MAPS

NUMMER 38 • VOORJAAR 2009

REDACTIE

Taco Hoekwater, hoofdredacteur
Wybo Dekker
Frans Goddijn

NTG NEDERLANDSTALIGE TEX GEBRUIKERSGROEP

De **Nederlandstalige TEX Gebruikersgroep (NTG)** is een vereniging die tot doel heeft de kennis en het gebruik van TEX te bevorderen. De NTG fungeert als een forum voor nieuwe ontwikkelingen met betrekking tot computergebaseerde documentopmaak in het algemeen en de ontwikkeling van 'TEX and friends' in het bijzonder. De doelstellingen probeert de NTG te realiseren door onder meer het uitwisselen van informatie, het organiseren van conferenties en symposia met betrekking tot TEX en daarmee verwante programmatuur.
De NTG biedt haar leden ondermeer:

- ☐ Tweemaal per jaar een NTG-bijeenkomst.
- ☐ Het NTG-tijdschrift MAPS.
- ☐ De 'TEX Live'-distributie op DVD/CDROM inclusief de complete CTAN software-archieven.
- ☐ Verschillende discussielijsten (mailing lists) over TEX-gerelateerde onderwerpen, zowel voor beginners als gevorderden, algemeen en specialistisch.
- ☐ De FTP server `ftp.ntg.nl` waarop vele honderden megabytes aan algemeen te gebruiken 'TEX-producten' staan.
- ☐ De WWW server `www.ntg.nl` waarop algemene informatie staat over de NTG, bijeenkomsten, publicaties en links naar andere TEX sites.
- ☐ Korting op (buitenlandse) TEX-conferenties en -cursussen en op het lidmaatschap van andere TEX-gebruikersgroepen.

Lid worden kan door overmaking van de verschuldigde contributie naar de NTG-giro (zie links); vermeld IBAN- zowel als SWIFT/BIC-code en selecteer shared cost. Daarnaast dient via `www.ntg.nl` een informatieformulier te worden ingevuld. Zonodig kan ook een papieren formulier bij het secretariaat worden opgevraagd.
De contributie bedraagt € 40; voor studenten geldt een tarief van € 20. Dit geeft alle lidmaatschapsvoordelen maar *geen stemrecht*. Een bewijs van inschrijving is vereist. Een gecombineerd NTG/TUG-lidmaatschap levert een korting van 10% op beide contributies op. De prijs in euro's wordt bepaald door de dollarkoers aan het begin van het jaar. De ongekorte TUG-contributie is momenteel $65.

MAPS bijdragen kunt u opsturen naar `maps@ntg.nl`, bij voorkeur in LaTEX- of ConTEXt formaat. Bijdragen op alle niveaus van expertise zijn welkom.

Productie. De Maps wordt gezet met behulp van een LaTEX class file en een ConTEXt module. Het pdf bestand voor de drukker wordt aangemaakt met behulp van pdftex 1.40.9 en luatex 0.40.1 draaiend onder Linux 2.6. De gebruikte fonts zijn Bitstream Charter, schreefloze en niet-proportionele fonts uit de Latin Modern collectie, en de Euler wiskunde fonts, alle vrij beschikbaar.

TEX is een door professor Donald E. Knuth ontwikkelde 'opmaaktaal' voor het letterzetten van documenten, een documentopmaaksysteem. Met TEX is het mogelijk om kwalitatief hoogstaand drukwerk te vervaardigen. Het is eveneens zeer geschikt voor formules in mathematische teksten.
Er is een aantal op TEX gebaseerde producten, waarmee ook de logische structuur van een document beschreven kan worden, met behoud van de letterzetmogelijkheden van TEX. Voorbeelden zijn LaTEX van Leslie Lamport, AMS-TEX van Michael Spivak, en ConTEXt van Hans Hagen.

Inhoudsopgave

Redactioneel

Dit jaar is een EuroTEX jaar voor de NTG, en deze EuroTEX conferentie is tegelijkertijd ook nog eens de ConTEXt gebruikers bijeenkomst voor dit jaar. U zult hopelijk begrijpen dat ondergetekende als één van de hoofd-organisators van dit evenement het daar momenteel best druk mee heeft, en daardoor heb ik wat minder tijd kunnen besteden aan vulling zoeken voor de Maps.

Het gevolg is een Maps die weliswaar niet de dunste ooit is, maar toch aardig in de buurt van het record komt (Maps 35 was maar 54 bladzijden). Daar staat tegenover dat u in het najaar een flinke pil tegemoet kunt zien: Maps 39 zal het proceedings-issue van de komende conferentie zijn, en het programma van de conferentie is zelfs nu al best flink gevuld. Ik ben dan ook vol vertrouwen dat een extra dik Maps issue nog net voor de kerstdagen in eenieders brievenbus zal vallen.

Goed, wat staat er dan wel in deze Maps? Allereerst nog een laatste aanmoediging om te registreren voor de conferentie. De inschrijving sluit definitief op **7 Juli**, dus mocht u willen komen, wacht dan niet te lang meer met het formulier invullen. En, ten overvloede, zowel bestuur als organisatiecomité bevelen registreren natuurlijk van harte aan!

Maar genoeg nu over EuroTEX. De eerste vier artikelen in deze Maps gaan allemaal op de één of andere manier over de combinatie van OpenType fonts en wiskundig zetwerk in TEX.

Het eerste van de twee artikelen van *Ulrik Vieth* stelt de vraag 'Do we need a 'Cork' math encoding?' Dit is een herdruk van het artikel dat ook al verscheen in de proceedings van de tug bijeenkomst van vorig jaar in Cork. Het verschijnt hier opnieuw omdat het een goede inleiding geeft over recente ontwikkelingen op math font gebied.

Ulrik's tweede artikel 'OpenType Math Illuminated' bespreekt in detail wat er allemaal mogelijk is met nieuwe OpenType Math fonts. De titel van dit artikel is een verwijzing naar 'Appendix G illuminated' van *Bogusław Jackowski* dat we afdrukten in Maps 34.

'Math in LuaTEX 0.40' van *Taco Hoekwater* documenteert allerhande uitbreidingen in LuaTEX die van doen hebben met wiskundig zetwerk. Uiteraard gaat het daarbij vooral over uitbreidigen die van toepassing zijn op het gebruik van OpenType fonts, maar het artikel begint met een setje kleine uitbreidingen met een meer algemene inslag.

Het laatste artikel over wiskunde en OpenType is van de hand van *Hans Hagen*. Hans' 'Unicode Math in ConTEXt' vertelt over de inpassing van de LuaTEX uitbreidingen uit het voorgaande artikel binnen de nieuwe ConTEXt MkIV.

Meteen hierna volgt nog een artikel over LuaTEX: 'LuaTEX – Halfway' geeft de tussenstand van het LuaTEX project en geeft vast een voorzichtige vooruitblik op wat nog komen gaat.

Iets heel anders is 'TEX Programming: The past, the present, and the future' door *Aditya Mahajan*. Nou ja, heel anders ... het gaat nog steeds grotendeels over LuaTEX, alleen gaat het nu over hoe je het typesetten van tabellen kunt programmeren in TEX macros of Lua functies.

In Maps 34 stond een artikel met daarin de resultaten van de 'Pearls' sessie van BachoTEX 2006. In deze Maps vind je de resultaten van de sessie van 2009. Wel onder de wat realistischer titel 'TEX beauties and oddities', maar nog steeds verzameld door Paweł Jackowski.

En dan zijn we er al weer zo goed als doorheen. Het laatste artikel is van *Siep Kroonenberg* en de titel is 'Doe-het-zelf presentaties'. In dat artikel demonstreert Siep hoe je met LaTEX ook *zonder* het 'beamer' pakket presentaties kunt maken.

Zoals gezegd, een wat mager Mapsje, maar toch veel leesplezier toegewenst, en hopelijk tot ziens in Den Haag!

Taco Hoekwater

EuroTEX 2009

3rd ConTEXt Meeting

The Dutch TEX Language User Group and the ConTEXt task force are pleased to invite you to the combined EuroTEX 2009 conference and third international ConTEXt meeting.

August 31 – September 4 2009, The Hague

Call for Papers

As usual, proposals for presentations and workshops are welcomed on just about any topic of interest to TEX users, but the conference focus will be on

Educational uses of TEX

such as manuals, courseware and college presentations, so we especially welcome proposals on subjects in those fields.

The language of the conference is English. Please send abstracts and proposals in plain text or TEX format to the conference committee at `eurotex@ntg.nl`.

Registration

The conference is made possible by the Netherlands Defence Academy (NLDA) that graciously invited us to their facilities, including the on-site hotel.

http://www.ntg.nl/EuroTeX2009/

Registration will close on July 7, so don't wait too long!

Do we need a ‘Cork’ math font encoding?*

Abstract
The city of Cork has become widely known in the TEX community, ever since it gave name to an encoding developed at the European TEX conference of 1990. The ‘Cork’ encoding, as it became known, was the first example of an 8-bit text font encoding that appeared after the release of TEX 3.0, and was later followed by a number of other encodings based on similar design principles.

As of today, the ‘Cork’ encoding represents only one out of several possible choices of 8-bit subsets from a much larger repertoire of glyphs provided in fonts such as Latin Modern or TEX Gyre. Moreover, recent developments of new TEX engines are making it possible to take advantage of OpenType font technology directly, largely eliminating the need for 8-bit font encodings altogether.

During the entire time since 1990 math fonts have always been lagging behind the developments in text fonts. While the need for new math font encodings was recognized early on and while several encoding proposals have been discussed, none of them ever reached production quality or became widely used.

In this paper, we review the situation of math fonts as of 2008, especially in view of recent developments of Unicode and OpenType math fonts such as the STIX fonts or Cambria Math. In particular, we try to answer the question whether a ‘Cork’ math font encoding is still needed or whether Unicode and OpenType might eliminate the need for TEX-specific math font encodings.

History and development of text fonts

The ‘Cork’ encoding

When the 5th European TEX conference was held in Cork in the summer of 1990, the TEX community was undergoing a major transition phase. TEX 3.0 had just been released that year, making it possible to switch from 7-bit to 8-bit font encodings and to support hyphenation for multiple languages.

Since the ability to properly typeset and hyphenate accented languages strongly depended on overcoming the previous limitations, European TEX users wanted to take advantage of the new features and started to work on new font encodings [1, 2, 3]. As a result, they came up with an encoding that became widely known as the ‘Cork’ encoding, named after the site of the conference [4].

The informal encoding name ‘Cork’ stayed in use for many years, even after LaTEX 2_ε and NFSS2 introduced a system of formal encoding names in 1993–94, assigning `OT`*n* for 7-bit old text encodings, `T`*n* for 8-bit standard text encodings, and `L`*n* for local or non-standard encodings [5]. The ‘Cork’ encoding was the first example of a standard 8-bit text font and thus became the `T1` encoding.

While the ‘Cork’ encoding was certainly an important achievement, it also introduced some novel features that may have seemed like a good idea at that time but would be seen as shortcomings or problems from today’s point of view, after nearly two decades of experience with font encodings.

In retrospect, the ‘Cork’ encoding represents a typical example of the TEX-specific way of doing things of the early 1990s without much regard for standards or technologies outside the TEX world.

