=1,mathsize=yes]
\definefontfeature[math-scriptscript-r2l][mathematics-r2l][ssty=2,mathsize=yes]
```

Even if it is relatively simple to do, it makes no sense to build complex mixed mode system, so currently we have to decide before we typeset a formula:

```
\setupmathematics[align=l2r]
\startformula
       \sqrt{x^2\over 4x}  \qquad
  {\bf \sqrt{x^2\over 4x}} \qquad
  {\mb \sqrt{x^2\over 4x}}
\stopformula
```

This gives a left to right formula:

$$\sqrt{\frac{x^2}{4x}} \qquad \sqrt{\frac{\mathbf{x^2}}{\mathbf{4x}}} \qquad \sqrt{\frac{x^2}{\mathbf{4}x}}$$

```
\setupmathematics[align=r2l]
\startformula
       \sqrt{ف^2\over 4ب}  \qquad
  {\bf \sqrt{ف^2\over 4ب}} \qquad
  {\mb \sqrt{ف^2\over 4ب}}
\stopformula
```

And here we get an Arabic formula, where the quality of course is determined by the completeness of the font.

$$\sqrt{\frac{\text{ف}^2}{\mathbf{4}\text{ب}}} \qquad \sqrt{\frac{\text{ف}^2}{\mathbf{4}\text{ب}}} \qquad \sqrt{\frac{\text{ف}^{٢}}{\text{٤ب}}}$$

The bold font has a partial bold implementation so unless I implement a more complex pseudo-bold mechanism you should not expect results. Because we have no official Arabic math alphabets they are not seen by the ConTEXt MkIV analyzers that normally take care of this. It's all a matter of demand and supply (combined with a dose of motivation). For instance while a base size might be covered, the extensibles might be missing.

About the time of writing this another variation was requested at the mailing list. For Persian math we keep the direction from left to right but the digits have to be in an Arabic font. We cannot use the bidirectional handler for this so we need to swap regular and bold digits in another way. We can use the fallback mechanism for this and a definition roughly boils down to this:

```
\definefontfallback
  [mathdigits]
  [dejavusansmono]
  [digitsarabicindic]
  [check=yes,
   force=yes,
   offset=digitsnormal]
```

This is used in:

```
\definefontsynonym
  [MathRoman]
  [file:xits-math.otf]
  [features=math\mathsizesuffix,
   goodies=xits-math,
   fallbacks=mathdigits]
```

The problem with this kind of feature is not so much in the implementation, because by now in ConTEXt we have plenty of ways to deal with such issues in a convenient way. The biggest challenge is to come up with an interface that somehow fits in the model of typescripts and with a couple of predefined typescripts we now have:

```
\usetypescriptfile[mathdigits]
\usetypescript [mathdigits] [xits-dejavu] [arabicindic]
\setupbodyfont[dejavu]
```

After that a formula like `$2 + 3 = 5$` comes out as ۲ + ۳ = ۵. In fact, if you want that in text mode, you can just use the ConTEXt MkIV font feature `anum`:

```
\definefontfeature [persian-fake-math] [arabic] [anum=yes]

\definefont[persianfakemath][dejavusans*persian-fake-math]
```

But of course you won't have proper math then. But as right-to-left math is still under construction, in due time we might end up with more advanced rendering. Currently you can exercise a little control. For instance by using the `align` parameter in combination with the `bidi` parameter. Of course support for special symbols like square roots depends on the font as well. We probably need to mirror a few more characters.

```
\m{   (   1 =   1)   }\quad
\m{   (123 = 123)   }\quad
\m{ a (   1 =   1) b }\quad
\m{ a (123 = 123) b }\quad
\m{ x = 123 y + (1 / \sqrt {x}) }
```

As in math we can assume sane usage of fences, we don't need extensive tests on pairing.

align	**bidi**					
`l2r`	`no`	$(1 = 1)$	$(123 = 123)$	$a(1 = 1)b$	$a(123 = 123)b$	$x = 123y + (1/\sqrt{x})$
`l2r`	`yes`	$(1 = 1)$	$(123 = 123)$	$a(1 = 1)b$	$a(123 = 123)b$	$x = 123y + (1/\sqrt{x})$
`r2l`	`no`	$)1 = 1($	$)321 = 321($	$b)1 = 1(a$	$b)321 = 321(a$	$)\overline{x}\surd/1(+ y321 = x$
`r2l`	`yes`	$(1 = 1)$	$(123 = 123)$	$b(1 = 1)a$	$b(123 = 123)a$	$(\overline{x}\surd/1) + y123 = x$

1.12 Structure

At some point publishers started asking for tagged PDF and as a consequence a typeset math formula suddenly becomes more than a blob of ink. There are several arguments for tagging content. One is accessibility and another is reflow. Personally I think that both arguments are not that relevant. For instance, if you want to help a visually impaired reader, it's far better to start from a well structured original and ship that along with the typeset version. And, if you want reflow, you can better provide a (probably) simplified version in for instance HTML format.

We are surrounded by all kinds of visualizations, and text on paper or some medium is one. We don't make a painting accessible either. If accessibility is a demand, it should be done as best as can be, and the source is then the starting point. Of course publishers don't like that because when a source is available, it's one step closer to reuse by others. But that problem can simply be ignored as we consider publishers to be some kind of facilitating organization that deliver content from others. Alas publishers don't play that humble role so as long as they're around they can demand from their suppliers tagging of something visual.

Of course when you use TEX tagging is no real issue as you can make the input as verbose and structured as you like. But authors don't always want to be verbose, take this:

```
$ f(x) = x^2 + 3x + 7 $
```

This enters TEX as a sequence of characters: $f(x) = x^2 + 3x + 7$. These characters can have properties, for instance they can represent a relation or be an opening or closing symbol, but in most cases they are just classified as ordinary. These properties to some extent control spacing and interplay between math elements. They are not structure. If you have seen presentation MathML you have noticed that there are operators (`mo`), identifiers (`mi`) and numbers (`mn`), as well as some structural elements like fences (`mfenced`), superscripts (`msup`), subscripts (`msub`). Because it is a presentational encoding, there is no guarantee about the quality of the input as well as the rendering, but it somehow made it into a standard that is also used for tagging PDF content.

Going from mostly unstructured TEX math input to more structured output is complicated by the fact that the intermediate somewhat structured math lists eventually become regular boxes, glyphs, kerns, glue etc. In ConTEXt we carry some persistent information around so that we can still reverse engineer the output to structured input but this can be improved by more explicit tagging. We plan to add some more of that to future versions but here is an example:

```
$ \apply{f}{(x)} = x^2 + 3x + 7 $
```

You can go over the top too:

```
$ \apply{f}{(x)} = \mi{x}^\mi{2} + \mi{3}\mi{x} + \mi{7} $
```

The trick is to find an optimal mix of structure and readability. For instance, in `\sin` we already have the apply done by default, so often extra tagging is only needed in situations where there are several ways to interpret the text. Of course we're not enforcing this, but by providing some structure related features, at least we hope to make users aware of the issue. Directly inputting MathML is also an option but has never become popular.

All this is mostly a macro package issue, and ConTeXt has the basics on board. Because there is no need to adapt LuaTeX the most we will do is add a bit more consistency in building the lists (two way pointers) and carrying over properties (like attributes). We also have on the agenda a math table model that suits MathML, because some of those tables are somewhat hard to deal with.

How the export and tagging evolves depends on demand. I must admit that I implemented it as an exercise mostly because these are features I don't need myself (and no one really asked for it anyway).

1.13 Italic correction

Here we face a special situation. In regular OpenType italic correction is not part of the game, although one can cook up some positioning feature that does a similar job. In OpenType math there is italic correction, but also a more powerful sharpe-related kerning which is to be preferred. In traditional TeX the italic correction was present but since it is a font specific feature there is no way to make it work across fonts, and Type1 based math has lots of them.

At some point we have discussed throwing italic correction out of the engine, if only because it was unclear how and when to apply it. In the meantime there is some compromise reached. Because ConTeXt is always in sync with the latest LuaTeX, we oscillated between solutions and this was complicated by the fact that we had to support a mix of OpenType math fonts and virtualized Type1 legacy fonts.

The italic correction related code is still somewhat experimental, but we have several options.[7] In most cases we insert the italic correction ourselves and as the engine then sees a kern already it will not add another one. This has the advantage that we can be more consistent if only because not all fonts have these corrections and not all cases are considered by the engine.

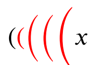
1. A math font can have italic correction per glyph. The engine gets this passed but before it can apply them we already inject them into the mathlist where needed.
2. This is a variant of the first one, but is always applied, and not controlled by the font. This makes it possible to add additional corrections. This method is kind of obsolete as we no longer generate missing corrections at font definition time.[8]
3. This variant looks at the shape and if it is italic (or bolditalic) then correction is applied. Here the correction is related to the emwidth and controlled by a factor. We use this method by default.
4. The fourth variant is a mixture of the first (font driven) and the third (emwidth driven).

Are we better off? I honestly don't know. It is a bit of a mess and will always be, simply because the reference font (cambria) and reference implementation (msword) is not clear about it and we follow them. In that respect I consider it a macro package issue mostly. In ConTeXt at least we can offer some options.

1.14 Big

When migrating math to MkIV I couldn't resist looking into some functionality that currently uses macro magic. An example is big delimiters.

```
$ ( \big( \Big( \bigg( \Bigg( x $
```



Personally I never use these, I just trust `\left` and `\right` to do the right job, but I'm no reference at all when it comes to math. The reason for looking into the bigs is that in plain TeX there are some magic numbers involved. The macros, when translated to ConTeXt boil down to this:

```
\left<delimiter>\vbox to 0.85\bodyfontsize{}\right.
\left<delimiter>\vbox to 1.15\bodyfontsize{}\right.
\left<delimiter>\vbox to 1.45\bodyfontsize{}\right.
\left<delimiter>\vbox to 1.75\bodyfontsize{}\right.
```

[7] In text mode we also have an advanced mechanism for italic correction but this operates independent from math.

[8] Because the font loader is also used for the generic code, we don't want to add such features there.

Knowing that we have a chain of sizes in the font, I was tempted to go for a solution where a specific size is chosen from the linked list of next sizes. There are several strategies possible when we delegate this to LUA but we don't provide a high level interface yet. Personally I'd like to set the low level configuration options as:

```
\setconstant\bigmathdelimitermethod \plusone
\setconstant\bigmathdelimitervariant\plusthree
```

But as users might expect plain–like behaviour, CONTEXT also provides the command

```
\plainbigdelimiters
```

which sets the method to 2. Currently that is the default. When method 1 is chosen there are four variants and the reason for keeping them all is that they are part of experiments and explorations.

1 choose size n from the available sizes
2 choose size 2n from the available sizes
3 choose the first variant that has $1.33^n \times (\text{ht} + \text{dp}) > \text{size}$
4 choose the first variant that has $1.33^n \times \text{bodyfontsize} > \text{size}$

The last three variants give similar results but they are not always the same as the plain method. This is because not all fonts provide the same range.

	pagella	latin modern	cambria
plain	$(((((x$	$(((((x$	$(((((x$
variant 1	$(((((x$	$(((((x$	$(((((x$
variant 2	$(((((x$	$(((((x$	$(((((x$
variant 3	$(((((x$	$(((((x$	$(((((x$
variant 4	$(((((x$	$(((((x$	$(((((x$

So, we are somewhat unpredictable but at least we have several ways to control the situation and better solutions might show up.

1.15 Macros

I already discussed roots and the traditional `\root` command is a nice example of one that can be simplified in LUATEX thanks to a new primitive. A macro package often has quite a lot of macros related to math that deal with tables and LUATEX doesn't change that. But there is a category of commands that became obsolete: the ones that are used to construct characters that are not in the fonts. Keep in mind that the number of fonts as well as their size was limited at the time TEX was written, so by providing building blocks additional characters could be made. Think of for instance the negated symbols: a new symbol could be made by overlaying a slash. The same is true for arrows: by prepending or appending minus signs, arrows of arbitrary length could be constructed.

Here I will stick to another example: dots. In plain TEX we have this definition:

```
\def\vdots
  {\vbox
     {\baselineskip4pt
      \lineskiplimit0pt
      \kern6pt
      \hbox{.}%
      \hbox{.}%
      \hbox{.}}}
```

This will typeset vertical dots, while the next does them diagonally:

```
\def\ddots
  {\mathinner
     {\mkern1mu
```

```
      \raise7pt\vbox{\kern7pt\hbox{.}}%
      \mkern2mu
      \raise4pt\hbox{.}%
      \mkern2mu
      \raise1pt\hbox{.}%
      \mkern1mu}}
```

Of course these dimensions relate to the font size of plain TEX so in CONTEXT MKII we have something like this:

```
\def\vdots
  {\vbox
     {\baselineskip4\points
      \lineskiplimit\zeropoint
      \kern6\points
      \hbox{$\mathsurround\zeropoint.$}%
      \hbox{$\mathsurround\zeropoint.$}%
      \hbox{$\mathsurround\zeropoint.$}}}

\def\ddots
  {\mathinner
     {\mkern1mu
      \raise7\points\vbox{\kern 7\points\hbox{$\mathsurround\zeropoint.$}}%
      \mkern2mu
      \raise4\points\hbox{$\mathsurround\zeropoint.$}%
      \mkern2mu
      \raise \points\hbox{$\mathsurround\zeropoint.$}%
      \mkern1mu}}
```

These two symbols are rendered (in MKII) as follows:

I must admit that I only noticed the rather special height when I turned these macros into virtual characters for the initial virtual UNICODE math that we needed in the first versions of MKIV. This is a side effect of their use in matrices. However, in MKIV we just use the characters in the font and get:

These characters look different because instead of three text periods a real symbol is used. The fact that we have more complete fonts and rely less on special font properties to achieve effects is a good thing, and in this respect it cannot be denied that LUATEX triggered the development of more complete fonts. Of course from the user's perspective the outcome is often the same, although ... using a single character instead of three has the advantage of smaller files (neglectable), less runtime (really neglectable) and cleaner output files (undeniable) from where such characters can now be copied as one.

1.16 Unscripting

If you ever looked into plain TEX you might have noticed this following section. The symbols are more related to programming languages than to math.