Instead of following established standards, such as using ISO Latin 1 or 2 or some extended versions for Western and Eastern European languages, the ‘Cork’ encoding tried to support as many languages as possible in a single font encoding, filling the 8-bit font table to the limit with accented characters at the expense of symbols. Since there was no more room left in the font table, typesetting symbols at first had to be taken from the old 7-bit fonts, until a supplementary text symbol `TS1` encoding [6] was introduced in 1995 to fill the gap.

When it came to implementing the `T1` and `TS1` encodings for PostScript fonts, it turned out that the encodings were designed without taking into account the range of glyphs commonly available in standard PostScript fonts.

Both font encodings could only be partially implemented with glyphs from the real font, while the remaining slots either had to be faked with virtual fonts or remain unavailable. At the same time, none of the encodings provided access to the full set of available glyphs from the real font.

*First published in *TUGboat* **29**:3 (2008), pp.426–434. Reprinted with permission.

Alternatives to the 'Cork' encoding

As an alternative to using the `T1` and `TS1` encodings for PostScript fonts, the `TeXnANSI` or `LY1` encoding was proposed [7], which was designed to provide access to the full range of commonly available symbols (similar to the `TeXBase1` encoding), but also matched the layout of the `OT1` encoding in the lower half, so that it could be used as drop-in replacement without any need for virtual fonts.

In addition to that, a number of non-standard encodings have come into use as local alternatives to the 'Cork' encoding, such as the Polish `QX`, the Czech `CS`, and the Lithuanian `L7X` encoding, each of them trying to provide better solutions for the needs of specific languages.

In summary, the 'Cork' encoding as the first example of an 8-bit text encoding (`T1`) was not only followed by additional encodings based on the same design principles for other languages (`T`*n*), but also supplemented by a text symbol encoding (`TS1`) and complemented by a variety of local or non-standard encodings (`LY1`, `QX`, `CS`, etc.).

As became clear over time, the original goal of the 'Cork' encoding of providing a single standard encoding for as many languages as possible couldn't possibly be achieved within the limits of 8-bit fonts, simply because there are far too many languages and symbols to consider, even when limiting the scope to Latin and possibly Cyrillic or Greek.

Recent developments of text fonts

Unicode support in new TEX fonts

It was only in recent years that the development of the Latin Modern [8, 9, 10] and TEX Gyre fonts [11, 12] has provided a consistent implementation for all the many choices of encodings.

As of today, the 'Cork' encoding represents only one out of several possible 8-bit subsets taken from a much larger repertoire of glyphs. The full set of glyphs, however, can be accessed only when moving beyond the limits of 8-bit fonts towards Unicode and OpenType font technology.

Unicode support in new TEX engines

As we are approaching the TUG 2008 conference at Cork, the TEX community is again undergoing a major transition phase. While TEX itself remains frozen and stable, a number of important developments have been going on in recent years.

Starting with the development of PDFTEX since the late 1990s the use of PDF output and scalable PostScript or TrueType fonts has largely replaced the use of DVI output and bitmap PK fonts.

Followed by the ongoing development of XƎTEX and LuaTEX in recent years the use of Unicode and OpenType font technology is also starting to replace the use of 8-bit font encodings as well as traditional PostScript or TrueType font formats.

Putting everything together, the development of new fonts and new TEX engines in recent years has enabled the TEX community to catch up with developments of font technology in the publishing industry and to prepare for the future.

The only thing still missing (besides finishing the ongoing development work) is the development of support for Unicode math in the new TEX engines and the development of OpenType math fonts for Latin Modern and TEX Gyre.

History and development of math fonts

When TEX was first developed in 1977–78, the 7-bit font encodings for text fonts and math fonts were developed simultaneously, since both of them were needed for typesetting mathematical textbooks like *The Art of Computer Programming*.

When TEX 3.0 made it possible to switch from 7-bit to 8-bit font encodings, it was the text fonts driving these new developments while the math fonts remained largely unchanged.

As a result, the development of math fonts has been lagging behind the corresponding text fonts for nearly two decades now, ever since the development of the 'Cork' encoding started in 1990.

In principle, a general need for new math fonts was recognized early on: When the first implementations of 'Cork' encoded text fonts became available, it was soon discovered that the new 8-bit text fonts couldn't fully replace the old 7-bit text fonts without resolving the inter-dependencies between text and math fonts. In practice, however, nothing much happened since there was no pressing need.

The 'Aston' proposal

The first bit of progress was made in the summer of 1993, when the LaTEX3 Project and some TEX users group sponsored a research student to work on math font encodings for a few months.

As a result, a proposal for the general layout of new 8-bit math font encodings was developed and presented at TUG 1993 at Aston University [13]. Unlike the 'Cork' encoding, which became widely known, this 'Aston' proposal was known only to some insiders and went largely unnoticed.

After only a few months of activity in 1993 the project mailing list went silent and nothing further happened for several years, even after a detailed report was published as a LaTEX3 Project Report [14].

The 'newmath' prototype

The next bit of progress was made in 1997–98, when the ideas of the 'Aston' proposal were taken up again and work on an implementation was started.

This time, instead of just discussing ideas or preparing research documents, the project focussed on developing a prototype implementation of new math fonts for several font families using a mixture of MetaFont and `fontinst` work [15].

When the results of the project were presented at the EuroTEX 1998 conference [16], the project was making good progress, although the results were still very preliminary and far from ready for production.

Unfortunately, the project then came to a halt soon after the conference when other activities came to the forefront and changed the scope and direction of the project [17, 18].

Before the conference, the goal of the project had been to develop a set of 8-bit math font encodings for use with traditional TEX engines (within the constraints of 16 families of 256 glyphs) and also to provide some example implementations by means of reencoding and enhancing existing font sets.

After the conference, that goal was set aside and put on hold for an indefinite time by the efforts to bring math into Unicode.

Recent developments of math fonts

Unicode math and the stix fonts

While the efforts to bring math into Unicode were certainly very important, they also brought along a lot of baggage in the form of a very large number of additional symbols, making it much more work to provide a reasonably complete implementation and nearly impossible to encode all those symbols within the constraints of traditional TEX engines.

In the end, the Unicode math efforts continued over several years until the symbols were accepted [19, 20] and several more years until an implementation of a Unicode math font was commissioned [21] by a consortium of scientific and technical publishers, known as the STIX Project.

When the first beta-test release of the so-called STIX fonts [22] finally became available in late 2007, nearly a decade had passed without making progress on math font encodings for TEX.

While the STIX fonts provide all the building blocks of Unicode math symbols, they are still lacking TEX support and may yet have to be repackaged in a different way to turn them into a usable font for use with TEX or other systems.

Despite the progress on providing the Unicode math symbols, the question of how to encode all the many Unicode math symbols in a set of 8-bit font encodings for use with traditional TEX engines still remains unresolved. Most likely, only a subset of the most commonly used symbols could be made available in a set of 8-bit fonts, whereas the full range of symbols would be available only when moving to Unicode and OpenType font technology.

OpenType math in ms Office 2007

While the TEX community and the consortium of scientific publishers were patiently awaiting the release of the STIX fonts before reconsidering the topic of math font encodings, outside developments have continued to move on. In particular, Microsoft has moved ahead and has implemented its own support for Unicode math in Office 2007.

They did so by adding support for math typesetting in OpenType font technology [23, 24] and by commissioning the design of the Cambria Math font as an implementation of an OpenType math font [25, 26, 27]. In addition, they have also adopted an input language called 'linear math' [28], which is strongly based on TEX concepts.

While OpenType math is officially still considered experimental and not yet part of the OpenType specification [29], it is already a *de facto* standard, not only because it has been deployed to millions of installations of Office 2007, but also because it has already been adopted by other projects, such as the FontForge font editor [30] and independent font designs such as Asana Math [31].

In addition, the next release of the STIX fonts scheduled for the summer of 2008 is also expected to include support for OpenType math.

OpenType math in new TEX engines

At the time of writing, current development versions of XƎTEX have added some (limited) support for OpenType math, so it is already possible to use fonts such as Cambria Math in XƎTEX [32], and this OpenType math support will soon become available to the TEX community at large with the upcoming release of TEX Live 2008.

Most likely, LuaTEX will also be adding support for OpenType math eventually, so OpenType math is likely to become a *de facto* standard in the TEX world as well, much as we have adopted other outside developments in the past.

OpenType math for new TEX fonts?

Given these developments, the question posed in the title of this paper about the need for new math font encodings may soon become a non-issue.

If we decide to adopt Unicode and OpenType math font technology in new TEX engines and new fonts, the real question is no longer how to design the layout of encoding tables but rather how to deal with the technology of OpenType math fonts, as we will discuss in the following sections.

Future developments in math fonts

Some background on OpenType math

The OpenType font format was developed jointly by Microsoft and Adobe, based on concepts adopted from the earlier TrueType and PostScript formats. The overall structure of OpenType fonts shares the extensible table structure of TrueType fonts, adding support for different flavors of glyph descriptions in either PostScript CFF or TrueType format. (An extensive documentation of the OpenType format and its features as well as many other important font formats can be found in [33].)

One of the most interesting points about OpenType is the support for 'advanced' typographic features, supporting a considerable amount of intelligence in the font, enabling complex manipulations of glyph positioning or glyph substitutions. At the user level, many of these 'advanced' typographic features can be controlled selectively by the activation of so-called OpenType feature tags.

Despite its name, the OpenType font format is not really open and remains a vendor-controlled specification, much like the previous TrueType and PostScript font formats developed by these vendors. The official OpenType specification is published on a Microsoft web site at [29], but that version may not necessarily reflect the latest developments.

In the case of OpenType math, Microsoft has used its powers as one of the vendors controlling the specification to implement an extension of the OpenType format and declare it as 'experimental' until they see fit to release it. Fortunately, Microsoft was smart enough to borrow from the best examples of math typesetting technology when they designed OpenType math, so they chose TEX as a model for many of the concepts of OpenType math.

The details of OpenType math

The OpenType* MATH *table. One of the most distinctive features of an OpenType math font is the presence of a MATH table. This table contains a number of global font metric parameters, much like the `\fontdimen` parameters of math fonts in TEX described in Appendix G of *The TEXbook*.

In a traditional TEX setup these parameters are essential for typesetting math, controlling various aspects such as the spacing of elements such as big operators, fractions, and indices [34, 35].

In an OpenType font the parameters of the MATH table have a similar role for typesetting math. From what is known, Microsoft apparently consulted with Don Knuth about the design of this table, so the result is not only similar to TEX, but even goes beyond TEX by adding new parameters for cases where hard-wired defaults are applied in TEX.

In the XƎTEX implementation the parameters of the OpenType MATH table are mapped internally to TEX's `\fontdimen` parameters. In most cases this mapping is quite obvious and straight-forward, but unfortunately there are also a few exceptions where some parameters in TEX do not have a direct correspondence in OpenType. It is not clear, however, whether these omissions are just an oversight or a deliberate design decision in case a parameter was deemed irrelevant or unnecessary.