```
% The following changes define internal codes as recommended
% in Appendix C of The TeXbook:
\mathcode`\^^@="2201 % \cdot
\mathcode`\^^A="3223 % \downarrow
\mathcode`\^^B="010B % \alpha
\mathcode`\^^C="010C % \beta
\mathcode`\^^D="225E % \land
\mathcode`\^^E="023A % \lnot
\mathcode`\^^F="3232 % \in
\mathcode`\^^G="0119 % \pi
\mathcode`\^^H="0115 % \lambda
\mathcode`\^^I="010D % \gamma
\mathcode`\^^J="010E % \delta
\mathcode`\^^K="3222 % \uparrow
```

```
\mathcode`\^^L="2206 % \pm
\mathcode`\^^M="2208 % \oplus
\mathcode`\^^N="0231 % \infty
\mathcode`\^^O="0140 % \partial
\mathcode`\^^P="321A % \subset
\mathcode`\^^Q="321B % \supset
\mathcode`\^^R="225C % \cap
\mathcode`\^^S="225B % \cup
\mathcode`\^^T="0238 % \forall
\mathcode`\^^U="0239 % \exists
\mathcode`\^^V="220A % \otimes
\mathcode`\^^W="3224 % \leftrightarrow
\mathcode`\^^X="3220 % \leftarrow
\mathcode`\^^Y="3221 % \rightarrow
\mathcode`\^^Z="8000 % \ne
\mathcode`\^^[="2205 % \diamond
\mathcode`\^^\="3214 % \le
\mathcode`\^^]="3215 % \ge
\mathcode`\^^^="3211 % \equiv
\mathcode`\^^_="225F % \lor
```

This means as much as: when I hit `Ctrl-Z` on my keyboard and my editor honors that by injecting character `U+1A` into the input then TEX will turn that into $\neq$, given that you're in math mode. I'm not sure how many keyboards and editors there are around that still do that but it illustrates that inputting in some kind of WYSIWYG is not alien to TEX.[9]

One of the subprojects of the ongoing TEX user group font project is to extend the already extensive Dejavu font with all relevant math characters so that we can edit a document in a more UNICODE savvy way. So, after more than three decades we might arrive where Don Knuth started: you see what you input and a similar shape will end up on paper.

Does this mean that all such input is good? Definitely not, because in UNICODE we find all kinds of characters that somehow ended up there as a result of merging existing encodings. At work we're accustomed to getting input that is a mix of everything a word processor can produce and often we run into characters that users find normal but are not that handy from a TEX perspective. It's the main reason why in math mode we intercept some of them, for instance in:

```
$ y = x² + x³ + x²³ + x²ᵃ $ % not all characters are in monospace
```

These superscripts are an inconsistent bunch so they will never be real substitutes for the ^ syntax, simply because a mix like above looks bad. But fortunately it comes out well: $y = x^2 + x^3 + x^{23} + x^{2a}$. This is because CONTEXT will transform such super- and subscripts into real ones and in the process also collapse multiple scripts into a group. This is typically one of the features that already showed up early in MKIV.

Here we have a feature that doesn't relate to fonts, the math machinery or the engine, but is just a macro package goodie. It's a way to respond to the variation in input, although probably hardly any TEX math user will need it. It's one of those features that comes in handy when you use TEX as invisible backend where the input is never seen by humans.

1.17 Combining fonts

I already mentioned that we started out with virtual math fonts. Defining them is not that hard and boils down to defining what fonts make up the desired math font. Normally one starts out with a decent complete OPENTYPE math font followed by mapping TYPE1 fonts onto specific alphabets and symbols. On top of this there are additional virtual characters constructed (including extensibles). However, this method will become kind of obsolete (read: not used) when all relevant OPENTYPE math fonts are available.

Does this mean that we have only simple font setups? In practice yes: you can set up a math font in a few lines in a regular typescript. There are of course a few more lines needed when defining bold and/or right-to-left math but users don't need to bother about it. All is predefined. There are signals that users want to combine fonts so the already present fallback mechanism for text fonts has been made to work with math fonts as well. This permits for instance to complement the not-yet-finished OPENTYPE Euler math fonts with Pagella. Of course you always need to keep consistency into account, but in principle you can overload for instance specific alphabets, something that can make sense when simple math is mixed with a font that has no math companion. In that case using the text italic in math mode might look better. For the at the time of this writing incomplete Euler font we can add characters like this:

```
\loadtypescriptfile[texgyre]
```

[9] There are more such hidden features, for instance, in some fonts special ligatures can be implemented that no one ever uses.

```
\loadtypescriptfile[dejavu]

\resetfontfallback  [euler]

\definefontfallback [euler] [texgyrepagella-math] [0x02100-0x02BFF]
\definefontfallback [euler] [texgyrepagella-math] [0x1D400-0x1D7FF]

\starttypescript [serif] [euler] [name]
  \setups[font:fallback:serif]
  \definefontsynonym [Serif] [euler] [features=default]
\stoptypescript

\starttypescript [math] [euler] [name]
  \definefontsynonym [MathRoman] [euler] [features=math\mathsizesuffix,fallbacks=euler]
\stoptypescript