Support for OpenType math in XƎTEX still remains somewhat limited for precisely this reason; until the mapping problems are resolved, XƎTEX has to rely on workarounds to extract the necessary parameters from the OpenType MATH table.

At the time of writing, the extra parameters introduced by OpenType generalizing the concepts of TEX have been silently ignored. It is conceivable, however, that future extensions of new TEX engines might eventually start to use these parameters in the math typesetting algorithms as well.

In the end, whatever technology is used to typeset OpenType math, it remains the responsibility of the font designer to set up the values of all the many parameters affecting the quality of math typesetting. Unfortunately, for a non-technical designer such a task feels like a burden, which is better left to a technical person as a font implementor.

For best results, it is essential to develop a good understanding of the significance of the parameters and how they affect the quality of math typesetting. In [35] we have presented a method for setting up the values of metric parameters of math fonts in TEX. For OpenType math fonts, we would obviously have to reconsider this procedure.

Font metrics of math fonts. Besides storing the global font metric parameters, the OpenType MATH table is also used to store additional glyph-specific information such as italic corrections or kern pairs, as well as information related to the placement of math accents, superscripts and subscripts.

In a traditional TEX setup the font metrics of math fonts have rather peculiar properties, because much of the glyph-specific information is encoded or hidden by overloading existing fields in the TFM metrics in an unusual or non-intuitive way [36].

For example, the width in the TFM metrics is not the real width of the glyph. Instead, it is used to indicate the position where to attach the subscript. Similarly, the italic correction is used to indicate the offset between subscript and superscript.

As another example, fake kern pairs involving a skewchar are used to indicate how much the visual center of the glyphs is skewed in order to determine the position where to attach a math accent.

In OpenType math fonts all such peculiarities will become obsolete, as the MATH table provides data structures to store all the glyph-specific metric information in a much better way. In the case of indices, OpenType math has extended the concepts of TEX by defining 'cut-ins' at the corners on both sides of a glyph and not just to the right.

Unfortunately, while the conceptual clarity of OpenType math may be very welcome in principle, it may cause an additional burden on font designers developing OpenType math fonts based on traditional TEX fonts (such as the Latin Modern fonts) and trying to maintain metric compatibility.

In such cases it may be necessary to examine the metrics of each glyph and to translate the original metrics into appropriate OpenType metrics.

Font encoding and organization. The encoding of OpenType fonts is essentially defined by Unicode code points. Most likely, a typical OpenType math font will include only a subset of Unicode limited to the relevant ranges of math symbols and alphabets, while the corresponding text font may contain a bigger range of scripts.

In a traditional TEX setup the math setup consists of a series of 8-bit fonts organized into families. Typically, each font will contain one set of alphabets in a particular style and a selection of symbols filling the remaining slots.

In a Unicode setup the math setup will consist of only one big OpenType font, containing all the math symbols and operators in the relevant Unicode slots, as well as all the many styles of math alphabets assigned to slots starting at U+1D400.

As a result, there will be several important conceptual implications to consider in the design and implementation of OpenType math fonts, such as how to handle font switches of math alphabets, how to include the various sizes of big operators, delimiters, or radicals, or how to include the optical sizes of superscripts and subscripts.

Handling of math alphabets. In a traditional TEX setup the letters of the Latin and Greek alphabets are subject to font switches between the various math families, usually containing a different style in each family (roman, italic, script, etc.).

In a Unicode setup each style of math alphabets has a different range of slots assigned to it, since each style is assumed to convey a different meaning.

When dealing with direct Unicode input, this might not be a problem, but when dealing with traditional TEX input, quite a lot of setup may be needed at the macro level to ensure that input such as `\mathrm{a}` or `\mathit{a}` or `\mathbf{a}` will be translated to the appropriate Unicode slots.

An additional complication arises because the Unicode code points assigned to the math alphabets are non-contiguous for historical reasons [37]. While most of the alphabetic letters are taken from one big block starting at U+1D400, a few letters which were part of Unicode already before the introduction of Unicode math have to be taken from another block starting at U+2100.

An example implementation of a LaTEX macro package for XƎTEX to support OpenType math is already available [32], and it shows how much setup is needed just to handle math alphabets. Fortunately, such a setup will be needed only once and will be applicable for all Unicode math fonts, quite unlike the case of traditional TEX fonts where each set of math fonts requires its own macro package.

Handling of size variants. Ever since the days of DVI files and PK fonts, TEX users have been accustomed to thinking of font encodings in terms of numeric slots in an encoding table, usually assuming a 1:1 mapping between code points and glyphs.

However, there have always been exceptions to this rule, most notably in the case of a math extension font, where special TFM features were used to set up a linked list from one code point to a series of next-larger glyph variants representing different sizes of operators, delimiters, radicals, or accents, optionally followed by an extensible version.

In a traditional TEX font each glyph variant has a slot by itself in the font encoding, even if it was addressed only indirectly.

In an OpenType font, however, the font encoding is determined by Unicode code points, so the additional glyph variants representing different sizes cannot be addressed directly by Unicode code points and have to remain unencoded, potentially mapped to the Unicode private use area, if needed.

While the conceptual ideas of vertical and horizontal variants and constructions in the OpenType MATH table are very similar to the concepts of charlists and extensible recipes in TEX font metrics, it is interesting to note that OpenType has generalized these concepts a little bit.

While TEX supports extensible recipes only in a vertical context of big delimiters, OpenType also supports horizontal extensible constructions, so it would be possible to define an extensible overbrace or underbrace in the font, rather than at the macro level using straight line segments for the extensible parts. In addition, the same concept could also be applied to arbitrarily long arrows.

Optical sizes for scripts. In a traditional TEX setup math fonts are organized into families, each of them consisting of three fonts loaded at different design sizes representing text style and first and second level script style.

If a math font provides optical design sizes, such as in the case of traditional MetaFont fonts, these fonts are typically loaded at sizes of 10 pt, 7 pt, 5 pt, each of them having different proportions adjusted for improved readability at smaller sizes.

If a math font doesn't provide optical sizes, such as in the case of typical PostScript fonts, scaled-down versions of the 10 pt design size will have to make do, but in such cases it may be necessary to use bigger sizes of first and second level scripts, such as 10 pt, 7.6 pt, 6 pt, since the font may otherwise become too unreadable at such small sizes.

In OpenType math the concept of optical sizes from TEX and MetaFont has been adopted as well, but it is implemented in a different way, typical for OpenType fonts. Instead of loading multiple fonts at different sizes, OpenType math fonts incorporate the multiple design variants in the same font and activate them by a standard OpenType substitution mechanism using a feature tag `ssty=0` and `ssty=1`, not much different from the standard substitutions for small caps or oldstyle figures in text fonts.

It is important to note that the optical design variants intended for use in first and second level scripts, using proportions adjusted for smaller sizes, are nevertheless provided at the basic design size and subsequently scaled down using a scaling factor defined in the OpenType MATH table.

If an OpenType math font lacks optical design variants for script sizes and does not support the `ssty` feature tag, a scaled-down version of the basic design size will be used automatically. The same will also apply to non-alphabetic symbols.

Use of OpenType feature tags. Besides using OpenType feature tags for specific purposes in math fonts, most professional OpenType text fonts also use feature tags for other purposes, such as for selecting small caps or switching between oldstyle and lining figures. Some OpenType fonts may provide a rich set of features, such as a number of stylistic variants, initial and final forms, or optical sizes.

Ultimately, it remains to be seen how the use of OpenType feature tags will influence the organization of OpenType fonts for TEX, such as Latin Modern or TEX Gyre, not just concerning new math fonts, but also existing text fonts.

So far, the Latin Modern fonts have very closely followed the model of the Computer Modern fonts, providing separate fonts for each design size and each font shape or variant.

While it might well be possible to eliminate some variants by making extensive use of OpenType feature tags, such as by embedding small caps into the roman fonts, implementing such a step would imply an important conceptual change and might cause unforeseen problems.

Incorporating multiple design sizes into a single font might have similar implications, but the effects might be less critical if they are limited to the well-controlled environment of math typesetting.

In the TEX Gyre fonts the situation is somewhat simpler, because these fonts are currently limited to the basic roman and italic fonts and do not have small caps variants or optical sizes.

Incorporating a potential addition of small caps in TEX Gyre fonts by means of OpenType feature tags might well be possible without causing any incompatible changes. Similarly, incorporating some expanded design variants with adjusted proportions for use in script sizes would also be conceivable when designing TEX Gyre math fonts.

The impact of OpenType math

As we have seen in the previous sections, OpenType math fonts provide a way of embedding all the relevant font-specific and glyph-specific information needed for high-quality math typesetting.

In many aspects, the concepts of OpenType math are very similar to TEX or go beyond TEX. However, the implementation of these concepts in OpenType fonts will be different in most cases.

Given the adoption of OpenType math as a *de facto* standard and its likelihood of becoming an official standard eventually, OpenType math seems to be the best choice for future developments of new math fonts for use with new TEX engines.

While XeTEX has already started to support OpenType math and LuaTEX is very likely to follow, adopting OpenType for the design of math fonts for Latin Modern or TEX Gyre will take more time and will require developing a deeper understanding of the concepts and data structures.

Most importantly, however, it will also require rethinking many traditional assumptions about the way fonts are organized.

Thus, while the topic of font encodings of math fonts may ultimately become a non-issue, the topic of font technology will certainly remain important.

The challenges of OpenType math

Developing a math font has never been an easy job, so attempting to develop a full-featured OpenType math font for Latin Modern or TEX Gyre certainly presents a major challenge to font designers or font implementors for a number reasons.

First, such a math font will be really large, even in comparison with text fonts, which already cover a large range of Unicode. (In the example of the Cambria Math font, the math font is reported to have more than 2900 glyphs compared to nearly 1000 glyphs in the Cambria text font.) It will have to extend across multiple 16-bit planes to account for the slots of the math alphabets starting at `U+1D400`, and it will also require a considerable number of unencoded glyphs to account for the size variants of extensible glyphs and the optical variants of math alphabets.

Besides the size of the font, such a project will also present many technical challenges in dealing with the technology of OpenType math fonts.

While setting up the font-specific parameters of the OpenType MATH table is comparable to setting up the `\fontdimen` parameters of TEX's math fonts, setting up the glyph-specific information will require detailed attention to each glyph as well as extensive testing and fine-tuning to achieve optimal placement of math accents and indices.

Finally, there will be the question of assembling the many diverse elements that have to be integrated in a comprehensive OpenType math font. So far, the various styles of math alphabets and the various optical sizes of these alphabets have been designed as individual fonts, but in OpenType all of them have to be combined in a single font. Moreover, the optical sizes will have to be set up as substitutions triggered by OpenType feature tags.

Summary and conclusions

In this paper we have reviewed the work on math font encodings since 1990 and the current situation of math fonts as of 2008, especially in view of recent developments in Unicode and OpenType font technology. In particular, we have looked in detail at the features of OpenType math in comparison to the well-known features of TEX's math fonts.

While OpenType math font technology looks very promising and seems to be the best choice for future developments of math fonts, it also presents many challenges that will have to be met.

While support for OpenType math in new TEX engines has already started to appear, the development of math fonts for Latin Modern or TEX Gyre using this font technology will not be easy and will take considerable time.