\starttypescript [euler]
  \definetypeface [\typescriptone] [rm] [serif] [euler]  [default]
  \definetypeface [\typescriptone] [tt] [mono]  [dejavu] [default] [rscale=0.9]
  \definetypeface [\typescriptone] [mm] [math]  [euler]  [default]
\stoptypescript
```

If needed one can use names instead of code ranges (like `uppercasescript`) as well as map one range onto another. This last option is handy for merging a regular text font into an alphabet (in which case the UniCode's don't match).

We expect math fonts to be rather complete because after all, a font designer has a large repertoire of free alphabets to choose from. So, in practice combining math fonts will happen seldom. In text mode this is more common, especially when multiple scripts are mixed. There is a whole bunch of modules that can generate all kind of tables and overviews for testing.

1.18 Experiments

I won't describe all experiments here. An example of an experiment is a better way of dealing with punctuation, especially the cultural determined period/comma treatment. I still have the code somewhere but the heuristics are too messy to keep around.

There are also some planned experiments, like breaking and aligning display math, but they have a low priority. It's not that hard to do, but I need a good reason. The same is true for equation number placement where primitives are used that can sometimes interfere or not be used in all cases. Currently that placement in combination with alignments is implemented with quite a lot of fuzzy macro code.

One of the areas where experimenting will continue is with fonts. Early in the development of MkIV font goodies showed up. A font (or collection of fonts) can have a file (or more files) that control functionality and can have fixes. There are some in place for math fonts. It is a convenient way to use the latest greatest fonts as we have ways to circumvent issues, for instance with math parameters. The virtual math fonts are also defined as goodies.

Some mechanisms will probably be made accessible from the TEX end so that users can exercise more control. And because we're not done yet, additional features will show up for sure. There are some math related subsystems like physics and chemistry and these already demanded some extensions and might need more. Introducing math symbol (and property) dictionaries as in OpenMath is probably a next step.

I already mentioned that typesetting and rendering related technology is driven by the web. This also reflects on UniCode and OpenType. For instance, we find not only emoticons like `U+1F632` (ASTONISHED FACE) in the standard but also 'MOUNT FUJI', `TOKYO TOWER`, `STATUE OF LIBERTY`, `SILHOUETTE OF JAPAN`. On the other hand, in one of our older projects we still have to provide some tweak for the unary minus (as when discussing scientific calculators used in math lessons) a distinction has to be made with a regular minus sign. And there are no symbols to refer to use of media (simulation, applet, etc.) and there is as far as I know no emoticon for a student asking a question. Somehow it's hard to defend that the Planck constant is as different from a math italic h as a 'GRINNING FACE' is from a 'GRINNING FACE WITH SMILING EYES', but the last both got a code point. I wonder with an `UNAMUSED FACE`.

Of course we can argue that this is all too visual to end up in UniCode, but the main point that I want to make is that as a TEX community (which is also related to education) we are of not that much importance and influence. Maybe it is because we always had a programmable system at hand, and folks who could make fonts, and were already extending and exploring before the web became a factor. Anyhow, in ConTEXt we solve these issues by making mechanisms extensible.

For instance we can extend fonts with virtual glyphs and add features to existing fonts on the fly. Simple examples are adding some glyphs and properties to math fonts or adding color properties to whatever font. More complex examples are implementing paragraph optimizers using feature sets of fonts (most noticeably the upcoming Husayni font for advanced arabic typesetting). And, math typesetting is a speciality anyway.

Upcoming extensions to UNICODE and OPENTYPE will demonstrate that the TEX community could have been a bit more demanding and innovative, given that it had known what to demand. Interesting is that some innovation already happened by providing special fonts and macros and engines, but I guess much gets unnoticed. On the other hand, I must admit that experimenting and providing solutions independent of evolving technology also has benefits: it made (and makes) some user group meetings interesting to go to and creates interesting niches of users. Without this experimental playground I for sure would not be around.

1.19 Tracing

Tracing is available for nearly all mechanisms and math is no exception. Most tracing happens at the LUA end and can be enabled with the tracker mechanism. Users will seldom use this, but for development the situation is definitely more comfortable in MKIV. Of course it helps that the penalty of tracing and logging has become less in recent times because memory as well as runtime is hardly influenced.

We provide several styles (modules) for generating lists and tables of characters and extensibles, visualizing features and comparing fonts. Here we benefit from LUA because we can use the database embedded in CONTEXT and looping and testing is more convenient in this language. Of course the rendering is done by TEX, so this is a typical example of hybrid usage.

1.20 Conclusion

It is somewhat ironic that while CONTEXT is sometimes tagged as ‘not to be used when you need to do math typesetting’ it is this macro package that drives the development of LUATEX with its updated math engine, which in turn influences the updated math engine in XETEX, that is used by other macro packages. In a similar fashion the possibility to process OPENTYPE math fonts in LUATEX triggered the development of such fonts as follow up on the Latin Modern and TEX Gyre projects. So, the fact that in CONTEXT we have a bit more freedom in experimenting with math (and engines) has some generic benefits as well.

I think that overall we’re better off. The implementation at the TEX end is much cleaner because we no longer have to deal with different math encodings and multiple families. Because in CONTEXT we’re less bound to traditional approaches and don’t need to be code compatible with other engines we can follow different routes than usual. After all, that was also one of the main motivations behind starting the LUATEX project: clean (better understandable code), less mean (no more hacks at the TEX end), even if that means to be less lean (quite a lot of LUA code). Between the lines above you can read that I think that we’ve missed some opportunities but that’s a side effect of the community not being that innovative which in turn is probably driven by more or less standard expectations of publishers, as they are more served by good old stability instead of progress. Therefore, we’re probably stuck for a while, if not forever, with what we have now. And a decent CONTEXT math implementation is not going to change that. What matters is that we can (still) keep up with developments outside our sphere of influence.

I don’t claim that the current implementation of math in MKIV is flawless, but eventually we will get there.

Hans Hagen
PRAGMA ADE
Hasselt NL
June-August 2013

Typesetting and Layout in Multiple Directions — Outline

John Plaice

Abstract

I propose a new, general way of looking at typesetting and layout in multiple directions. It subsumes the left-to-right and right-to-left horizontal writing used in most of the world, as well as the vertical writing used in East Asia. The generality allows the development of layout schemes for situations when several writing directions appear on the same page.

The key to the approach is that managing multidirectional text requires a separation of writing style from box direction. It turns out that there are only three different kinds of writing style, and eight kinds of directional box, and that simple rules can be used to define how these different writing styles may appear in different kinds of box.

Outline

There have been numerous attempts to create layout rules for multidirectional text. Some have focused on mixing left-to-right writing with vertical writing, as is commonly needed in East Asia, others have focused on mixed left-to-right and right-to-left writing, as is commonly needed in the Middle East.

My previous endeavor in this direction, when working on the Omega system, came up with 32 possible writing directions, taking into account the direction in which successive lines followed each other on the page, the direction in which successive lines followed each other on a line, and the direction in which each individual glyph looked "up".

Notwithstanding the impressive number of possible writing directions, the proposed solution was not sufficiently general, as it did not make provisions for phenomena such as typesetting to a curve.

In this paper, a completely separate approach, both simpler and more general, is taken: when typesetting, a particular writing style (there are three) is used, and the text will be placed in a secondary (text) box, of which there are four kinds. Secondary boxes are lined up in a primary box.

The three writing styles are as follows:

- In *axial writing* (`A`), there is an axis flowing through the glyphs, typically in the middle, and the glyphs are pinned onto the axis, one after the other. The vertical typesetting used in Japan, China and Korea is axial writing.
- In *left-to-right baseline writing* (`BL`), there is a baseline upon which the glyphs are sitting, and successive glyphs are placed to the right of previous ones. Most alphabetic scripts use this form of writing.
- In *right-to-left baseline writing* (`BR`), there is also a baseline, but in which successive glyphs are placed to the left of previous ones. This form of writing is used for Arabic and Hebrew, but one can also consider the Uighur and Mongolian vertical scripts to be using the same style.

Note that with this notation, there is no notion of writing direction. As a result, one could use any of these three styles in a box of a given direction, but also when typesetting onto a curve. Furthermore, we can do Greek *boustropheidon* typesetting, with alternate writing directions for successive lines.

When using rectangular boxes, we generalize the TEX vboxes and hboxes: we call these *primary boxes* and *secondary boxes*. For each of these boxes, there is a *primary direction* and a *secondary direction*, which must be orthogonal. The directions are defined by a side of the page: hence `TL` means a box whose primary direction is top-down and whose secondary direction is left-to-right, while `RT` means a box whose primary direction is right-to-left and whose secondary direction is top-down. There are therefore eight different direction pairs.

Text is placed inside secondary boxes, each with one or more lines going through the entire length. For each secondary box, upon creation, its base writing style must be defined.

Should this style be axial, then the secondary box will have a *principal axis*, and there can also be a *left axis* and a *right axis*, parallel to the principal axis. These supplementary axes can be used to place furiganas or other forms of annotation, as is common in Japanese and Chinese, or could also be used to place long braces to bracket certain parts of text.

Should this style be one of the two baseline styles, then the secondary box will have a *principal baseline*, and there can also be an *upper baseline* and a *lower baseline*, which serve the same purpose as the left and right axes above.

Any piece of text can be embedded into a text of another style, and into any kind of secondary box, but may need to be rotated or mirrored so that it can fit. The talk and the full paper will give the parameterizations for this infrastructure to work, and explain how existing TEX engines can be adapted appropriately.

◇ John Plaice
Montreal, Canada
UNSW, Sydney, Australia
`johnplaice (at) gmail dot com`
`plaice.web.cse.unsw.edu.au`

Tsukurimashou: a Japanese-language font meta-family

Matthew Skala

Abstract

METAFONT-based font projects for the Chinese, Japanese, and Korean (CJK) languages have been announced every few years since the early 1980s, even predating the current form of the METAFONT language. Except for a few non-parameterized conversions of fonts that originated in other formats, in 30 years every METAFONT CJK font has been abandoned at or before the 8-bit barrier of 256 *kanji*, nowhere near the thousands required for practical typesetting. In this presentation I describe the first project to break that barrier: Tsukurimashou (`http://tsukurimashou.sourceforge.jp/`), currently at over 1500 *kanji* (as well as kana, Latin, and Korean hangul) and steadily growing. I discuss technical and human challenges facing this kind of project, how to solve them, and spin-off technologies such as the IDSgrep *kanji* structural query system.

1 Introduction

The Han script, used by the Chinese, Japanese, and Korean (CJK) languages among others, includes very many characters. Just counting them is tricky, but a human being might typically need to know a few thousand for basic literacy in a Han-script language. The list of 2136 characters taught in the Japanese school system (the *jouyou kanji*) is one benchmark, near the low end. Chinese requires more, and a typesetting system may require more still, because of rare characters found in names, historical contexts, and so on. A human being can get away with failing to read the occasional character; typesetting systems need to be able to print nearly all of them. Computer fonts considered usable for Japanese typically cover between six and twelve thousand characters. Databases of rare characters used in linguistic research cover tens or hundreds of thousands.

The sheer number of characters that go into a CJK font, and the quantity of work implied by that number, is daunting. Considering the difficulty of building even a simple Latin font with METAFONT, it may be no surprise that there are no complete METAFONT-native CJK typefaces. But on the other hand, examination of Han-script text (even, or especially, by someone who cannot read it) quickly reveals that characters can be decomposed into smaller parts, as shown in Figure 1. Computer scientists who examine Figure 1 are likely to believe

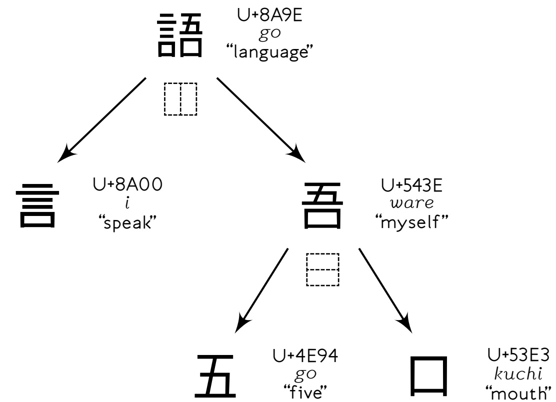


Figure 1: Breaking a character into its parts.

they understand it. "Of course," one supposes, "the tens of thousands of Han characters are just a small vocabulary of primitive shapes, perhaps only a few dozen of those, which combine in straightforward ways according to a spatial grammar to form tree structures!"

Computer scientists know how to deal with such things. It should be only the work of a week or two for a good programmer to lash together a prototype CJK font generator. Each primitive shape can be a subroutine; there can be other subroutines expressing the combining operations such as "place this one above that one"; a few parameters applied to the low-level shapes can allow for creating a wide range of styles; and the only real challenge is looking in the dictionary that lists the tree decompositions of all the characters. That book must exist in China, so we'll get it by interlibrary loan. This project might even be easier than building a Latin font meta-family.

The earliest METAFONT CJK project I know of was LCCD, the Language for Chinese Character Design, described in a 1980 Stanford technical report by Tung Yun Mei [11]. The METAFONT language in its current form did not exist at the time, but Mei collaborated with Knuth and based LCCD on the early METAFONT work. Even in 1980, many of the ideas were already in place that a present-day computer scientist would naturally think of on viewing Figure 1. Mei's report includes images of 346 "basic strokes and radicals," and 112 completed characters.

Subsequent work on METAFONT-native CJK fonts includes that of Hobby and Guoan in 1984, who created 128 characters [5]; Hosek in 1989, character count unknown but two are displayed in the *TUGboat* article [6]; Yiu and Wong in 2003, in a project that targeted on-demand creation of rare

characters rather than a font as such [16]; and Laguna circa 2005, with 130 characters in the last available version [10]. All these used a relatively small number of basic components, combining according to a spatial grammar to form more complicated characters.

I listed published METAFONT-related projects. Similar ideas have also been used behind closed doors in commercial font foundries (CDL from Wenlin Institute seems to be an example [15]), and non-METAFONT research projects like the LISP-based Wadalab toolkit [13]. The Wadalab font project ran during the 1990s; much of the work was lost or withdrawn after hard drive failures and copyright infringement concerns that came to light in 2003, but some of its fonts survived to become widely used in the free software world. These kinds of projects use grammars of character parts, but they lack the full parameterization that METAFONT users expect. There has also been work on using CJK fonts from other sources in TEX documents, sometimes including METAFONT incidentally in the workflow, but again without parameterization. For instance, the Poor Man's Chinese and Japanese package [12] converts bitmap fonts into METAFONT code that renders scaled versions (without smoothing!) at arbitrary resolution.

It may be difficult to create fonts in METAFONT in general, regardless of the script; but human beings have done it. Several if not many METAFONT-native Latin fonts exist, and we can typeset a wide range of documents in Latin-script languages with parameterized METAFONT-native fonts. So after more than three decades of work, why are there no usable, parameterized, METAFONT-native CJK fonts at all?

2 Scaling issues

It is no coincidence that past attempts to build CJK fonts in METAFONT have been abandoned at the same stage in development, around 120 characters. *That is the roughly the size of a Latin font.* METAFONT was designed to build fonts with sizes on that order, and METAFONT users have built expertise and developed tools for building fonts the size of Latin fonts. When fonts get larger, unforeseen difficulties show up like *nurikabe* — the plaster wall monsters of Japanese folklore blamed for delaying travellers by night.

2.1 Technical limitations

Many font file formats are limited to 256 glyphs by their use of 8-bit character codes. People who attempt to typeset CJK documents in classical TEX use elaborate workarounds involving slicing their fonts into 256-glyph sub-fonts. Handling the input encoding for documents written in large character sets with these slicing schemes is a tough problem too, but fortunately not one we must solve as font designers. There are extended versions of the TEX interpreter designed to use longer character codes directly (XƎLATEX is one), and those may also be able to work with font formats that store tens of thousands of glyphs per file and don't need to be sliced; but there is no similarly extended METAFONT to produce fonts in such formats.

Thousands of glyphs in a font does not just mean a bigger file. It also means more time spent compiling, and more memory consumption. One run of METAFONT may run out of memory or other resources trying to process an entire multi-thousand-glyph CJK font, and the user may run out of patience recompiling the whole thing after changing one glyph. To succeed at the thousand-glyph level, a project must have build tools allowing separate compilation of parts of the project. There should be tracking of dependencies among the different parts. Just being able to *find* pieces of code in a project this size — answering questions like "what was the name of the subroutine for such and such a shape?" — is an issue. These are elementary problems in software engineering, but there is little or no previous work on them in the METAFONT context because nobody has built systems this size in METAFONT before.