In the past, the TEX conference in Cork in 1990 was the starting point for major developments in text fonts, which have ultimately led to the adoption of Unicode and OpenType font technology.

Hopefully, the TEX conference at Cork in 2008 might become the starting point for major developments of math fonts in a similar way, except that this time there will be no more need for a new encoding that could be named after the site of the conference.

Acknowledgements

The author wishes to acknowledge feedback, suggestions, and corrections from some of the developers of projects discussed in this review.

A preprint of this paper has been circulated on the Unicode math mailing list hosted at Google Groups [38] and future discussions on the topics of this paper are invited to be directed here.

References

[1] Yannis Haralambous: TEX and Latin alphabet languages. *TUGboat*, 10(3):342–345, 1989. `http://www.tug.org/TUGboat/Articles/tb10-3/tb25hara-latin.pdf`

[2] Nelson Beebe: Character set encoding. *TUGboat*, 11(2):171–175, 1990. `http://www.tug.org/TUGboat/Articles/tb11-2/tb28beebe.pdf`

[3] Janusz S. Bień: On standards for CM font extensions. *TUGboat*, 11(2):175–183, 1990. `http://www.tug.org/TUGboat/Articles/tb11-2/tb28bien.pdf`

[4] Michael Ferguson: Report on multilingual activities. *TUGboat*, 11(4):514–516, 1990. `http://www.tug.org/TUGboat/Articles/tb11-4/tb30ferguson.pdf`

[5] Frank Mittelbach, Robin Fairbairns, Werner Lemberg: LaTEX font encodings, 2006. `http://www.ctan.org/tex-archive/macros/latex/doc/encguide.pdf`

[6] Jörg Knappen: The release 1.2 of the Cork encoded DC fonts and text companion fonts. *TUGboat*, 16(4):381–387, 1995. Reprint from the Proceedings of the 9th European TEX Conference 1995, Arnhem, The Netherlands. `http://www.tug.org/TUGboat/Articles/tb16-4/tb49knap.pdf`

[7] Berthold K. P. Horn: The European Modern fonts. *TUGboat*, 19(1):62–63, 1998
http://www.tug.org/TUGboat/Articles/tb19-1/tb58horn.pdf

[8] Bogusław Jackowski, Janusz M. Nowacki: Latin Modern: Enhancing Computer Modern with accents, accents, accents. *TUGboat*, 24(1):64–74, 2003. Proceedings of the TUG 2003 Conference, Hawaii, USA.
http://www.tug.org/TUGboat/Articles/tb24-1/jackowski.pdf

[9] Bogusław Jackowski, Janusz M. Nowacki: Latin Modern: How less means more. *TUGboat*, 27(0):171–178, 2006 Proceedings of the 15th European TEX Conference 2005, Pont-à-Mousson, France.
http://www.tug.org/TUGboat/Articles/tb27-0/jackowski.pdf

[10] Will Robertson: An exploration of the Latin Modern fonts. *TUGboat*, 28(2):177-180, 2007.
http://www.tug.org/TUGboat/Articles/tb28-2/tb89robertson.pdf

[11] Hans Hagen, Jerzy B. Ludwichowski, Volker RW Schaa: The new font project: TEX Gyre. *TUGboat*, 27(2):250–253, 2006. Proceedings of the TUG 2006 Conference, Marrakesh, Morocco.
http://www.tug.org/TUGboat/Articles/tb27-2/tb87hagen-gyre.pdf

[12] Jerzy B. Ludwichowski, Bogusław Jackowski, Janusz M. Nowacki: Five years after: Report on international TEX font projects. *TUGboat*, 29(1):25–26, 2008. Proceedings of the 17th European TEX Conference 2007, Bachotek, Poland.
https://www.tug.org/TUGboat/Articles/tb29-1/tb91ludwichowski-fonts.pdf

[13] Alan Jeffrey: Math font encodings: A workshop summary. *TUGboat*, 14(3):293–295, 1993. Proceedings of the TUG 1993 Conference, Aston University, Birmingham, UK.
http://www.tug.org/TUGboat/Articles/tb14-3/tb40mathenc.pdf

[14] Justin Ziegler: Technical report on math font encodings. LaTEX3 Project Report, 1993.
http://www.ctan.org/tex-archive/info/ltx3pub/processed/l3d007.pdf

[15] Math Font Group (MFG) web site, archives, papers, and mailing list.
http://www.tug.org/twg/mfg/
http://www.tug.org/twg/mfg/archive/
http://www.tug.org/twg/mfg/papers/
http://www.tug.org/mailman/listinfo/math-font-discuss

[16] Matthias Clasen, Ulrik Vieth: Towards a new Math Font Encoding for AllTEX. *Cahiers GUTenberg*, 28–29:94–121, 1998. Proceedings of the 10th European TEX Conference 1998, St. Malo, France.
http://www.gutenberg.eu.org/pub/GUTenberg/publicationsPDF/28-29-clasen.pdf

[17] Ulrik Vieth *et al.*: Summary of math font-related activities at EuroTEX 1998. *MAPS*, 20:243–246, 1998.
http://www.ntg.nl/maps/20/36.pdf

[18] Ulrik Vieth: What is the status of new math font encodings? Posting to mailing list, 2007.
http:/www./tug.org/pipermail/math-font-discuss/2007-May/000068.html

[19] Barbara Beeton, Asmus Freytag, Murray Sargent III: Unicode Support for Mathematics. Unicode Technical Report UTR#25. 2001.
http://www.unicode.org/reports/tr25/

[20] Barbara Beeton: Unicode and math, a combination whose time has come–Finally! *TUGboat*, 21(3):174–185, 2000. Proceedings of the TUG 2000 Conference, Oxford, UK.
http://www.tug.org/TUGboat/Articles/tb21-3/tb68beet.pdf

[21] Barbara Beeton: The STIX Project–From Unicode to fonts. *TUGboat*, 28(3):299–304, 2007. Proceedings of the TUG 2007 Conference, San Diego, California, USA.
http://www.tug.org/TUGboat/Articles/tb28-3/tb90beet.pdf

[22] STIX Fonts Project: Web Site and Frequently Asked Questions.
http://www.stixfonts.org/
http://www.stixfonts.org/STIXfaq.html

[23] Murray Sargent III: Math in Office Blog.
http://blogs.msdn.com/murrays/default.aspx

[24] Murray Sargent III: High-quality editing and display of mathematical text in Office 2007.
http://blogs.msdn.com/murrays/archive/2006/09/13/752206.aspx

[25] Tiro Typeworks: Cambria Math Specimen.
http://www.tiro.nu/Articles/Cambria/Cambria_Math_Basic_Spec_V1.pdf

[26] John Hudson, Ross Mills: Mathematical Typesetting: Mathematical and scientific typesetting solutions from Microsoft. Promotional Booklet, Microsoft, 2006.
http://www.tiro.com/projects/

[27] Daniel Rhatigan: Three typefaces for mathematics. The development of Times 4-line Mathematics, AMS Euler, and Cambria Math. Dissertation for the MA in typeface design,

University of Reading, 2007.
http://www.typeculture.com/academic_resource/articles_essays/pdfs/tc_article_47.pdf

[28] Murray Sargent III: Unicode Nearly Plain Text Encodings of Mathematics. Unicode Technical Note UTN#28, 2006.
http://www.unicode.org/notes/tn28/

[29] Microsoft Typography: OpenType specification version 1.5.
http://www.microsoft.com/typography/otspec/

[30] George Williams: FontForge. Math typesetting information.
http://fontforge.sourceforge.net/math.html

[31] Apostolos Syropoulos: Asana Math.
www.ctan.org/tex-archive/fonts/Asana-Math/

[32] Will Robertson: Experimental Unicode math typesetting: The unicode-math package.
http://github.com/wspr/unicode-math/tree/master

[33] Yannis Haralambous: Fonts and Encodings. O'Reilly Media, 2007. ISBN 0-596-10242-9
http://oreilly.com/catalog/9780596102425/

[34] Bogusław Jackowski: Appendix G Illuminated. *TUGboat*, 27(1):83–90, 2006. Proceedings of the 16th European TEX Conference 2006, Debrecen, Hungary.
http://www.tug.org/TUGboat/Articles/tb27-1/tb86jackowski.pdf

[35] Ulrik Vieth: Understanding the æsthetics of math typesetting. *Biuletyn* GUST, 5–12, 2008. Proceedings of the 16th BachoTEX Conference 2008, Bachotek, Poland.
http://www.gust.org.pl/projects/e-foundry/math-support/vieth2008.pdf

[36] Ulrik Vieth: Math Typesetting in TEX: The Good, the Bad, the Ugly. *MAPS*, 26:207–216, 2001. Proceedings of the 12th European TEX Conference 2001, Kerkrade, Netherlands.
http://www.ntg.nl/maps/26/27.pdf

[37] Unicode Consortium: Code Charts for Symbols and Punctuation.
http://www.unicode.org/charts/symbols.html

[38] Google Groups: Unicode math for TEX.
http://groups.google.com/group/unimath

Ulrik Vieth
Vaihinger Straße 69
70567 Stuttgart
Germany
ulrik dot vieth (at) arcor dot de

OpenType Math Illuminated

Abstract
In recent years, we have seen the development of new TEX engines, XƎTEX and LuaTEX, adopting OpenType font technology for providing Unicode typesetting support. While there are already plenty of OpenType text fonts available for use, both from the TEX community and from commercial font suppliers, there is little support for OpenType math fonts so far. Ironically, it was left to Microsoft to develop a *de facto* standard for OpenType math font information and to provide the first reference implementation of a full-featured OpenType math font.

In order to develop the much-needed math support for Latin Modern and TEX Gyre fonts, it will be crucially important to develop a good understanding of the internals of OpenType math tables, much as it is necessary to develop a good understanding of Appendix G and TEX's `\fontdimen` parameters to develop math support for traditional TEX fonts. In this paper, we try to help improve the understanding of OpenType math internals, summarizing the parameters of OpenType math fonts as well as illustrating the similarities and differences between traditional TEX math fonts and OpenType math fonts.

Background on OpenType math

In recent years, the TEX community has been going through a phase of very significant developments. Among the most important achievements, we have seen the development of new TEX engines, XƎTEX and LuaTEX, providing support for Unicode and OpenType font technology. At about the same time we have also seen the development of new font distributions, Latin Modern and TEX Gyre, provided simultaneously in Type 1 format as a set of 8-bit font encodings as well as in OpenType format.

Together these developments have enabled TEX users to keep up with current trends in the publishing industry, providing users of the new TEX engines with a comprehensive set of free OpenType fonts and enabling them to take advantage of the many offerings by commercial font suppliers.

As far as text typesetting is concerned, support for OpenType font technology in the new TEX engines is already very advanced, supporting not only traditional typographic features of Latin alphabets, but also addressing the very complex and challenging requirements of Arabic typography.

However, when it comes to math typesetting, one of the traditional strongholds of TEX, support for Unicode and OpenType math is only just beginning to take shape.

Ironically, it was left to Microsoft to develop the first system to offer support for Unicode math. When Microsoft introduced support for math typesetting in Office 2007 [1, 2], they extended the OpenType font format and commissioned the design of Cambria Math [3] as a reference implementation of a full-featured OpenType math font.