Classical METAFONT is designed to produce bitmap fonts, but bitmap fonts are no longer such a desired commodity. A present-day CJK font project will presumably target a vector format, but making METAFONT or some variation of it produce vector fonts requires additional layers of software, all of which are to some extent experimental. Bugs in the beyond-METAFONT software, previously undetected because previous fonts were smaller, will show up and need to be fixed. Keeping a handle on the bugs requires a test suite. The need for multiple steps in font compilation underscores the need for a capable build system. Human designers cannot be expected to issue five or six different commands in the right order to recompile every font, every time.

Earlier work on METAFONT CJK fonts has concentrated on writing code in METAFONT to draw the shapes of Han characters, as if that were the only problem to solve. Infrastructure that can scale to the size of the finished product is at least as significant.

2.2 Human factors

It is easy to underestimate how much work is involved in building a CJK font. We know how much work it is to design a Latin font. We know a CJK font has about 30 times as many glyphs. But it is easy to think, looking at Figure 1, that the CJK font should actually only be something like two or three times as much work as the Latin font (or even less), because so much code can be reused. In fact, less work is saved by code reuse than one might hope: every glyph requires some human attention. In computer science terms, font design is not much less than $\Omega(n)$.

Once it becomes clear that a human being must spend time on every single glyph — it gets easier as more code exists to reuse, but there is no break point after which hundreds of characters will suddenly come for free — it is natural to hope for that human being not to be oneself. If we can just build a sufficiently good, easy to use set of tools, we can put them on the Web, maybe use a Wiki, and have many people in the community build a few glyphs each. Many hands make light work, once the infrastructure exists.

But to hope for someone else to build the actual glyphs after the tools are designed is to ignore why people participate in free software projects in the first place. Designing tools for glyph construction is *fun*. Going through a list of 6000 glyphs one by one, doing simple repetitive tasks on each of them, is *work*. It is not easy to get volunteers for that sort of thing at the best of times, let alone when the volunteers must also have proficiency in an obscure programming language. The most successful large-scale collaboration is probably GlyphWiki [9], which sacrifices parameterization for a more purely graphical approach that demands less from the participants.

Finally, many of the potential rewards of a METAFONT CJK project, such as academic publications, can be had at the start, before the boring part; and then there are no more rewards until the end, and few then. You can publish one paper about your innovative techniques for building fonts; and you can publish one paper saying you have finished, years later. There is little in between. Knowing that this is the reward structure makes it tempting to write only the first paper and then start work on something else.

2.3 The script itself

The Han script itself may be the most ferocious *nurikabe*. Figure 1 with its clean decomposition of "language" into "speak," "five," and "mouth," is deceptive. Many characters can be described as simply as that, but many others cannot. Consider Figures 2, 3, 4, and I could draw many more.

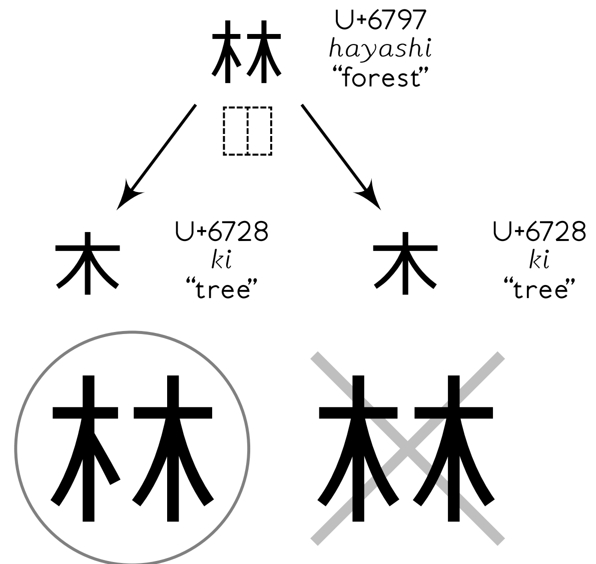


Figure 2: A forest is not two identical trees.

In Figure 2, "forest" is two copies of "tree" placed side by side. But the "tree" on the left is different from the "tree" on the right. If you make the two sides of "forest" look identical, readers will still know that you meant to write "forest," but it will not look right. For a high-quality font, it has got to look right. This entails either creating two different primitives for the two trees, or having a smarter tree that knows how to change itself when it is on the left. Many character components change when they appear on the left. The modifications made when a component appears on the left are partially systematic, so we might hope to write code that can derive the left side shape automatically from the other shape, but it will not be simple, it will require manual supervision, and some projects have not gotten as far as noticing that it was an issue in the first place.

In Figure 3, the left side of "outlook," in addition to not being a character in its own right, is some kind of hard to describe combination of "arrow" and "old bird." It is not good enough to just print a scaled copy of "arrow" on top of "old bird" and hope for the best; getting it right requires modifying and deleting strokes in both parts. A generic overlap operation is unlikely to be flexible enough to do the right thing here. Every character that contains this sort of thing will require specific human

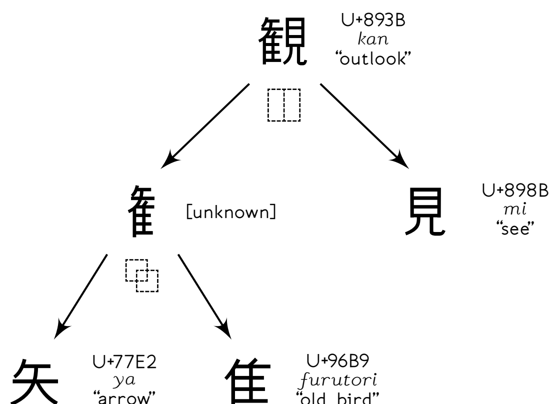


Figure 3: Combining operations are not always simple.

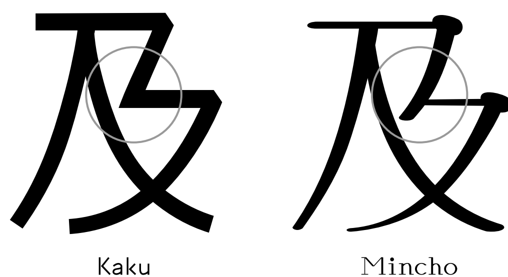


Figure 4: Two styles of U+53CA (*oyo*, "reach").

attention to adjust it beyond just saying "overlap." If the components change parametrically, then making sure they look right for all parameter values becomes even more complicated.

In Figure 4, two different styles of the same character are topologically different: one contains a single zigzag stroke that in the other is made up of two separate pieces. It is not easy to parameterize that in a way that will look good at every step in between, and if we make it a binary choice, giving up on the idea of interpolation, this difference will require some sort of "if" statement in the character description. A straightforward implementation of the grammar of shapes and combining operations suggested by Figure 1 would not provide for "if" statements.

These issues in the Han writing system point to an important conclusion: a simple grammar of parts and combining operations is not enough for building parametric fonts, even though it may be a useful starting point. Many characters can be decomposed into parts in the clean way implied by Figure 1, and such decompositions may be enough to support dictionary searches. It is easy to find enough well-behaved characters to put together a slide show or grant application, and to fool others or even oneself into thinking the whole character set will be easy.

But in order to produce high-quality fonts with full parameterization, with all the characters needed to typeset real documents, we must be able to override the simple descriptions and combinations of parts in arbitrarily complicated ways — per character and depending non-linearly on the parameters. To work at full scale, the font description language must have the power of a general-purpose programming language.

3 Tsukurimashou

My own attempt at building a METAFONT CJK font family is called the Tsukurimashou Project. The name means "Let's make something!"; it is an *anime* reference. As of version 0.8, released 26 August 2013, Tsukurimashou covers 1502 Japanese *kanji* (Han script) characters including all those taught in Japanese schools through Grade Four, as well as essentially complete coverage of *kana* (Japanese phonetic script), Latin, *hangul* (Korean alphabetic script), punctuation, and some miscellaneous ornaments and graphical characters. This is the work of one person, on a hobby basis while doing other things full-time for pay, since late 2010. It remains far from being a complete font family usable for typesetting general documents in Japanese, but it is already far past the point reached by any previous parameterized METAFONT-native CJK font project, and I believe my project is the first with a credible prospect of eventually reaching complete coverage.

Here are some terms of reference distinguishing Tsukurimashou from other projects already discussed:

- Tsukurimashou is a parameterized meta-family, not a single font or a collection of independent fonts.
- Tsukurimashou is a font project, not primarily a dictionary of characters.
- Tsukurimashou is code, not data.
- Tsukurimashou is intended to achieve full coverage, at least of the characters needed for basic literacy in Japanese; it is not a proof of concept.
- Tsukurimashou is one person's non-commercial project; not a for-profit corporate or large-scale collaborative effort.

Tsukurimashou is hosted as a free software project on SourceForge Japan, with the bilingual project home page at `http://tsukurimashou.`

sourceforge.jp/ featuring downloadable packages, a Subversion repository for the source code, a bug tracker, mailing list, and so on. The package as a whole is distributed under the GNU General Public License, version 3, with a clarifying paragraph added to explicitly permit embedding the fonts in documents.

3.1 Motivation

The issues of human labour described in the previous section make it difficult for a CJK METAFONT project to reach complete coverage. Tsukurimashou's solution to the amount of work involved in font design is to redefine that large amount of work as *the main goal* of the project instead of *an unfortunate cost* of the project. This point alone seems to be largely responsible for Tsukurimashou's success to date.

I want to learn to read Japanese. Learning to read entails spending some time practicing and studying every character. But just studying a book and tracing copies on paper, as well as being boring, is not a particularly effective way to learn. I would also like to become skilled at using METAFONT and related font technologies. I believe I acquire skills best by completing tasks that require the skills. Designing a font family for Japanese, as a project that requires knowledge of the *kanji* and of METAFONT, including concentration on every character in turn, is a good way to acquire that knowledge. And from that point of view, the actual finished fonts are not even important. The fonts are my excuse for spending time thinking about every character, which is the real goal. With that goal in mind, *avoiding* human attention to every character stops being necessary or even desirable.

Of course, the project may have desirable side effects. Work on Tsukurimashou has required me to invent new technology that may be useful in other projects. Some of it is publishable research in computer science, certainly welcome for someone hoping to establish an academic career. And because it places heavy (in some cases unprecedented) demands on other free software systems, Tsukurimashou has proven useful in the development of those systems. Given that I am already committing to spend some time per character on learning the language, the hope is to make that time pay off in as many ways as possible.

3.2 A brief tour of the fonts

Tsukurimashou as a software package generates OpenType font files as its main output. Those are intended for use in general typesetting and word pro-

Tsukurimashou Font Meta-Family
さてさて、何が出来るかな？

Kaku 角	Extra Light 白字
Mincho 明朝	Light 軽字
Maru 丸	Normal 本
Bokukko 僕女	Demibold 半太字
Monospace	**Bold 太字**
Proportional	**Extra Bold 黒字**
TsuIta Atama PS	ツイタ頭 *PS*
TsuIta Soku PS	ツイタ足 *PS*
Jieubsida 지읍시다	Dodum 돋움
Batang 바탕	Sun-Moon 신문

Figure 5: A sample of the Tsukurimashou meta-family of fonts.

cessing, not only within the TEX world. I most often use them with XƎTEX. The OpenType fonts are divided up into families, of which the main supported ones are named Tsukurimashou, TsuIta, and Jieubsida; then there is parameterization within each family for overall style, boldness, and monospace or proportional spacing. The main supported styles for the Tsukurimashou family are "Kaku" (a traditional sans-serif style), "Maru" (sans-serif with rounded stroke ends), "Mincho" (a less traditional version of the common Mincho serif style), and "Bokukko" (which somewhat resembles handwriting with a felt-tipped pen). Finer-grained parameters are used internally and could be made visible by modifying the code, much in the way that Computer Modern has internal parameters like "`stem_corr`" as well as preset styles like "Roman." Figure 5 shows a sample of the font styles; Figure 6 shows more of the Japanese characters in the Mincho style. Version 0.8 with all options enabled will build a total of 120 OpenType files, including some that are experimental and not intended for actual use.

These are outline fonts intended for high-resolution printing. They contain hinting for bitmap conversion, but it is done automatically and not expected to be extremely high quality. Japanese-language typesetting has traditionally used monospace metrics, simple scaling (i.e., no corrections for optical weight), and no slanting or italicization; Tsukurimashou currently offers a choice between monospace or proportional, no optical weight features, and italics for the Latin script only.

わらやまはなたさかあ　ワラヤマハナタサカア
ゐり　みひにちしきい　ヰリ　ミヒニチシキイ
　るゆむふぬつすくうん　ルユムフヌツスクウン
ゑれ　めへねてせけえ　ヱレ　メヘネテセケエ
をろよもほのとそこお　ヲロヨモホノトソコオ
一七三上下中九二五人休先入八六円出力十千
口右名四土夕大天女子字学小山川左年手文日
早月木本村林校森正気水火犬玉王生田男町白
百目石空立竹糸耳花草虫見貝赤足車金雨青音

Figure 6: *Kana* and Grade One *kanji* in Tsukurimashou Mincho.

Although the largest use of Tsukurimashou fonts to date has been for typesetting the project's own documentation in English, the design of the Tsukurimashou Latin glyphs, especially in the Mincho style, is intended primarily for setting the short fragments of English that sometimes occur in Japanese text. Tsukurimashou Mincho used for pure English text ends up looking like a display face and might not be appropriate for entire sentences and paragraphs. Tsukurimashou Kaku is more suitable for extended settings in English.

The Jieubsida family (the name is a translation to Korean of "Tsukurimashou") is intended to support Korean *hangul* (alphabetic) script. *Hanja* (the Korean equivalent of *kanji*) are not included. This character set is relatively orthogonal: the main sequence of 11172 glyphs is algorithmically generated from a few tens of basic parts, though many less common letters had to be defined with more human intervention. Work on these fonts has proven useful in debugging the infrastructure at full scale, given that the Tsukurimashou series of fonts will eventually grow to a significant fraction of the size already reached by the Jieubsida series.

Beyond the main Tsukurimashou package, there are several smaller software packages called "parasites," which appear in subdirectories of the distribution or may be detached. Some of these are font packages that share some of the Tsukurimashou infrastructure without really being part of the same meta-family; others are related software of other kinds. The only one discussed here will be the IDSgrep structural query system.

3.3 The infrastructure

Tsukurimashou's infrastructure is designed like a typical free software project. It has source code that compiles into binary files, it has build scripts to accomplish that, and a would-be user can download a tarball, unpack it, and type `./configure` and `make`.

The build system is based on GNU Autotools. Choosing which source code files are needed for which font styles involves doing some logical inference that would not be convenient to do in a Makefile, so the Makefiles invoke additional code written in a subset of Prolog to evaluate the style selections, then run Perl scripts that scan the METAFONT sources to look for dependencies. The results of that computation are written into additional Makefiles, which guide the actual compilation process.

Knuth's METAFONT was designed with bitmap fonts in mind, whereas Tsukurimashou's target is OpenType outline fonts. There are several METAFONT variants that can produce outline output from METAFONT source. I chose MetaType1 [7] for Tsukurimashou. This package originates with the Polish TEX users group GUST and may be most famous for its use in the Latin Modern project [8]. It consists primarily of a macro package for Metapost and a postprocessing script for GNU `awk`. One run of Metapost generates the glyphs of a font as EPS files; another generates metrics; then the `gawk` script merges those and does some rewriting of the Postscript code to turn them into a single Postscript Type 1 font.

In recent versions, Tsukurimashou's version of MetaType1 has diverged somewhat from the one distributed by GUST. I started with the (very old) `mtype13` distribution, tried to upgrade it to use the latest MetaType1 scripts, and ended up rewriting large sections of code. Many features of MetaType1 are not used in Tsukurimashou (for instance, hinting; the "metrics" pass; and the entire processing chain in the reverse direction from Postscript back to METAFONT), and it proved useful to remove them, streamlining the code considerably. The core flow of information through Tsukurimashou's version of MetaType1 remains similar to that of the original, however: the Metapost interpreter executes code in the METAFONT language, writing one EPS file for each glyph, and then those are postprocessed into Postscript Type 1 fonts.

Each Postscript font contains up to 256 glyphs (but usually far fewer than that), corresponding to a 256-character block of the Unicode character space. Many of these Postscript fonts are needed for each full-coverage OpenType font. The build system runs them individually through a FontForge script that removes overlapping sections of splines, this being an easier operation in FontForge than on the METAFONT side, and then once

all Postscript fonts for an OpenType font have had their overlaps removed, it runs another FontForge script to combine them into the final OpenType font. Doing the overlap removal as a separate step is an optimization for the common case during development where only some of the Postscript fonts have changed: it reduces the amount of work needed to reassemble the updated OpenType font.

There are additional stages of processing in FontForge after the Postscript fonts are merged. The raw outlines generated by METAFONT may contain excessive or poorly-located spline control points; scripts in FontForge attempt to remove those. Similarly, some technical rules of the font formats (such as having points at the x and y extrema of each curve) need to be enforced. There is another processing chain for automated horizontal spacing and kerning of the proportionally-spaced styles. In that chain, the build system generates bitmap fonts in BDF format and a C program calculates spacing corrections, which are then applied back to the merged OpenType fonts. Other scripts run on the side do things like constructing OpenType glyph-substitution tables for Korean *hangul* support, and collecting data for proof generation. According to recent statistics from Ohloh [2], 63% of the project's code is written in Metapost (the font descriptions proper), 8% is in LaTeX (documentation), and the remaining 29% is spread among 11 other programming languages: the infrastructure and some small spin-off packages.

3.4 The METAFONT code

Here is Tsukurimashou's code defining the "language" glyph of Figure 1; three styles of it are shown at the top of Figure 7. This glyph is of about average complexity; some are even simpler, and a few involve much more complicated operations, such as calculating positions of strokes based on the intersections of other strokes, or doing interpolation and conditional processing on style parameters.

```
vardef kanji.grtwo.language =
  push_pbox_toexpand(
    "kanji.grtwo.language");
  build_kanji.level(build_kanji.lr(450,0)
    (kanji.grtwo.word)
    (tsu_xform(identity yscaled 0.95)
      (kanji.grnine.my)));
  expand_pbox;
enddef;
```

This code exists in a file named `tsuku-8a.mp`, which covers the Unicode code points U+8A00 to U+8AFF. A character like this one, which happens not to be used as part of any other character, is defined right there in the Unicode-range Metapost file. Parts that are shared among more than one such file are moved to other files that can be included in multiple places; for instance, `kanji.grtwo.word` is in `gradetwo.mp`. Splitting macro definitions across many files like this makes it easier to avoid recompiling the whole system when something changes, but it also requires the build system to keep track of all the inter-file dependencies.

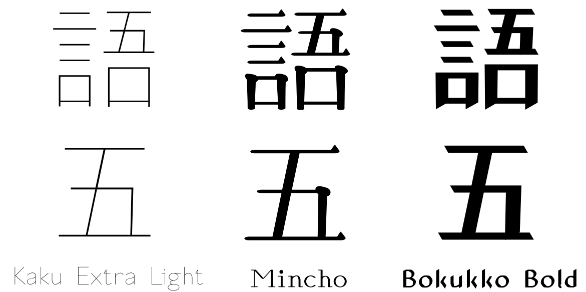


Figure 7: Three styles of "language" and "five."

Tsukurimashou frequently uses a sort of functional programming via METAFONT's concept of text arguments to macros. There is a global stack data structure of objects (several kinds) that will eventually be rendered into the glyph. A macro will receive one or more arguments that are themselves fragments of code; it runs them, then examines the objects they added to the stack and possibly makes modifications. Macros that create *kanji* or parts of *kanji* normally put them into a square of arbitrary two-dimensional space defined by the coordinates from $(50, -50)$ to $(950, 850)$; the outer-level macros can then shift and scale that square into its final location in the finished glyph.

The macro `build_kanji.lr`, for combining things left-to-right, allows its two arguments to run, then scales and shifts their results to cover two smaller rectangles. The numeric arguments $(450, 0)$ specify that in this case, the dividing line is at x coordinate 450, and the two rectangles overlap by an amount of 0. So the left side runs from $(50, -50)$ to $(450, 850)$ and the right side is from $(450, -50)$ to $(950, 850)$.

Many of the visual adjustments needed when parts are combined, can be had just by choosing the right values for the dividing line and overlap amount. But other macros seen in this sample include `build_kanji.level`, which adjusts the stroke widths in its argument to all be the same (which often, but not always, looks better) and `tsu_xform`, which applies an additional METAFONT transformation matrix to make `kanji.grnine.my` a little

smaller. Even in this very simple glyph, some tweaking was necessary beyond just putting together existing pieces in a standardized way.

Here is code for the *kanji* numeral "five," which is invoked indirectly by `kanji.grtwo.language` when it calls `kanji.grnine.my`. This glyph is shown at the bottom of Figure 7. This is typical of the basic shapes that are not made up of smaller components.