Fortunately for us, Microsoft was smart enough to borrow from the best examples of math typesetting technology, thus many concepts of OpenType math are not only derived from the model of TEX, but also go beyond TEX and introduce extensions or generalizations of familiar concepts.

While OpenType math is officially still considered experimental, it is quickly becoming a *de facto* standard, as it has already been widely deployed to millions of installations of Microsoft Office 2007 and it is also being been adopted by other projects such as the FontForge [4] font editor or independent font designs such as Asana Math [5].

Most importantly, support for OpenType math has already been implemented or is currently being implemented in the new TEX engines, thus adopting OpenType math for the development of the much-needed Unicode math support for Latin Modern and TEX Gyre obviously seems to be most promising choice of technology.

Design and quality of math fonts

When it comes to developing math fonts, designing the glyph shapes is only part of the job. Another part, which is equally important, is to adjust the glyph metrics of individual glyphs and to set up the global parameters affecting various aspects of glyph positioning in math typesetting.

As we have discussed at previous conferences, the quality of math typesetting crucially depends on the fine-tuning of these parameters. Developing a good understanding of these parameters will therefore become an important prerequisite to support the development of new math fonts.

In the case of traditional TeX math fonts, we have to deal with the many `\fontdimen` parameters which have been analyzed in Bogusław Jackowski's paper *Appendix G Illuminated* and a follow-up paper by the present author [6, 7].

In the case of OpenType math fonts, we need to develop a similar understanding of the various tables and parameters and how the concepts of OpenType math relate to the concepts of TeX.

Overview of the OpenType font format

The OpenType font format [8] was developed jointly by Adobe and Microsoft, based on elements of the earlier PostScript and TrueType font formats by the same vendors. The overall structure of OpenType fonts consists of a number of tables, some of which are required while others are optional [9].

In the case of OpenType math, the extension of the font format essentially consists of adding another optional table, the so-called MATH table, containing all the information related to math typesetting. Since it is an optional table, it would be interpreted only by software which knows about it (such as the new TeX engines or Microsoft Office 2007), while it would be ignored by other software.

Unlike a database table, which has a very rigid format, an OpenType font table can have a fairly complex structure, combining a variety of different kinds of information in the same table. In the case of the OpenType MATH table, we have the following kinds of information:

- □ a number of global parameters specific to math typesetting (similar to TeX's many `\fontdimen` parameters of Appendix G)
- □ instructions for vertical and horizontal variants and/or constructions (similar to TeX's charlists and extensible recipes)
- □ additional glyph metric information specific to math mode (such as italic corrections, accent placement, or kerning)

In the following sections, we will discuss some of these parameters in more detail, illustrating the similarities and differences between traditional TeX math fonts and OpenType math fonts.

Parameters of OpenType math fonts

The parameters of the OpenType MATH table play a similar role as TeX's `\fontdimen` parameters, controlling various aspects of math typesetting, such as the placement of limits on big operators, the placement of numerators and denominators in fractions, or the placement of superscripts and subscripts.

While a number of parameters are specified in TeX through the `\fontdimen` parameters of math fonts, there are other parameters which are defined by built-in rules of TeX's math typesetting engine. In many such cases, additional parameters have been introduced in the OpenType MATH table, making it possible to specify all the relevant parameters in the math font without relying on built-in rules of any particular typesetting engine.

In view of the conference motto, it is interesting to note that the two new TeX engines, XeTeX and LuaTeX, have taken a very different approach how to support the additional parameters of OpenType math fonts: While XeTeX has retained TeX's original math typesetting engine and uses an internal mapping to set up `\fontdimen` parameters from OpenType parameters [10], LuaTeX has introduced an extension of TeX's math typesetting engine [11], which will allow to take full advantage of most of the additional OpenType parameters. (More precisely, while XeTeX only provides access to the OpenType parameters as additional `\fontdimens`, LuaTeX uses an internal data structure based on the combined set of OpenType and TeX parameters, making it possible to supply missing values which are not supported in either OpenType math fonts or traditional TeX math fonts.)

For font designers developing OpenType math fonts, it may be best to supply all of the additional OpenType parameters in order to make their fonts as widely usable as possible with any typesetting engine, not necessarily limited to any specific one of the new TeX engines.

In the following sections, we will take a closer look at the various groups of OpenType parameters, organized in a similar way as they are presented to font designers in the FontForge font editor, but not necessarily in the same order.

We will use the figures from [6, 7] as a visual clue to illustrate how the various parameters are defined in TeX, while summarizing the similarities and differences between OpenType parameters and TeX parameters in tabular form.

Limits on big operators

In TeX math fonts, there are five parameters controlling the placement of limits on big operators (see figure 1), which are denoted as ξ_9 to ξ_{13} using the notation of Appendix G.

Two of them control the default position of the limits (ξ_{10} and ξ_{12}), two of them control the inside gap (ξ_9 and ξ_{11}), while the final one controls the outside gap above and below the limits (ξ_{13}).

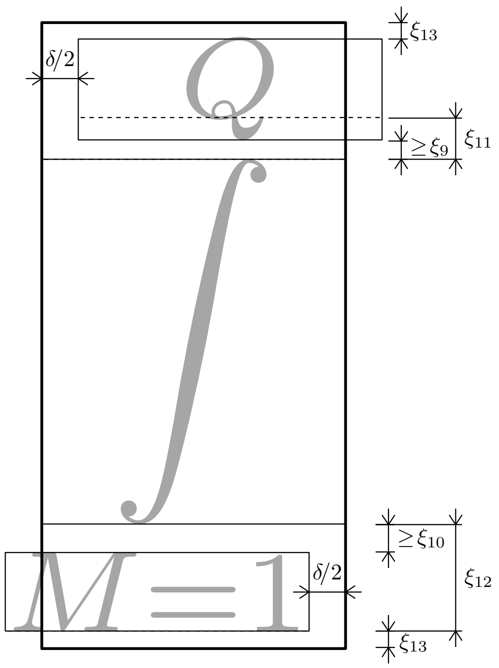


Figure 1. TEX font metric parameters affecting the placement of limits above or below big operators.

In OpenType math fonts, the MATH table contains only four parameters controlling the placement of limits on big operators. Those four parameters have a direct correspondence to TEX's parameters (as shown in table 1), while the remaining one has no correspondence and is effectively set to zero. (Considering the approach taken in other circumstances, it is very likely that if there were any such correspondence, there might actually be two parameters in OpenType instead of only one, such as `UpperLimitExtraAscender` and `LowerLimitExtraDescender`. In LuaTEX's internal data structures, there are actually two parameters for this purpose, which are either initialized from TEX's parameter ξ_{13} when using TEX math fonts or set to zero when using OpenType math fonts.)

OpenType parameter	TEX parameter
`UpperLimitBaselineRiseMin`	ξ_{11}
`UpperLimitGapMin`	ξ_9
`LowerLimitGapMin`	ξ_{10}
`LowerLimitBaselineDropMin`	ξ_{12}
(no correspondence)	ξ_{13}

Table 1. Correspondence of font metric parameters between OpenType and TEX affecting the placement of limits above or below big operators.

OpenType parameter	TEX parameter
`StretchStackTopShiftUp`	ξ_{11}
`StretchStackGapAboveMin`	ξ_9
`StretchStackGapBelowMin`	ξ_{10}
`StretchStackBottomShiftDown`	ξ_{12}

Table 2. Correspondence of font metric parameters between OpenType and TEX related to stretch stacks.

Stretch Stacks

Stretch stacks are a new feature in OpenType math fonts, which do not have a direct correspondence in TEX. They can be understood in terms of material stacked above or below stretchable elements such as overbraces, underbraces or long arrows.

In TEX, such elements were typically handled at the macro level and effectively treated in the same way as limits on big operators.

In LuaTEX, such elements will be implemented by new primitives using either the new OpenType parameters for stretch stacks (as shown in table 2) or the parameters for limits on big operators when using traditional TEX math fonts.

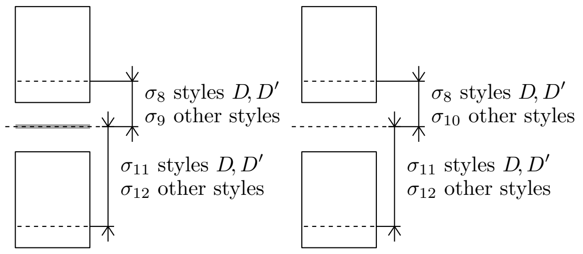
Overbars and Underbars

In TEX math fonts, there are no specific parameters related to the placement of overlines and underlines. Instead, there is only one parameter controlling the default rule thickness (ξ_8), which is used in a number of different situations where other parameters are expressed in multiples of the rule thickness.

In OpenType math fonts, a different approach was taken, introducing extra parameters for each purpose, even supporting different sets of parameters for overlines and underlines. Thus the MATH table contains the following parameters related to overlines and underlines (as shown in table 3), which only have an indirect correspondence in TEX.

OpenType parameter	TEX parameter
`OverbarExtraAscender`	$(= \xi_8)$
`OverbarRuleThickness`	$(= \xi_8)$
`OverbarVerticalGap`	$(= 3\,\xi_8)$
`UnderbarVerticalGap`	$(= 3\,\xi_8)$
`UnderbarRuleThickness`	$(= \xi_8)$
`UnderbarExtraDescender`	$(= \xi_8)$

Table 3. Correspondence of font metric parameters between OpenType and TEX affecting the placement of overlines and underlines.



σ_8 styles D, D'
σ_9 other styles
σ_{11} styles D, D'
σ_{12} other styles
σ_8 styles D, D'
σ_{10} other styles
σ_{11} styles D, D'
σ_{12} other styles

Figure 2. TEX font metric parameters affecting the placement of numerators and denominators in regular and generalized fractions.

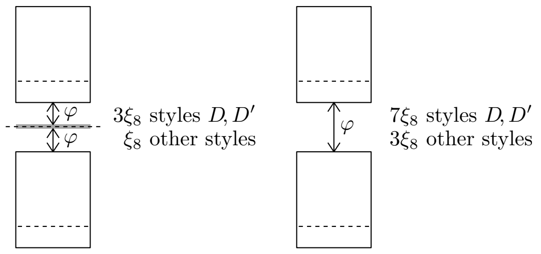


Figure 3. TEX's boundary conditions affecting the placement of numerators and denominators in regular and generalized fractions.

It is interesting to note that the introduction of additional parameters in OpenType math fonts provides for greater flexibility of the font designer to adjust the values for best results.

While TEX's built-in rules always use a fixed multiplier of the rule thickness regardless of its size, OpenType math fonts can compensate for a larger rule thickness by using a smaller multiplier.

An example can be found when inspecting the parameter values of Cambria Math: In relative terms the inside gap is only about 2.5 times rather than 3 times the rule thickness, while the latter (at about 0.65 pt compared to 0.4 pt) is quite a bit larger than in typical TEX fonts.

Obviously, making use of the individual OpenType parameters (as in LuaTEX) instead of relying on TEX's built-in rules (as in XƎTEX) would more closely reflect the intention of the font designer.