```
vardef kanji.grone.five =
  push_pbox_toexpand("kanji.grone.five");
  push_stroke((170,740)--(830,740),
    (1.6,1.6)--(1.6,1.6));
  set_boserif(0,1,9);
  push_stroke((500,740)--(350,20),
    (1.6,1.6)--(1.6,1.6));
  push_stroke(
    (220,410)--(730,410)--(720,20),
    (1.5,1.5)--(1.5,1.5)--(1.4,1.4));
  set_boserif(0,1,4);
  set_botip(0,1,1);
  push_stroke((120,20)--(880,20),
    (1.6,1.6)--(1.6,1.6));
  set_boserif(0,1,9);
  expand_pbox;
enddef;
```

The `push_stroke` macros save paths on the stack, with each stroke defined by one path for the spine of the stroke, and a second path describing how the stroke weight (eventually translated to "width" through a style-dependent matrix) changes along the length of the stroke. Other macros, such as `set_boserif`, push other objects on the stack to indicate where serifs (*uroko*) should be added in styles that use them. The whole thing, like `kanji.grtwo.language` before it, is bracketed by `push_pbox_toexpand` and `expand_pbox`, which respectively save, and adjust the size of, an object called a "proof box."

After all the macros for a glyph have run, rendering code unwinds the stack and generates outlines for all the objects, writing them to the Postscript output. This code is where most aspects of the font style are applied. Styles define the pens used for stroking, transformations for calculating pen size, the shape of serifs and whether to use them, and can potentially override parts of the rendering code by defining hook macros to apply further effects.

I have never fully understood METAFONT's traditional proof system based on greyscale fonts and "literate" programming, and in any case its reliance on the standard coordinate array `z[]` would not mix well with Tsukurimashou's object stack concept. Tsukruimashou generates proofs in a completely different way. When unwinding the stack the rendering code writes a "proof file," essentially a machine-readable log of all the things it is rendering. The build system collects the proof files and runs them through Perl scripts which generate TikZ/LaTeX files for an illustrated and cross-referenced edition of the source code. The proof boxes from `push_pbox_toexpand` result in annotations on the pictures, showing which part of each glyph came from which macro. Some information from the proof files also feeds into the kerning program, and is used for purposes like advising FontForge of white-on-black reversed glyphs, which represent exceptions to the overlap-removal rules otherwise applied.

4 Character datbases and IDSgrep

Adding characters to Tsukurimashou requires knowing what is already in the system and what is in the language: when looking at something like the left side of "outlook," I need to know whether such a thing already exists as a macro somewhere in the code base; whether many other characters in the language also include it, which would support the decision to create a new macro for future use; and which of its parts may be related to common shapes that could be used as guides for the new code. There are also simple coding questions like "What was the name of that macro?" and "Which source code file is it in?"

More generally, anyone working with Han characters who does not read them fluently may wish to search a dictionary on partial descriptions: "What is this character I don't recognize that has 'speak' on the left and 'five' at the upper right?" Existing dictionaries sometimes offer what is called "multi-radical" search, whereby the user can specify one or more components and then see a list of all *kanji* that contain all those components. But multi-radical search features seldom if ever capture structural information like "on the left"; such a system would just show all the characters that contain "speak" in one pile for the user to dig through. In the initial stages of laying out Tsukurimashou's *kanji* support, I frequently found myself wishing I could use the power of Unix regular expressions, or something like them, to make more precise queries: why not run `grep` on the writing system itself?

The IDSgrep package attempts to serve that need. With some irony intended, IDSgrep's stated goal is to bring the user-friendliness of `grep` to Han character dictionaries. IDSgrep is one of the Tsukurimashou parasites: it comes included with the full

【林】⿰木木

【語】⿰言⿱五口

【観】⿰⿻矢隹⿱目儿

【涼】⿰水⿱亠⿱口小

【葉】⿱艹⿱世木

Figure 8: Unicode Ideographic Description Sequences.

distribution in a separate directory, or can be distributed on its own.

Recall the tree decomposition of Figure 1. That tree might be rendered into a simple ASCII-based prefix notation as "`[lr](speak)[tb](five)(mouth)`": it is a left-right combination of two things, the first of which is "speak" and the second is a top-bottom combination of "five" and "mouth." As argued earlier in this paper, such descriptions are not enough to render high-quality glyphs; but maybe if we include a few general catch-all categories like "overlap," and accept that not all descriptions will be detailed enough for rendering graphics, we can come up with a description for every character sufficient to offer useful dictionary searches.

The Unicode standard specifies syntax for Ideographic Description Sequences (IDSes), intended to support exactly this kind of pursuit [14]. There are special characters defined in the range U+2FF0 to U+2FFB to represent the prefix operators. Figure 8 shows some examples of the notation. Note the way the IDS notation conceals some details: for instance, the two sides of "forest" are both denoted by the same character, even though they look different when rendered. This looks promising: maybe we could get away with "just running `grep`" on a database of such decompositions.

In practice there are some additional challenges. For theoretical reasons, namely the difference between regular and context-free languages, a true regular expression search on these descriptions may be less than satisfactory. IDSgrep implements a tree-matching query language in which the user can specify character components to search for explicitly, or use matching operators like wildcard, match-anywhere, Boolean operations, and so on. The IDS syntax is not quite sufficiently flexible and well-defined to encompass all the tasks IDSgrep demands of it, and the special Unicode combining operation characters are difficult to type (and to typeset in Computer Modern!); so IDSgrep defines extensions to the syntax and ASCII synonyms for the special characters, forming a language of Extended Ideographic Description Sequences (EIDSes) that subsumes the Unicode IDS syntax.

IDSgrep's user interface consists of a Unix command-line utility similar to `grep`. It reads a database of trees in EIDS syntax, from files or standard input, and writes out any that match the matching pattern specified on the command line: just like `grep`. The syntax for matching patterns is complicated because it is powerful, but no worse for skilled users than standard regular expressions. After learning the syntax, a user can easily and quickly compose queries like "What characters have this
component in that location, but not that other component anywhere?"

The latest version, IDSgrep 0.4, uses Bloom filters and binary decision diagrams to speed up searches. Although the full tree-matching algorithm is not slow, a complete search of hundreds of thousands of *kanji* dictionary entries may take a few seconds. So during installation, IDSgrep precomputes bit vector indices for the databases being installed; when searching those databases, it can do quick tests on the bit vectors to reject the large majority of possible matches, running the more expensive tree match on the candidates that make it past the bit vector check. The amount of speed-up is variable, but typically around a factor of 15.

But a critical question remains: where does the data come from? Databases of *kanji* marked up with structural data are not easy to find, let alone in IDSgrep's native format. The Tsukurimashou fonts generate (using information extracted from the proof files) a dictionary of character decompositions *as the characters appear in the fonts*. Querying how Tsukurimashou decomposes a character is often useful, but Tsukurimashou by definition does not cover the characters I have yet to add, and its decompositions may not reflect traditional etymology and other concerns. IDSgrep also ships with code to extract EIDS character decompositions from the KanjiVG Project's XML files [1] and from the CHISE IDS database [4]. It can do a "join" of any of the *kanji* databases with EDICT2 [3] to create an experimental dictionary of words and meanings with character decompositions. None of these databases is perfect; but especially by searching several at once, users can usually succeed in finding what they are looking for.

5 Conclusions and future work

There has been much past CJK METAFONT work, with few results and no finished fonts. I have described my own project, the Tsukurimashou parametric font meta-family, which is unfinished too. However, Tsukurimashou has made more progress than any similar system to date. I have described issues facing this kind of project, Tsukurimashou's solutions for some of them, and associated technology including the IDSgrep *kanji* structural query system.

The obvious direction for future work is to complete Tsukurimashou's *kanji* coverage. My hope, however, is that some of the code and ideas from this project will also be applicable in other languages and other projects.

References

[1] Ulrich Apel. KanjiVG. Online `http://kanjivg.tagaini.net/`.

[2] Black Duck Software. The Tsukurimashou open source project on Ohloh: Languages page. Online `https://www.ohloh.net/p/tsukurimashou/analyses/latest/languages_summary`.

[3] Jim Breen. The EDICT dictionary file. Online `http://www.csse.monash.edu.au/~jwb/edict.html`.

[4] CHISE Project. Home page. Online `http://www.chise.org/`.

[5] John D. Hobby and Gu Guoan. A Chinese meta-font. *TUGboat*, 5(2):119–136, November 1984.

[6] Don Hosek. Design of Oriental characters with METAFONT. *TUGboat*, 10(4):499–502, December 1989.

[7] Bogusław Jackowski, Janusz Nowacki, and Piotr Strzelczyk. Programming PostScript Type 1 fonts using MetaType1: Auditing, enhancing, creating. *TUGboat*, 24(3):575–581, 2003.

[8] Bogusław Jackowski and Janusz M. Nowacki. Latin Modern: Enhancing Computer Modern with accents, accents, accents. *TUGboat*, 24(1):64–74, 2003.

[9] Koichi Kamichi. Glyphwiki. Online `http://en.glyphwiki.org/wiki/GlyphWiki:MainPage`.

[10] Javier Rodríguez Laguna. Hóng-Zì: A Chinese METAFONT. *TUGboat*, 26(2):125–128, 2005.

[11] Tung Yun Mei. LCCD, a language for Chinese character design. Report STAN-CS-80-824, Stanford University, Department of Computer Science, 1980.

[12] Tom Ridgeway. Poor Man's Chinese and Japanese, 1990. Online `http://www.ctan.org/tex-archive/fonts/poorman`.

[13] Tetsuro Tanaka. Wadalab-Toolkit. Web page in Japanese. Online `http://gps.tanaka.ecc.u-tokyo.ac.jp/wadalabfont/`.

[14] Unicode Consortium. Ideographic description characters. In *The Unicode Standard, Version 6.2.0*, section 12.2. The Unicode Consortium, Mountain View, USA, 2012. Online `http://www.unicode.org/versions/Unicode6.2.0/ch12.pdf`.

[15] Wenlin Institute. Character description language. Online `http://www.wenlin.com/cdl/`.

[16] Candy L. K. Yiu and Wai Wong. Chinese character synthesis using MetaPost. *TUGboat*, 24(1):85–93, 2003.

◇ Matthew Skala
Department of Computer Science
E2–445 EITC
University of Manitoba
Winnipeg MB R3T 2N2
Canada
`mskala@ansuz.sooke.bc.ca`
`http://ansuz.sooke.bc.ca/`

The XΥMTEX System for Publishing Interdisciplinary Chemistry/Mathematics Books

Shinsaku Fujita

1 XΥMTEX Version 5.01

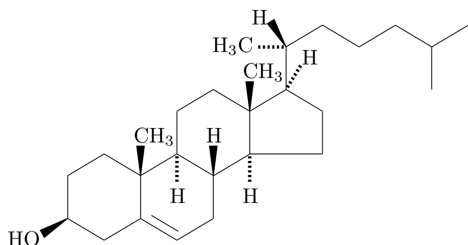
I have recently released XΥMTEX Version 5.01 for drawing chemical structural formulas, where its zip file (xymtx501.zip) is available from my personal homepage (http://xymtex.com/). The XΥMTEX system supports three modes for drawing:

1. the LATEX-compatible mode, which is based on the LATEX-picture environment along with the epic package,
2. the PostScript-compatible mode, which is based on the PSTricks package, and
3. the PDF-compatible mode, which is based on the pgf/TikZ package.

The three modes can be switched by loading the xymtex, xymtexps, or xymtexpdf package by using the ¥usepackage command. If structural formulas of high quality are necessary, the latter two modes should be selected. A typical template for switching the three modes is shown below:

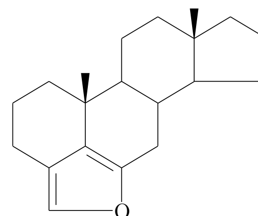
```
¥documentclass{article}
%¥usepackage{xymtex}%LaTeX mode
%¥usepackage{xymtexps}%PostScript mode
¥usepackage{xymtexpdf}%PDF mode
¥usepackage{graphicx}
¥begin{document}
¥cholestane[e]{3B==HO}%XyMTeX command
¥end{document}
```

The XΥMTEX command ¥cholestane with the arguments [e] and {3B==HO} generates the chemical structural formula of cholest-5-en-3β-ol as follows:



XΥMTEX commands are equipped with facilities for drawing complex structures, i.e., the substitution technique for attaching substituents, the addition technique for drawing fused rings, and the replacement technique for drawing spiro rings. For example, the structure of furo[40,30,20:4,5,6]androstane is drawn by the addition technique, where the ¥fivefusevi command for drawing a 5-membered fusing unit is declared in the bond list of the ¥steroid command for drawing a steroid skeleton:

```
¥steroid
[{c{¥fivefusevi[ad]{3==O}{}{e}[a]}}]
{{10}B==¥null;{13}B==¥null}
```



The detailed document of the XΥMTEX system [1] is available from my homepage located at http://xymtex.com/.

2 Interdisciplinary Chemistry/Mathematics Books

The development of the XΥMTEX system highly reflects the personal history of my researches aiming at the integration of chemistry and mathematics, e.g., the concept of imaginary transition structures (ITSs) [2], the USCI (unit-subduced-cycle-index) approach [3, 4], the concept of stereoisograms [5, 6], the proligand method [7], and the concept of mandalas [8].

2.1 Manual Drawing Without Using the XΥMTEX System

In 1991, I published an interdisciplinary monograph for combinatorial enumeration of chemical compounds as three-dimensional structures (the USCI approach) [9]. This book contains many structural formulas of organic compounds along with mathematical equations because of its interdisciplinary nature. Such mathematical equations were successfully typeset by means of the original utilities of the TEX/LATEX system. However, the structural formulas contained in this book were drawn manually and pasted on the camera-ready manuscript, because the TEX/LATEX system supported no reliable utility for drawing structural formulas at that time and because commercially available systems such as ChemDraw were too expensive to be used for personal purposes.

The concept of imaginary transition structures (ITSs), which serves as computer-oriented representation of organic reactions, was developed mainly during the 1980s. Just in 2001, rather belatedly, I published a monograph on the concept of ITSs [10]. Although such ITSs can be regarded as extended

structural formulas with colored bonds (par-bonds, out-bonds, and in-bonds), the XΥMTEX system at that time did not support utilities of coloring bonds. It follows that the ITSs contained in this book were drawn manually and pasted on the camera-ready manuscript.

2.2 Drawing by the XΥMTEX System

The XΥMTEX system was developed and released in 1993 as a LaTeX tool for drawing structural formulas. The manual was published as a book in 1997 [11]. However, it was not until version 4.00 that the XΥMTEX system supported the PostScript-compatible mode for drawing structural formulas of high printing quality [12].

The PostScript-compatible mode was applied to prepare a book for surveying organic compounds for color photography [13]. Along with chemical or mathematical equations, this book contains 480 figures, each of which consists of several structural formulas drawn by the XΥMTEX system.

The book published in 2007 deals with a new concept *mandalas*, which I have proposed as a basis for rationalizing enumeration of three-dimensional structures [14]. This book contains many mathematical equations as well as structural formulas because of interdisciplinary nature, where the mathematical equations were typeset by the original TEX/LaTeXutilities and the structural formulas were drawn by the XΥMTEX system.

The book published in 2013 is concerned with the *proligand method*, which I have proposed to enumerate three-dimensional structures [15]. This book indicates that the proligand method for enumerating three-dimensional structures can be degenerated into the Pólya's method for enumerating graphs.

Moreover, the on-line manual [1] of the XΥMTEX system itself provides us with an illustrative example for publishing a book which contains both chemical structural formulas and mathematical equations.

Because the XΥMTEX Version 5.01 supports utilities for coloring structural formulas, the book published in 2001 would be rewritten with maintaining bond colors (par-bonds, out-bonds, and in-bonds). This has been briefly discussed in Section 39.4 in the on-line manual [1].

3 Conclusion

As clarified by the publication of the interdisciplinary chemistry/mathematics books described above, the XΥMTEX system coupled with the LaTeX system has been proven to be a reliable tool for publishing books of high printing quality which contain structural formulas along with mathematical equations.

◇ Shinsaku Fujita
Shonan Institute of
Chemoinformatics and
Mathematical Chemistry
`http://xymtex.com`

References

[1] S. Fujita, "XΥMTEX: Reliable Tool for Drawing Chemical Structural Formulas," Shonan Institute of Chemoinformatics and Mathematical Chemistry, Kanagawa (2013), http://xymtex.com/fujitas3/xymtex/indexe.html.

[2] S. Fujita, *J. Chem. Inf. Comput. Sci.*, **26**, 205–212 (1986).

[3] S. Fujita, *Theor. Chim. Acta*, **76**, 247–268 (1989).

[4] S. Fujita, *J. Am. Chem. Soc.*, **112**, 3390–3397 (1990).

[5] S. Fujita, *J. Org. Chem.*, **69**, 3158–3165 (2004).

[6] S. Fujita, *Tetrahedron*, **60**, 11629–11638 (2004).

[7] S. Fujita, *Theor. Chem. Acc.*, **113**, 73–79 (2005).

[8] S. Fujita, *J. Math. Chem.*, **42**, 481–534 (2007).

[9] S. Fujita, "Symmetry and Combinatorial Enumeration in Chemistry," Springer-Verlag, Berlin-Heidelberg (1991).

[10] S. Fujita, "Computer-Oriented Representation of Organic Reactions," Yoshioka-Shoten, Kyoto (2001).

[11] S. Fujita, "XΥMTEX—Typesetting Chemical Structural Formulas," Addison-Wesley Japan, Tokyo (1997).

[12] S. Fujita, *J. Comput. Chem. Jpn.*, **4**, 69–78 (2005).

[13] S. Fujita, "Organic Chemistry of Photography," Springer-Verlag, Berlin-Heidelberg (2004).

[14] S. Fujita, "Diagrammatical Approach to Molecular Symmetry and Enumeration of Stereoisomers," University of Kragujevac, Faculty of Science, Kragujevac (2007).

[15] S. Fujita, "Combinatorial Enumeration of Graphs, Three-Dimensional Structures, and Chemical Compounds," University of Kragujevac, Faculty of Science, Kragujevac (2013).

The Multibibliography Package

Michael Cohen, Yannis Haralambous and Boris Veytsman

Abstract

Conventional standards for bibliography styles entail a forced choice between index and name-year citations and corresponding references. We reject this false dichotomy, and describe a multibibliography, comprising alphabetic, sequenced, and also chronological orderings of references. An extended inline citation format is presented which integrates such heterogeneous styles, and is useful even without separate bibliographies. Richly hyperlinked for electronic browsing, the citations are articulated to select particular bibliographies, and the bibliographies are cross-referenced through their labels, linking among them.

1 Introduction

One of the aims of the list of references in an academic paper or book is to show the reader the current state of the field. A good bibliography creates a narrative, showing the context of the current paper or book in the general picture of scientific inquiry — those proverbial "shoulders of giants" on which it stands.

There are two main ways to organize such a narrative: either around the ideas or around the authors. In the first case the order of citation follows the order of their mention in the main text. Thus the logic of the text is reflected in the bibliography list. In the second case the order of citations follows the authorship: we want alphabetic order by authors (with chronological subordering of works by the same authors). Accordingly, the inline citations in the first cases are usually numerical, whereas in the second case they are either numerical or, when possible, based on the authors' names and publication years (perhaps abbreviated or contracted). This is the main difference between "numerical" and "named" bibliography styles [Daly, 2011: 1]. Both these styles have their own advantages and disadvantages. It is possible to imagine a third option: ordering the citation primarily accordingly to publication year, thus showing the chronology of the progress in the field.

One may ask, why not use the advantages of both the currently employed styles, generating down not one, but multiple lists of references? In the old days, when bibliographies were created and sorted manually, such a task was prohibitively expensive. This is no longer true.

Encouraged by the programmability of bibliographic styles and the flexibility of compiled formatting, we propose an extension of academic and scientific bibliographic styles. Conventional inline bibliographic citations, indicating full references in a separated bibliography, are either ordinal numbers generated according to first appearance in a document or a tag composed (perhaps abbreviated or contracted) of respective authors' names and publication year. To reconcile desire to simultaneously deploy these heretofore mutually exclusive styles, we introduce a "multibibliography," combining both "numerical" and "named" styles. We also add a chronological list, integrating all the information for the inline citation. This idea was conceived by the first author and implemented partly by the second author and the third author.

Rather than having to choose between citations generated as

index numbers,

- corresponding to alphabetically sorted authors names, as in BibTeXs "`plain`" style,
- in order of first appearance in the document, as in the "`unsrt`" style, or

author names and publication year (or some abbreviation thereof), as in the "`alpha`" style,

we use both, mixing the two styles, as in "(Suzuki, 2013: 57)", or, in case of associated page numbers, "(Suzuki, 2013: 57, p. 45–67)".

This is admittedly unorthodox, unusual and unique, but satisfies our desire to have an easily understood cross-reference (without ambiguity in the case of name collision) and an ordinal reference (the last entry also serving as a cardinal reference count), and also our preference to be able to see an inline reminder of the respective authors. As a bonus highlighting such usefulness, a "timeline" bibliography is also generated in chronological order.

2 Implementation and Invocation

The multibibliography comprises three separate orderings. A perl script compiles the multibibliography source. Running "`perl multibibliography.pl` <fn>," instead of `bibtex` (after the 1st-pass "`latex` <fn>" and before the usual 2nd and 3rd passes), generates three `.bbl` files:

"`apalike`" style, sorted alphabetically, by first author's family name,

"`unsrt`" style, in order of first appearance in the document, and with the label adjusted to lead with the sequence number, and also

"`chronological`" style, sorted according to date of publication, as in a timeline.

This functionality is different from both the multibib package,[1] which facilitates having separate bibliographies for each chapter in a monograph, and the multibbl package,[2] which facilitates separating referenced sources by their language [Mori, 2009: 2].

In multibibliography.sty, which should be loaded at the top of any invoking document, the "thebibliography" command is redefined and the "bibliographysequence" and "bibliographytimeline" commands are newly defined, all of which respectively redefine the bibitem command accordingly to generate references in the appropriate format and order. The chronological.bst file in the package, made with the makebst [Daly, 2007: 3] and docstrip utilities and using the merlin.mbs generic bibliography [Daly, 2011: 1], augments the built-in apalike and unsrt styles.

At the end of the document, the multibibliography is rendered thusly:

```
\renewcommand{\bibname}{References
    sorted by name}
  \markboth{References sorted by name
      }{References sorted by name}
\bibliographystyle{apalike}
\addcontentsline{toc}{chapter}{
    References sorted by name}
\bibliography{.bib files}

\clearpage
\renewcommand{\bibname}{References
    sorted by first appearance}
  \markboth{References sorted by
      first appearance}{References
      sorted by first appearance}
\addcontentsline{toc}{chapter}{
    References sorted by appearance}
\bibliographysequence{.bib files}