Fractions and Stacks

In TEX math fonts, there are five parameters controlling the placement of numerators and denominators (see figure 2), which are denoted as σ_8 to σ_{12} using the notation of Appendix G.

Four of them apply to regular fractions, either in display style (σ_8 and σ_{11}) or in text style and below (σ_9 and σ_{12}), while the remaining one applies to the special case of generalized fractions when the fraction bar is absent (σ_{10}).

Besides those specific parameters, there are also a number of parameters which are based on built-in rules of TEX's math typesetting engine, expressed in multiples of the rule thickness (ξ_8), such as the thickness of the fraction rule or the inside gap above and below the fraction rule (see figure 3).

In OpenType math fonts, a different approach was once again taken, introducing a considerable number of additional parameters for each purpose. Thus the MATH table contains 9 parameters related to regular fractions and 6 more parameters related to generalized fractions (also known as stacks).

As shown in table 4, there is a correspondence for all TEX parameters, but this correspondence isn't necessarily unique when the same TEX parameter is used for multiple purposes in fractions and stacks. Obviously, font designers of OpenType math fonts should be careful about choosing the values of OpenType parameters in a consistent way.

Analyzing the font parameters of Cambria Math once again shows how the introduction of additional parameters increases the flexibility of the designer to adjust the parameters for best results: In relative terms, `FractionDisplayStyleGapMin` is only about 2 times rather than 3 times the rule thickness. Similarly, `StackDisplayStyleGapMin` is only about 4.5 times rather than 7 times the rule thickness. In absolute terms, however, both parameters are about the same order of magnitude as in typical TEX fonts.

OpenType parameter	TEX parameter
`FractionNumeratorDisplayStyleShiftUp`	σ_8
`FractionNumeratorShiftUp`	σ_9
`FractionNumeratorDisplayStyleGapMin`	$(= 3\,\xi_8)$
`FractionNumeratorGapMin`	$(= \xi_8)$
`FractionRuleThickness`	$(= \xi_8)$
`FractionDenominatorDisplayStyleGapMin`	$(= 3\,\xi_8)$
`FractionDenominatorGapMin`	$(= \xi_8)$
`FractionDenominatorDisplayStyleShiftDown`	σ_{11}
`FractionDenominatorShiftDown`	σ_{12}
`StackTopDisplayStyleShiftUp`	σ_8
`StackTopShiftUp`	σ_{10}
`StackDisplayStyleGapMin`	$(= 7\,\xi_8)$
`StackGapMin`	$(= 3\,\xi_8)$
`StackBottomDisplayStyleShiftDown`	σ_{11}
`StackBottomShiftDown`	σ_{12}

Table 4. Correspondence of font metric parameters between OpenType and TEX affecting the placement of numerators and denominators.

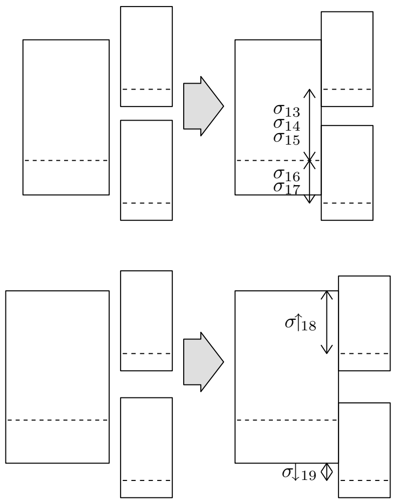


Figure 4. TEX font metric parameters affecting the placement of superscripts and subscripts on a simple character or a boxed subformula.

Superscripts and Subscripts

In TEX math fonts, there are seven parameters controlling the placement of superscripts and subscripts (see figure 4), which are denoted as σ_{13} to σ_{19} using the notation of Appendix G.

Three of them apply to superscripts, either in display style (σ_{13}), in text style and below (σ_{14}), or in cramped style (σ_{15}), while the other two apply to the placement of subscripts, either with or without a superscript (σ_{16} and σ_{17}).

Finally, there are two more parameters which apply to superscripts and subscripts on a boxed subformula (σ_{18} and σ_{19}), which also apply to limits attached to big operators with `\nolimits`.

Besides those specific parameters, there are also a number of parameters which are based on TEX's built-in rules, expressed in multiples of the x-height (σ_5) or the rule thickness (ξ_8), most of them related to resolving collisions between superscripts and subscripts or adjusting the position when a superscript or subscript becomes too big (see figure 5).

In OpenType math fonts, we once again find a number of additional parameters for each specific purpose, as shown in table 5.

It is interesting to note that some of the usual distinctions made in TEX were apparently omitted in the OpenType MATH table, as there is no specific value for the superscript position in display style, nor are there any differences in subscript position in the presence or absence of superscripts.

While it is not clear why there is no correspondence for these parameters, it is quite possible that there was a conscious design decision to omit them, perhaps to avoid inconsistencies in alignment.

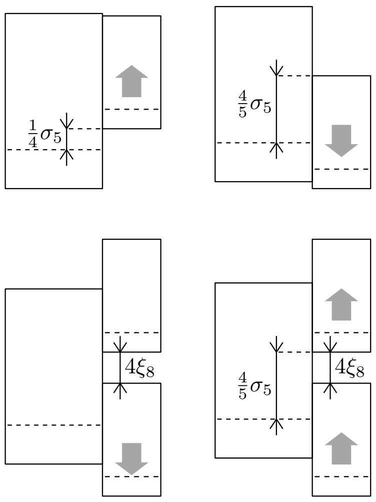


Figure 5. TEX font metric parameters affecting the placement of superscripts and subscripts in cases of resolving collisions.

OpenType parameter	TEX parameter
SuperscriptShiftUp	σ_{13}, σ_{14}
SuperscriptShiftUpCramped	σ_{15}
SubscriptShiftDown	σ_{16}, σ_{17}
SuperscriptBaselineDropMax	σ_{18}
SubscriptBaselineDropMin	σ_{19}
SuperscriptBottomMin	$(=\frac{1}{4}\sigma_5)$
SubscriptTopMax	$(=\frac{4}{5}\sigma_5)$
SubSuperscriptGapMin	$(=4\,\xi_8)$
SuperscriptBottomMaxWithSubscript	$(=\frac{4}{5}\sigma_5)$
SpaceAfterScript	`\scriptspace`

Table 5. Correspondence of font metric parameters between OpenType and TEX affecting the placement of superscripts and subscripts.

Radicals

In TEX math fonts, there are no specific parameters related to typesetting radicals. Instead, the relevant parameters are based on built-in rules of TEX's math typesetting engine, expressed in multiples of the rule thickness (ξ_8) or the x-height (σ_5).

To be precise, there are even more complications involved [6], as the height of the fraction rule is actually taken from the height of the fraction glyph rather than the default rule thickness to account for effects of pixel rounding in bitmap fonts.

OpenType parameter	TEX parameter
RadicalExtraAscender	$(= \xi_8)$
RadicalRuleThickness	$(= \xi_8)$
RadicalDisplayStyleVerticalGap	$(= \xi_8 + \frac{1}{4}\sigma_5)$
RadicalVerticalGap	$(= \xi_8 + \frac{1}{4}\xi_8)$
RadicalKernBeforeDegree	e. g. $\frac{5}{18}$ em
RadicalKernAfterDegree	e. g. $\frac{10}{18}$ em
RadicalDegreeBottomRaisePercent	e. g. 60 %

Table 6. Correspondence of font metric parameters between OpenType and TEX affecting the placement of radicals.

In OpenType math fonts, we once again find a number of additional parameters for each purpose, as shown in table 6.

While there is a correspondence for all of the parameters built into TEX's typesetting algorithms, it is interesting to note that OpenType math has also introduced some additional parameters related to the placement of the degree of an n th root ($\sqrt[n]{x}$), which is usually handled at the macro level in TEX's format files `plain.tex` or `latex.ltx`:

```
\newbox\rootbox
\def\root#1\of{%
  \setbox\rootbox
  \hbox{$\m@th\scriptscriptstyle{#1}$}%
  \mathpalette\r@@t}
\def\r@@t#1#2{%
  \setbox\z@\hbox{$\m@th#1\sqrtsign{#2}$}%
  \dimen@=\ht\z@ \advance\dimen@-\dp\z@
  \mkern5mu\raise.6\dimen@\copy\rootbox
  \mkern-10mu\box\z@}
```

As shown in the listing, the definition of the `\root` macro contains a number of hard-coded parameters, such as a positive kern before the box containing the degree and negative kern thereafter, expressed in multiples of the font-specific math unit. In addition, there is also a raise factor expressed relative to the size of the box containing the radical sign.

Obviously, the extra OpenType parameters related to the degree of radicals correspond directly to the parameters used internally in the `\root` macro, making it possible to supply a set of font-specific values instead of using hard-coded values expressed in multiples of font-specific units.

In LuaTEX, this approach has been taken one step further, introducing a new `\Uroot` primitive as an extension of the `\Uradical` primitive, making it possible to replace the processing at the macro level by processing at the algorithmic level in LuaTEX's extended math typesetting engine [11].

OpenType parameter	TEX parameter
ScriptPercentScaleDown	e. g. 70–80 %
ScriptScriptPercentScaleDown	e. g. 50–60 %
DisplayOperatorMinHeight	?? (e. g. 12–15 pt)
(no correspondence)	σ_{20} (e. g. 20–24 pt)
DelimitedSubFormulaMinHeight	σ_{21} (e. g. 10–12 pt)
AxisHeight	σ_{22} (axis height)
AccentBaseHeight	σ_5 (x-height)
FlattenedAccentBaseHeight	?? (capital height)

Table 7. Correspondence of font metric parameters between OpenType and TEX affecting some general aspects of math typesetting.

General parameters

The final group of OpenType parameters combines a mixed bag of parameters for various purposes. Some of them have a straight-forward correspondence in TEX (such as the math axis position), while others do not have any correspondence at all. As shown in table 7, there are some very noteworthy parameters in this group, which deserve some further explanations in the following paragraphs.

`(Script)ScriptPercentScaleDown`.
These OpenType parameters represent the font sizes of the first and second level script fonts relative to the base font. In TEX math fonts, these parameters do not have a correspondence in the font metrics. Instead they are usually specified at the macro level when a family of math fonts is loaded.

If a font family provides multiple design sizes (as in Computer Modern), font loading of math fonts in TEX might look like the following, using different design sizes, each at their natural size:

```
\newfam\symbols
\textfont\symbols=cmsy10
\scriptfont\symbols=cmsy7
\scriptscriptfont\symbols=cmsy5
```

If a font family does not provide multiple design sizes (as in MathTime), font loading of math fonts will use scaled-down versions of the base font:

```
\newfam\symbols
\textfont\symbols=mtsy10 at 10pt
\scriptfont\symbols=mtsy10 at 7.6pt
\scriptscriptfont\symbols=mtsy10 at 6pt
```

The appropriate scaling factors depend on the font design, but are usually defined in macro packages or in format files using higher-level macros such as `\DeclareMathSizes` in LaTEX.

In OpenType math fonts, it will be possible to package optical design variants for script sizes into a single font by using OpenType feature selectors to address the design variants and using scaling factors as specified in the MATH table. (As discussed in [12], there are many issues to consider regarding the development of OpenType math fonts besides setting up the font parameters. One such issue is the question of font organization regarding the inclusion of optical design variants into the base font.)

The corresponding code for font loading of full-featured OpenType math fonts in new TEX engines might look like the following:

```
\newfam\symbols
\textfont\symbols="CambriaMath"
\scriptfont\symbols="CambriaMath:+ssty0"
  scaled <ScriptPercentScaleDown>
\scriptscriptfont\symbols="CambriaMath:+ssty1"
  scaled <ScriptScriptPercentScaleDown>
```

If the font provides optical design variants for some letters and symbols, they will be substituted using the +ssty0 or +ssty1 feature selectors, but the scaling factor of (Script)ScriptPercentScaleDown will be applied in any case regardless of substitutions.

`DisplayOperatorMinHeight`.
This OpenType parameter represents the minimum size of big operators in display style. While TEX only supports two sizes of operators, which are used in text style and display style, OpenType can support multiple sizes of big operators and it needs an additional parameter to determine the smallest size to use in display style.

For font designers, it should be easy to set this parameter based on the design size of big operators, e. g. using 14 pt for display style operators combined with 10 pt for text style operators.

`DelimitedSubFormulaMinHeight`.
This OpenType parameter represents the minimum size of delimited subformulas and it might also be applied to the special case of delimited fractions.

To illustrate the significance, some explanations may be necessary to point out the difference between the usual case of fractions with delimiters and the special case of delimited fractions.

If a generalized fraction with delimiters is coded like the following

```
$ \left( {n \atop k} \right) $
```

the contents will be treated as a standard case of a generalized fraction, and the size of delimiters will be determined by taking into account the effects of \delimiterfactor and \delimitershortfall as set up in the format file.

As a result, we will typically get 10 pt or 12 pt delimiters in text style and 18 pt or 24 pt delimiters in display style. For typical settings, the delimiters only have to cover 90 % of the required size and they may fall short by at most 5 pt.

If a generalized fraction with delimiters is coded like the following

```
$ {n \atopwithdelims() k} $
```

the contents will be treated as a delimited fraction, and in this case the size of delimiters will depend on the \fontdimen parameters σ_{20} and σ_{21} applicable in either display style or text style.

As a result, regardless of the contents, we will always get 10 pt delimiters in text style and 24 pt delimiters in display style, even if 18 pt delimiters would be big enough in the standard case.

While DelimitedSubFormulaMinHeight may be the best choice of OpenType parameters to supply a value for TEX's \fontdimen parameters related to delimited fractions, it will be insufficient by itself to represent a distinction between display style and text style values needed in TEX. (Unless we simply assume a factor, such as $\sigma_{20} = 2\,\sigma_{21}$.)

In the absence of a better solution, it may be best to simply avoid using \atopwithdelims with OpenType math fonts in the new TEX engines and to redefine user-level macros (such as \choose) in terms of \left and \right delimiters.

`(Flattened)AccentBaseHeight`.
These OpenType parameters affect the placement of math accents and are closely related to design parameters of the font design.

While TEX assumes that accents are designed to fit on top of base glyphs which do not exceed the x-height (σ_5) and adjusts the vertical position of accents accordingly, OpenType provides a separate parameter for this purpose, which doesn't have to match the x-height of the font, but plays a similar role with respect to accent placement.

In addition to that, OpenType has introduced another mechanism to replace accents by flattened accents if the size of the base glyph exceeds a certain size, which is most likely related to the height of capital letters. At the time of writing, support for flattened accents has not yet been implemented in the new TEX engines, but it is being considered for LuaTEX version 0.40 [11].

In view of these developments, font designers are well advised to supply a complete set of values for all the OpenType math parameters since new TEX engines working on implementing full support for OpenType math may start using them sooner rather than later.

So far, we have discussed only one aspect of the information contained in the OpenType MATH table, focusing on the global parameters which correspond to TEX's `\fontdimen` parameters or to built-in rules of TEX's math typesetting algorithms.

Besides those global parameters, there are other data structures in the OpenType MATH table, which are also important to consider, as we will discuss in the following sections.

Instructions for vertical and horizontal variants and constructions

The concepts of vertical and horizontal variants and constructions in OpenType math are obviously very similar to TEX's concepts of charlists and extensible recipes. However, there are some subtle differences regarding when and how these concepts are applied in the math typesetting algorithms.

In TEX, charlists and extensible recipes are only used in certain situations when typesetting elements such as big operators, big delimiters, big radicals or wide accents. In OpenType math fonts, these concepts have been extended and generalized, allowing them to be used also for other stretchable elements such as long arrows or over- and underbraces.

Vertical variants and constructions

Big delimiters. When typesetting big delimiters or radicals TEX uses charlists to switch to the next-larger vertical variants, optionally followed by extensible recipes for vertical constructions. In OpenType math, these concepts apply in the same way.

It is customary to provide at least four fixed-size variants, using a progression of sizes such as 12 pt, 18 pt, 24 pt, 30 pt, before switching to an extensible version, but there is no requirement for that other than compatibility and user expectations. (At the macro level these sizes can be accessed by using `\big` (12 pt), `\Big` (18 pt), `\bigg` (24 pt), `\Bigg` (30 pt).)

Font designers are free to provide any number of additional or intermediate sizes, but in TEX they used to be limited by constraints such as 256 glyphs per 8-bit font table and no more than 16 different heights and depths in TFM files. In OpenType math fonts, they are no longer subject to such restrictions, and in the example of Cambria Math big delimiters are indeed provided in seven sizes.

Big operators. When typesetting big operators TEX uses the charlist mechanism to switch from text style to display style operators, but only once. There is no support for multiple sizes of display operators, nor are there extensible versions.

In OpenType math, these concepts have been extended, so it would be possible to have multiple sizes of display style operators as well as extensible versions of operators, if desired.

While LuaTEX has already implemented most of the new features of OpenType math, it has not yet addressed additional sizes of big operators, and it is not clear how that would be done.

Most likely, this would require some changes to the semantics of math markup at the user level, so that operators would be defined to apply to a scope of a subformula, which could then be measured to determine the required size of operators.

In addition, such a change might also require adding new parameters to decide when an operator is big enough, similar to the role of the parameters `\delimiterfactor` and `\delimitershortfall` in the case of big delimiters.

Horizontal variants and constructions

Wide accents. When typesetting wide math accents TEX uses charlists to switch to the next-larger horizontal variants, but it doesn't support extensible recipes for horizontal constructions.

As a result, math accents in traditional TEX fonts cannot grow beyond a certain maximum size, and stretchable horizontal elements of arbitrary size have to be implemented using other mechanisms, such as alignments at the macro level.

In OpenType math, these concepts have been extended, making it possible to introduce extensible versions of wide math accents (or similar elements), if desired. In addition, new mechanisms for bottom accents have also been added, complementing the existing mechanisms for top accents.

Over- and underbraces. When typesetting some stretchable elements such as over- and underbraces, TEX uses an alignment construction at the macro level to get an extensible brace of the required size, which is then typeset as a math operator with upper or lower limits attached.

While it would be possible to define extensible over- and underbraces in OpenType math fonts as extensible versions of math accents, the semantics of math accents aren't well suited to handle upper or lower limits attached to those elements.

In LuaTEX, new primitives `\Uoverdelimiter` and `\Uunderdelimiter` have been added as a new concept to represent stretchable horizontal elements which may have upper or lower limits attached. The placement of these limits is handled similar to limits on big operators in terms of so-called 'stretch stacks' as discussed earlier in section .

Long arrows. In TeX math fonts, long horizontal arrows are constructed at the macro level by overlapping the glyphs of short arrows and suitable extension modules (such as $-$ or $=$). Similarly, arrows with hooks or tails are constructed by overlapping the glyphs of regular arrows and suitable glyphs for the hooks or tails.

In OpenType math fonts, all such constructions can be defined at the font level in terms of horizontal constructions rather than relying on the macro level. However, in most cases such constructions will also contain an extensible part, making the resulting long arrows strechable as well.

In LuaTeX, stretchable long arrows can also be defined using the new primitives `\Uoverdelimiter` as discussed in the case of over- and underbraces. The placement of limits on such elements more or less corresponds to using macros such as `\stackrel` to stack text on top of a relation symbol.

Encoding of variants and constructions

In traditional TeX math fonts, glyphs are addressed by a slot number in a font-specific output encoding. Each variant glyph in a charlist and each building block in an extensible recipe needs to have a slot of its own in the font table. However, only the entry points to the charlists need to be encoded at the macro level and these entry points in a font-specific input encoding do not even have to coincide with the slot numbers in the output encoding.

In OpenType math fonts, the situation is somewhat different. The underlying input encoding is assumed to consist of Unicode characters. However these Unicode codes are internally mapped to font programs using glyph names, which can be either symbolic (such as `summation` or `integral`) or purely technical (such as `uni2345` or `glyph3456`).

With few exceptions, most of the variant glyphs and building blocks cannot be allocated in standard Unicode slots, so these glyphs have to be mapped to the private use area with font-specific glyph names. In Cambria Math, variant glyphs use suffix names (such as `glyph.vsize<n>` or `glyph.hsize<n>`), while other fonts such as Asana Math use different names (such as `glyphbig<n>` or `glyphwide<n>`).

For font designers developing OpenType math fonts, setting up vertical or horizontal variants is pretty straight-forward, such as

```
summation: summation.vsize1 summation.vsize2 ...
integral : integral.vsize1  integral.vsize2  ...
```

or

```
tildecomb: tildecomb.hsize1 tildecomb.hsize2
```

provided that the variant glyphs use suffix names.

Setting up vertical or horizontal constructions is slightly more complicated, as it also requires some additional information which pieces are of fixed size and which are extensible, such as

```
integral : integralbt:0 uni23AE:1 integraltp:0
```

or

```
arrowboth :
  arrowleft.left:0 uni23AF:1 arrowright.right:0
```

It is interesting to note that some of the building blocks (such as `uni23AE` or `uni23AF`) have Unicode slots by themselves, while others have to placed in the private use area, using private glyph names such as `glyph.left`, `glyph.mid`, or `glyph.right`.

Moreover, vertical or horizontal constructions may also contain multiple extensible parts, such as in the example of over- and underbraces, where the left, middle, and right parts are of fixed size while the extensible part appears twice on either side.

Additional glyph metric information

Besides the global parameters and the instructions for vertical and horizontal variants and constructions, there is yet another kind of information stored in the OpenType MATH table, containing additions to the font metrics of individual glyphs.

In traditional TeX math fonts, the file format of TFM fonts only provides a limited number of fields to store font metric information. As a workaround, certain fields which are needed in math mode only are stored in rather non-intuitive way by overloading fields for other purposes [13].

For example, the nominal width of a glyph is used to store the subscript position, while the italic correction is used to indicate the horizontal offset between the subscript and superscript position.

As a result, the nominal width doesn't represent the actual width of the glyph and the accent position may turn out incorrect. As a secondary correction, fake kern pairs with a so-called skewchar are used to store an offset to the accent position.