\clearpage
\renewcommand{\bibname}{References
    sorted chronologically}
  \markboth{References sorted
      chronologically}{References
      sorted chronologically}
\addcontentsline{toc}{chapter}{
    References sorted chronologically}
\bibliographytimeline{.bib files}
```

(For shorter document styles, such as this article, \bibname should be changed above to \refname, and adjustments to the Table of Contents as well as the \clearpages may be elided.)

This multibibliography system is copotentiated by the hypertextual hyperref[3] package. When using them together with an appropriate viewer or browser (such as xdvi, acrobat, or Adobe Reader), clicking an inline citation jumps to the respective entry in one of the reference lists. As illustrated by Fig. 1, the multibibliography inline hyperref hotspot is articulated to allow clicking on

the name, which jumps to the corresponding entry in the alphabetical bibliography;

the index number, which jumps to the respective entry in the sequential bibliography; or

the date, which jumps to the matching entry in the chronological bibliography.

Similarly, cross-references among the respective sub-bibliographies are also hyperlinked, although from the labels, and not the bibitem bodies of the citations. The "[backref=page]" hyperref extension[4] is also compatible, generating the familiar and useful back-references in all three subbibliographies: lists of clickable page number links associated with each entry in the bibliography pointing back to the respective citations (excepting those generated by nocites). The generation of these back-references, indicated by the hollow arrowheads in Fig. 1, represents "closing the loop" on the fully crossed relation set.

In the future, the date should be articulated to add the month to the sort.label in the presort FUNCTION in the chronological.bst file, since it isn't one of the built-in keys of the makebst package [Markey, 2009: 6], as the merlin system didn't anticipate such fine-grained sortings. As [Markey, 2009: 6, p. 13] observes, the month is somewhat problematic, since it is indicated by a character string, but is really an ordinal. If built-in macros ("jan", "feb", etc.) are used, they can be easily mapped to months and used for sorting, but if, as is often the case, the field is reinterpreted to mean date (bimonthly publications indicated by something like month = "March & April", quarterly dates as month = "Autumn", etc.), this scheme will not easily generalize.

We have not yet experimented with combining this package with other bibliographic packages [Patashnik, 1998: 4] such as natbib[5] or chapterbib[6] [Kopka and Daly, 2003: 5, p. 217–221].

[1] www.ctan.org/pkg/multibib
[2] www.ctan.org/pkg/multibbl
[3] www.ctan.org/pkg/hyperref
[4] www.tug.org/applications/hyperref/manual.html
[5] www.ctan.org/pkg/natbib
[6] www.ctan.org/pkg/chapterbib

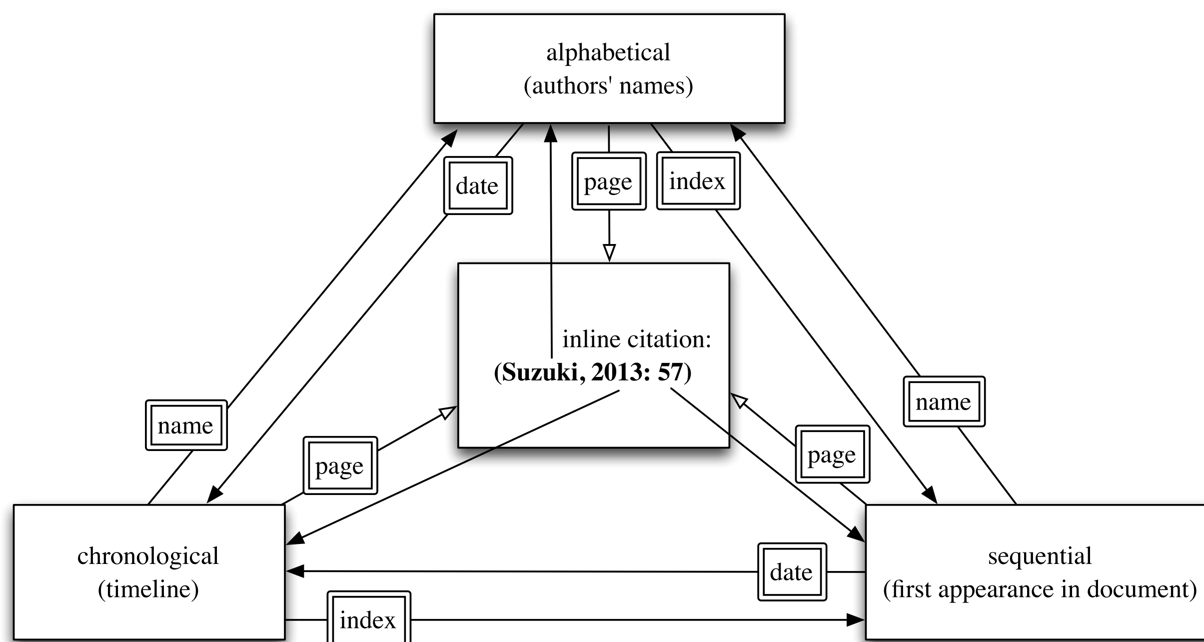


Figure 1: Hyperreferential links across document and among the multibibliographies: Each inline citation, exemplified by the block in the center, is linked to references in subbibliographies, which are cross-linked to each other and can also be linked back to the inline callout. Hollow arrowheads represent links provided by `hyperref`'s `backref`; solid arrowheads represent links provided by the `multibibliography` package.

3 Implications

The extended inline citation style was designed for the multibibliography, but can be deployed and is useful even without it. The bibliographic dilation is perhaps more appropriate or at least more appealing for electronic dissemination, as traditional print-based publishers might resent the cost of extra pages. The fully crossed hyperreferential links are a convenient way of establishing the context of references, seamlessly expressing citations' appearances in the document and in time.

Bibliographic subsections might be sorted by other ad-hoc keys. Maybe the three "slices" through the bibliographic database that we have organized suffice for most ordinary publishing, but presumably someone could make even more styles of bibliographic lists, corresponding to special purposes, sorted by attributes such as number of authors, number of pages, conference or journal, location, etc. For simple examples, a subbibliography near the end of an art book could be sorted by name of artist considered by each monograph, or music books could have bibliographies sorted by name of performing group or family or given name of artist. Of course all monographs about each artist would be grouped together in that subbibliography, and each citation would include page numbers for each call-out within the text.

Such extended subbibliographies both represent and also re-present references, showing them in fresh and useful settings. There are two related activities encouraged by such recontextualization:

- looking up a particular entry (including page call-outs), and
- exploiting "locality of reference," so that other related sources are likely to be nearby.

Such lists could be explicitly sequenced by an author, but only painstakingly. However, such concordances would lack the automatic back references to the call-outs. A modern, flexible scheme enables various presentations of bibliographic information, so that each references subsection acts as a kind of special index, but with granularity not at the topic level, but at the document level.

The philosophy is to leverage the power of hyperreferential idioms to augment reading by considering a document as a special kind of database that is indexed in appropriate dimensions, especially including the name–value pairs in its associated bibliographic information (such as that captured by

BibTeX files) plus derived information available after compilation (such as sequence number and appearance location).

It is our hope that old-fashioned conventions, established in the context of technological restrictions that have now been overcome, may be relaxed. We anticipate that future schemes will allow dynamic reordering, as if the bibliography were a spreadsheet-like database, which of course it is. We find this multidimensional presentation useful, are adopting it at the first author's university as a recommended style for masters theses and doctoral dissertations, and hereby encourage other institutions to emulate this innovation, especially for extended works such as monographs and books.

References sorted by name

[Daly, 2007: 3] Daly, P. W. (2007). Customizing bibliographic style files. `http://mirror.ctan.org/macros/latex/contrib/custom-bib/makebst.pdf`. 1002

[Daly, 2011: 1] Daly, P. W. (2011). A Master Bibliographic Style File for numerical, author–year, multilingual applications. `http://mirror.hmc.edu/ctan/macros/latex/contrib/custom-bib/merlin.pdf`, `http://ftp.jaist.ac.jp/pub/CTAN/macros/latex/contrib/custom-bib/merlin.pdf`, v. 4.33. 1001, 1002

[Kopka and Daly, 2003: 5] Kopka, H. and Daly, P. W. (2003). *Guide to LaTeX*. Addison-Wesley Professional, 4th edition. 1002

[Markey, 2009: 6] Markey, N. (2009). Tame the BeaST: The B to X of BibTeX. `ftp://ftp.tex.ac.uk/tex-archive/info/bibtex/tamethebeast/ttb_en.pdf`, v. 1.4. 1002

[Mori, 2009: 2] Mori, L. F. (2009). Managing bibliographies with LaTeX. *TUG: TeX Users Group Meeting*, 30(1):36–48. 1001

[Patashnik, 1998: 4] Patashnik, O. (1998). BibTeXing. `http://mirror.ctan.org/biblio/bibtex/contrib/doc/btxdoc.pdf`. 1002

References sorted by appearance

[1: Daly, 2011] Patrick W. Daly. A Master Bibliographic Style File for numerical, author–year, multilingual applications, October 2011. `http://mirror.hmc.edu/ctan/macros/latex/contrib/custom-bib/merlin.pdf`, `http://ftp.jaist.ac.jp/pub/CTAN/macros/latex/contrib/custom-bib/merlin.pdf`, v. 4.33. 1001, 1002

[2: Mori, 2009] Lapo F. Mori. Managing bibliographies with LaTeX. *TUG: TeX Users Group Meeting*, 30(1):36–48, 2009. 1001

[3: Daly, 2007] Patrick W. Daly. Customizing bibliographic style files, 2007. `http://mirror.ctan.org/macros/latex/contrib/custom-bib/makebst.pdf`. 1002

[4: Patashnik, 1998] Oren Patashnik. BibTeXing, 1998. `http://mirror.ctan.org/biblio/bibtex/contrib/doc/btxdoc.pdf`. 1002

[5: Kopka and Daly, 2003] Helmut Kopka and Patrick W. Daly. *Guide to LaTeX*. Addison-Wesley Professional, 4th edition, 2003. 1002

[6: Markey, 2009] Nicolas Markey. Tame the BeaST: The B to X of BibTeX, October 2009. `ftp://ftp.tex.ac.uk/tex-archive/info/bibtex/tamethebeast/ttb_en.pdf`, v. 1.4. 1002

References sorted by year

[Patashnik, 1998: 4] Patashnik, O. BibTeXing. 1998. `http://mirror.ctan.org/biblio/bibtex/contrib/doc/btxdoc.pdf`. 1002

[Kopka and Daly, 2003: 5] Kopka, H. and Daly, P. W. *Guide to LaTeX*. Addison-Wesley Professional, 4th edition, 2003. ISBN 0-321-17385-6. 1002

[Daly, 2007: 3] Daly, P. W. Customizing bibliographic style files. 2007. `http://mirror.ctan.org/macros/latex/contrib/custom-bib/makebst.pdf`. 1002

[Markey, 2009: 6] Markey, N. Tame the BeaST: The B to X of BibTeX. 2009. `ftp://ftp.tex.ac.uk/tex-archive/info/bibtex/tamethebeast/ttb_en.pdf`, v. 1.4. 1002

[Mori, 2009: 2] Mori, L. F. Managing bibliographies with LaTeX. *TUG: TeX Users Group Meeting*, 30(1):36–48, 2009. 1001

[Daly, 2011: 1] Daly, P. W. A Master Bibliographic Style File for numerical, author–year, multilingual applications. 2011. `http://mirror.hmc.edu/ctan/macros/latex/contrib/custom-bib/merlin.pdf`, `http://ftp.jaist.ac.jp/pub/CTAN/macros/latex/contrib/custom-bib/merlin.pdf`, v. 4.33. 1001, 1002

Michael Cohen, Yannis Haralambous and Boris Veytsman

◇ Michael Cohen
Spatial Media Group, Computer Arts Lab.
University of Aizu
Aizu-Wakamatsu, Fukushima 965-8580
Japan
mcohen@u-aizu.ac.jp
www.u-aizu.ac.jp/~mcohen

◇ Yannis Haralambous
Département Informatique Télécom Bretagne
Technopôle de Brest Iroise, CS 83818
29238 Brest Cedex 3
France
yannis.haralambous@telecom-bretagne.eu
international.telecom-bretagne.eu/welcome/studies/msc/professors/haralambous.php

◇ Boris Veytsman
Systems Biology School and Computational Materials Science Center
MS 6A2
George Mason University
Fairfax, VA 22030
USA
borisv@lk.net
borisv.lk.net

Sponsors

Graduate School of Mathematical Sciences, the University of Tokyo （東京大学大学院数理科学研究科）

SANBI Printing Co., Ltd.（三美印刷株式会社）

Gijutsu-Hyohron Co., Ltd.（株式会社技術評論社）

Tokyo Educational Institute Co., Ltd. (Tetsuryokukai)（株式会社東京教育研（鉄緑会））

Green Cherry Ltd.（株式会社 Green Cherry）

PLAIN corporation（株式会社プレイン）

Livretech Co., Ltd.（株式会社リーブルテック）

Top Studio Co., Ltd.（株式会社トップスタジオ）

ULS & Company（株式会社ウルス）

Tatsu-zine Publishing Inc.（株式会社達人出版会）

Ohmsha, Ltd.（株式会社オーム社）

Kato Bunmeisha Co., Ltd.（株式会社加藤文明社印刷所）

Fujiwara Printing Co., Ltd.（藤原印刷株式会社）

Saiensu-sha Co., Ltd.（株式会社サイエンス社）

Suurikougaku-sha Co., Ltd.（株式会社数理工学社）

Enishi Tech Inc.（株式会社えにしテック）

Maruzen Publishing Co., Ltd.（丸善出版株式会社）

Dainippon Hourei Printing inc.（大日本法令印刷株式会社）

Gravel Road Inc.（株式会社グラベルロード）

the TEX Users Group

Korean TEX Society

DANTE e.V.

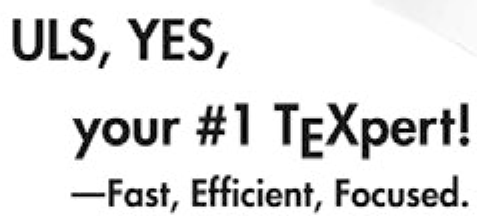
Thomas Bietenhader ♦ Kosakai Eiichiro（小酒井 英一郎）♦ KIEDA Yuwsuke（木枝 祐介）♦ SHIBATA Mitsuya（柴田 充也）♦ FUJIMURA Yukitoshi（藤村 行俊）♦ NARIAI Kyoji（成相 恭二）♦ Hiroki Kanou（狩野 宏樹）♦ SHIKANO Keiichiro（鹿野 桂一郎）

and a lot of anonymous sponsors.

Thanks to all!

ご協力頂いた全ての方へ感謝いたします.

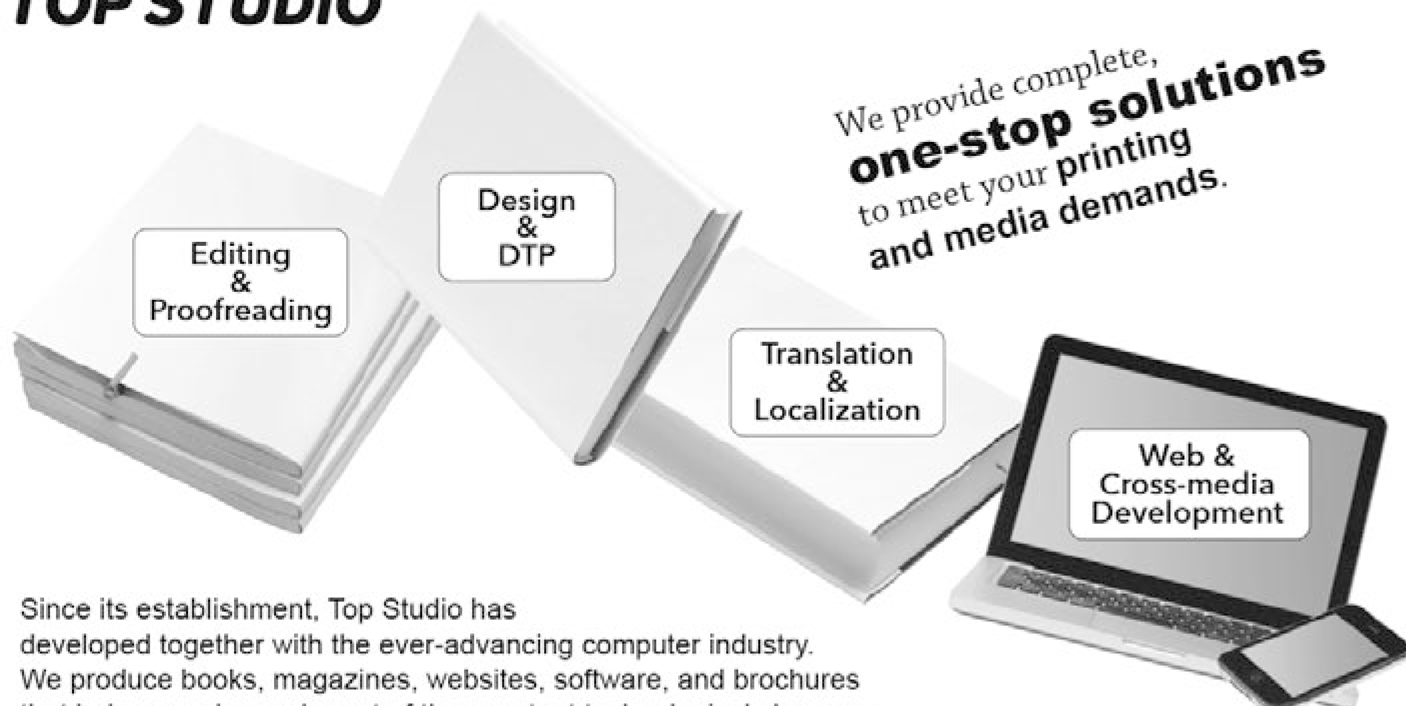
Conference committee:
ABE Noriyuki (Hokkaido Univ.) • Karl BERRY (TUG) • HONDA Tomoaki (SANBI Printing) • ICHII Shingo (the Univ. of Tokyo) • KUROKI Yusuke • OKUMURA Haruhiko (Mie Univ.) • Steve PETER (TUG) • Norbert PREINING (JAIST) • TAKAHASHI Masayoshi (Tatsu-zine Publishing) • YAMAMOTO Munehiro (Green Cherry)



編集・出版プロダクション
TEXコンサルティング
株式会社ウルス http://www.uls.co.jp/

TEX USERS GROUP

TOP STUDIO



Since its establishment, Top Studio has
developed together with the ever-advancing computer industry.
We produce books, magazines, websites, software, and brochures
that help users keep abreast of the constant technological changes.
We pride ourselves on being your one-stop solution – from planning to realization,
from creating the drafts to delivering the finished products.
We not only help our clients achieve their needs, but we also get involved.

Contact
TEL. +81 3 5457 7191 **FAX** +81 3 5457 7195 **http://www.topstudio.co.jp/** **Top Studio Co., Ltd.**

The Manga Guide Series

The Manga Guide to **Statistics**
The Manga Guide to **Databases**
The Manga Guide to **Calculus**
The Manga Guide to **Fourier Analysis**
The Manga Guide to **Physics**
The Manga Guide to **Electricity**
The Manga Guide to **Molecular Biology**
The Manga Guide to **The Theory of Relativity**
The Manga Guide to **The Universe**
(many more titles available)

For English translated editions：www.nostarch.com/manga/

Ohmsha, Ltd.
3-1 Kanda Nishikicho
Chiyoda-ku
Tokyo 101-8460 Japan

Tel ：+81-3-3233-2425 e-mail：kaigaika@ohmsha.co.jp
Fax：+81-3-3233-2426 URL ：http://www.ohmsha.co.jp/

例えば…、
生徒一人ひとりの
弱点を克服するための
個別教材をつくりたい。

リーブルテックの
必要な時に 必要な物を 必要なだけ
オンデマンド印刷

★ 高品質印刷でカラー・モノクロに対応
★ 少部数印刷対応で在庫レス
★ データからダイレクトに印刷
★ 個別対応型の印刷が可能

◎編集・デザインから発送までトータルにサポート◎

編集・デザイン → プリプレス → 印刷 → 製本 → 発送

業務内容：教育用図書並びに一般図書・商業印刷物の編集・デザイン、印刷、製本及び梱包、発送、保管、DTP、「TeX」組版、データベース構築など

教育の印刷・信頼の技術
株式会社 リーブルテック
TEL：03-3927-6411（代）
http://www.livretech.co.jp
〒114-0004
東京都北区堀船1-23-31

JAPAN COLOR

TEX ×

→

私たち加藤文明社は、あなたの本づくりをサポートいたします。

クラス・スタイル設計／TEX 組版・レイアウト／TEX 入力業務／図版作成
事典・便覧などの多執筆者書籍のワークフロー構築・業務の取りまとめ／専門書籍や学術論文雑誌の編集代行
学術雑誌のオンライン化（J-STAGE にも対応）／TEX データから、または TEX データへのデータ変換

文書処理に関するあらゆるご相談から、印刷・製本・発送までお受けいたします。

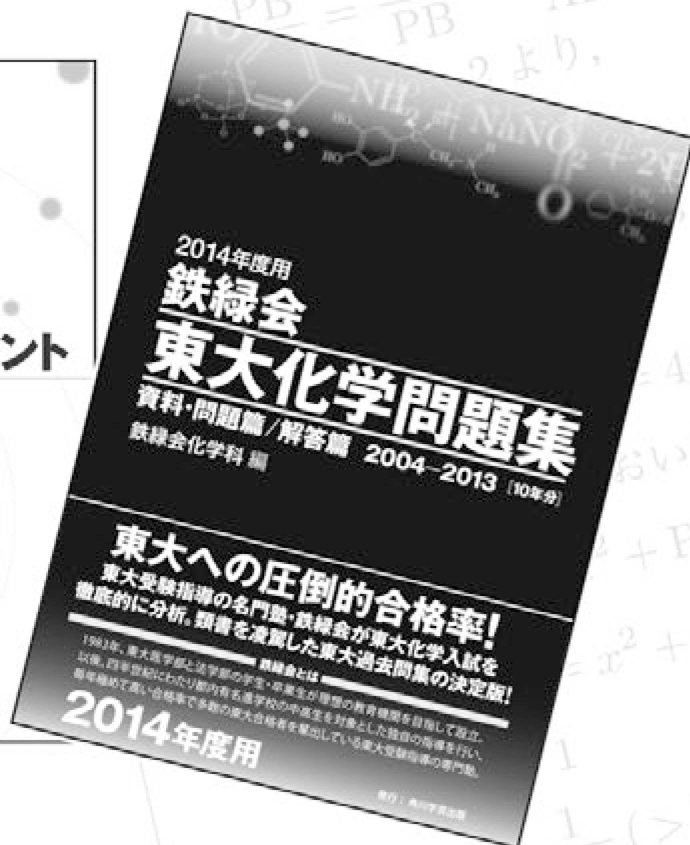
株式会社
加藤文明社
〒101–0061 東京都千代田区三崎町2–15–6
Tel: 03–3261–8281 Fax: 03–3261–8292
http://www.bunmeisha.co.jp

Green Cherry

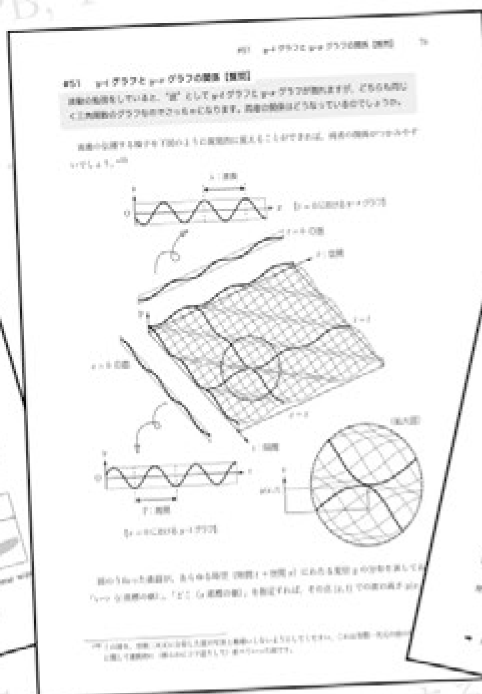
Just Another TeXnicians

和の「おもてなしの心」をこめた日本語本作りや
「人間くさい操作性」を大事にした
直感的な操作ができるソフトウェア作りをはじめ、
アナログとデジタルのどちらも大事にした
ものづくりやサービスを心がけております。
「どうすればどんな人でも
知的な情報を支障なく利用できるだろうか？」、
そのために私たちができることを考えて、
仲間どうしと幅広く情報を共有し、
そして実践していきます。

Green Cherry Ltd.
info@greencherry.jp
http://GreenCherry.jp/



TEX — we support it, it supports us.



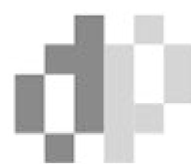
Tokyo Educational Institute

Tetsuryokukai

http://www.tetsuryokukai.co.jp/

株式会社**技術評論社**は
信頼と実績のある
書籍・雑誌を作り続ける**専門書出版社**です。

現在は、紙の書籍・雑誌に加えて
"未来の普通"に向けた
電子出版・電子書籍づくりにも
積極的に取り組んでいます。


GIHYO
Digital Publishing

技術評論社　〒162-0846　東京都新宿区市谷左内町 21-13
TEL 03-3513-6158　FAX 03-3513-6151　URL http://gihyo.jp

紙メディアと電子メディアの融合、私たちはお客様の課題にお応えするソリューションを提供します。

出版印刷が三美印刷の原点です。

印刷の発明によって誕生した「本」という媒体が大量情報の保存と知識の伝播を可能にし、それによって科学、技術、文化は飛躍的に発展しました。

今、検索、保存、移動などで多くの利点を有する電子書籍に期待が高まっていますが、五感（味覚は除きますが）に訴える力を持ち、愛着が生まれるのは紙の本です。脳に良い刺激を与えるとも言われています。私たちは紙の本がこれまで果してきた役割と、今後果たしていく役割の重要性を確信し、これからもより良い本作りに取り組んでまいります。

特に数式組版は理工系の出版社、学会から高いご評価をいただき、自動組版技術は医学系出版社より高いご評価をいただいています。

また印刷においては菊半5色機から菊全両面8色機まで37胴を設備、各機に検査装置を設備し品質に万全を期しております。

ライブラリ

常識を変えるPUR製本

PURの接着力は現行EVRのほぼ倍、耐熱、耐寒性に優れ、インキによる劣化もありません。PURを使えば、丈夫で長持ちする本が、高価なあかがりでなく、安価な無線トジで作れます。

ノドの奥まで開いても壊れない、60度を超える暑さにも、マイナス20度の寒さにも耐えるPURは今後製本の主流となることが予想されています。

三美印刷は出版社様への一層のお役立ちを目指し、この「理想の無線トジ─PUR製本ライン」を自社印刷工場内に構築しました。

三美印刷が提供する組版、印刷、製本技術にご期待ください。

【得意とする技術】
データベース・XML を利用した自動組版
Tex による数式組版
プログラムを利用する用字用語の統一
情報処理技術
ルビ振り技術
PUR 製本

■主な製品
理工学系図書
医学書
辞典・事典
論文誌・学会誌
絵本
シラバス

255/199

三美印刷では年間2000論文を超える$\TeX$原稿を処理しています。
$\TeX$で本を作成したい、これから$\TeX$を使って本を書きたい、コストを抑えて本を作りたい、
などの御希望に対し、専門スタッフが対応しております。
お気軽に是非ご相談ください。

SANBI
PRINTING Co.,Ltd.

三美印刷株式会社

〒116-0013 東京都荒川区西日暮里 5-9-8
TEL.03-3803-3131 ㈹ FAX.03-5604-7039
E-mail sanbi@sanbi.co.jp
URL http://www.sanbi.co.jp