In OpenType math fonts, all such non-intuitive ways of storing information can be avoided by using additional data fields for glyph-specific font metric information in the MATH table.

For example, the horizontal offset of the optical center of a glyph is stored in a `top_accent` table, so any adjustments to the placement of math accents can be expressed in a straight-forward way instead of relying on kern pairs with a skewchar.

Similarly, the italic correction is no longer used for the offset between superscripts and subscripts. Instead, the position of indices can be expressed more specifically in a `math_kern` array, representing cut-ins at each corner of the glyphs.

Summary and conclusions

In this paper, we have tried to help improve the understanding of the internals of OpenType math fonts. We have done this in order to contribute to the much-needed development of math support for Latin Modern and TEX Gyre fonts.

In the previous sections, we have discussed the parameters of the OpenType MATH table in great detail, illustrating the similarities and differences between traditional TEX math fonts and OpenType math fonts. However, we have covered other aspects of OpenType math fonts only superficially, as it is impossible to cover everything in one paper.

For a more extensive overview of the features and functionality of OpenType math fonts as well as a discussion of the resulting challenges to font developers, readers are also referred to [12].

In view of the conference motto, it is interesting to note that recent versions of LuaTEX have started to provide a full-featured implementation of OpenType math support in LuaTEX and Context [14, 15], which differs significantly from the implementation of OpenType math support in XƎTEX [10]. In this paper, we have pointed out some of these differences, but further discussions of this topic are beyond of the scope of this paper.

Acknowledgments

The author once again wishes to thank Bogusław Jackowski for permission to reproduce and adapt the figures from his paper *Appendix G Illuminated* [6]. In addition, the author also wishes to acknowledge feedback and suggestions from Taco Hoekwater and Hans Hagen regarding the state of OpenType math support in LuaTEX.

References

[1] Murray Sargent III: Math in Office Blog. http://blogs.msdn.com/murrays/default.aspx

[2] Murray Sargent III: High-quality editing and display of mathematical text in Office 2007. http://blogs.msdn.com/murrays/archive/2006/09/13/752206.aspx

[3] John Hudson, Ross Mills: Mathematical Typesetting: Mathematical and scientific typesetting solutions from Microsoft. Promotional Booklet, Microsoft, 2006. http://www.tiro.com/projects/

[4] George Williams: FontForge. Math typesetting information. http://fontforge.sourceforge.net/math.html

[5] Apostolos Syropoulos: Asana Math. www.ctan.org/tex-archive/fonts/Asana-Math/

[6] Bogusław Jackowski: Appendix G Illuminated. Proceedings of the 16th EuroTEX Conference 2006, Debrecen, Hungary. http://www.gust.org.pl/projects/e-foundry/math-support/tb87jackowski.pdf

[7] Ulrik Vieth: Understanding the æsthetics of math typesetting. *Biuletyn* GUST, 5–12, 2008. Proceedings of the 16th BachoTEX Conference 2008, Bachotek, Poland. http://www.gust.org.pl/projects/e-foundry/math-support/vieth2008.pdf

[8] Microsoft Typography: OpenType specification. Version 1.5, May 2008. http://www.microsoft.com/typography/otspec/

[9] Yannis Haralambous: Fonts and Encodings. O'Reilly Media, 2007. ISBN 0-596-10242-9 http://oreilly.com/catalog/9780596102425/

[10] Will Robertson: The unicode-math package. Version 0.3b, August 2008. http://github.com/wspr/unicode-math/tree/master

[11] Taco Hoekwater: LuaTEX Reference Manual. Version 0.37, 31 March 2009. http://www.luatex.org/svn/trunk/manual/luatexref-t.pdf

[12] Ulrik Vieth: Do we need a 'Cork' math font encoding? TUG*boat*, 29(3), 426–434, 2008. Proceedings of the TUG 2008 Annual Meeting, Cork, Ireland. Reprinted in this MAPS, 3–11. https://www.tug.org/members/TUGboat/tb29-3/tb93vieth.pdf

[13] Ulrik Vieth: Math Typesetting: The Good, The Bad, The Ugly. MAPS, 26, 207–216, 2001. Proceedings of the 12th EuroTEX Conference 2001, Kerkrade, Netherlands. http://www.ntg.nl/maps/26/27.pdf

[14] Taco Hoekwater: Math extensions in LuaTEX. Published elsewhere in this MAPS issue.

[15] Hans Hagen: Unicode math in Context Mk IV. Published elsewhere in this MAPS issue.

Ulrik Vieth
Vaihinger Straße 69
70567 Stuttgart
Germany
ulrik dot vieth (at) arcor dot de

Math in LuaTEX 0.40

Abstract
The math machinery in luaTEX has been completely overhauled in version 0.40. The handling of mathematics in luaTEX has been extended quite a bit compared to how TEX82 (and therefore pdfTEX) handles math. First, luaTEX adds primitives and extends some others so that Unicode input can be used easily. Second, all of TEX82's internal special values (for example for operator spacing) have been made accessible and changeable via control sequences. Third, there are extensions that make it easier to use OpenType math fonts. And finally, there are some extensions that have been proposed in the past that are now added to the engine.

Introduction

We (the luaTEX team) started thinking about OpenType Math support almost immediately after Cambria Math was released, but it took us more than a year to get around to actually writing the implementation. The extensions to the math engine are not complete yet, but there is now enough stuff worthy of publication. This article tries to give a complete overview of all work done so far, but that also means that it is sketchy on details in some places. For the definitive reference, you should read the Math chapter in the luaTEX reference manual.

Pre-existing math primitives

TEX82

Besides the math primitives found in TEX82, luaTEX has support for the extended math primitives that were added by Aleph and XƎTEX.

The TEX82 primitives have been left untouched, except for the fact that when there is a character number needed on the left side of the equation sign (for `\mathcode` and `\delcode`), this number can make use of the full Unicode range.

Typical example code of TEX82 primitives:

```
\mathcode`\+="202B
\delcode`\(="028300
\mathchardef\alpha="010B
\mathchar"1270
\mathaccent"017E
\delimiter"3222378
\radical"270370
```

Aleph

The Aleph math primitives use a syntax that is a fairly straightforward extension of the TEX82 primitives. The difference is that everything has been extended to allow for 16-bit character codes and 256 families. For `\odelcode`, `\odelimiter`, and `\oradical`, this forced the syntax into using two integers for the value to be assigned (because more than 31 bits are needed) but other than that every extension is quite straightforward.

Once again, luaTEX extends the character code on the left side of the equals sign for `\omathcode` and `\odelcode` to the full Unicode range.

Typical example code of Aleph primitives:

```
\omathcode`\+="200002B
\odelcode`\(="000028 "030000
\omathchardef\alpha="001000B
\omathchar"1020070
\omathaccent"001007E
\odelimiter"3020022 "030078
\oradical"020070 "030070
```

XƎTEX

The XƎTEX primitives need to pack even more information: like Aleph, XƎTEX has 256 math families, but each of those is encoded using the full Unicode range. This makes it hard to come up with a nice hexadecimal notation, so instead the values are split up into their class, family, slot segments, for example:

```
\def\overbrace {\Umathaccent 0 1 "23DE }
```

When a math class is required, this is given by the first integer, which ranges from 0 to 7. The next integer is the family number and ranges from 0 to 255. The last integer is the Unicode code point, which ranges from 0 to hexadecimal 0x10FFFF (1,114,111 in decimal).

There are always just two or three integers needed, because XƎTEX never bothers to list ‘small’ and ‘large’ versions of delimiters. The use of large vs. small items is controlled via OpenType font parameters.

LuaTEX includes primitives that are fully compatible with their XeTEX counterparts except for their names. Where XeTEX uses the \XeTeX prefix, luaTEX uses \U.

Typical example code of XeTEX-compatible primitives:

```
\Umathcode`\+="2 "0 "2B
\Udelcode`\(= "0 "28
\Umathchardef\alpha="0 "1 "B
\Umathchar "1 "2 "70
\Umathaccent "0 "1 "7E
\Udelimiter "3 "2 "22
\Uradical "2 "70
```

For the sake of completeness, the 'packed' XeTEX primitives \Umathcharnum, \Umathcodenum and \Udelcodenum are also provided, but their use is discouraged.

General new math extensions

Cramped math styles

TEX's math engine has four main math styles: display style, text style, script style, and scriptscript style. Each of those four main styles can also appear in a 'cramped' form that is suitable for use in situations where something lives on top of the current sub-formula (like in the denominator part of a fraction). This makes for a total of eight styles. In TEX82, it is possible to force a particular main math style by using one of these primitives:

```
\displaystyle
\textstyle
\scriptstyle
\scriptscriptstyle
```

However, until now it was not possible to explicitly switch to one of the cramped modes. For this, luaTEX adds the following four new primitives:

```
\crampeddisplaystyle
\crampedtextstyle
\crampedscriptstyle
\crampedscriptscriptstyle
```

Math characters in text mode

LuaTEX allows \mathchar, \omathchar, and \Umathchar and control sequences that are the result of \mathchardef, \omathchardef, or \Umathchardef outside math mode.

When luaTEX sees an object like this, it uses the \textfont from the requested math family to produce a normal glyph node.

For example, assume that \alpha is defined as before and that \omega is defined as a \mathchardef with value "121. Further assume that \textfont1 is \teni (as it is in the plain macros). Under these conditions,

```
From \alpha\ to \omega.
```

and

```
From {\teni\char"B} to {\teni \char"21}.
```

are equivalent. Both will produce:

From α to ω.

Querying the math style

The new expandable primitive \mathstyle returns a value between 0 and 7 (in math mode), or −1 (all other modes). The returned number represents the current math style value.

Higher numbers represent smaller styles: 0 stands for \displaystyle, 1 for \crampeddisplaystyle, and 7 for \crampedscriptscriptstyle.

Using these new primitives, you can write code like this:

```
\def\uncramped#1{{\ifcase\mathstyle
    \or \displaystyle      \or
    \or \textstyle         \or
    \or \scriptstyle       \or
    \or \scriptscriptstyle \fi  #1}}
```

or even create a fully expandable version of \mathchoice:

```
\def\mathchoice#1#2#3#4{{\ifcase\mathstyle
    #1\or #1\or
    #2\or #2\or
    #3\or #3\or
    #4\or #4\fi}}
```

To make it easier to test the return value of \mathstyle, the four old and the four new math style commands have been altered so that they can be used as numeric values for testing. This allows constructs like this:

```
\ifnum\mathstyle=\textstyle
   \message{normal text style}
\fi
```

But there is a small catch: there are a few primitives (\over, \atop, \overwithdelims, \atopwithdelims) in TEX82 where the style that will be used is not known at the start, and these commands would therefore return wrong values for \mathstyle.

To make it possible to get the correct math style in all cases, luaTEX introduces the new primitive `\Ustack`, that can (should) be used as a prefix for the commands given above.

```
$\Ustack { ... a ... \over ... b ... }$
```

The `\Ustack` command will make sure that `\mathstyle` is returning the correct values, even inside the `... a ...` branch. A `\Ustack` can be nested inside another, if needed.

Bottom accents

Besides the normal top accents, luaTEX also supports bottom accents in math mode. For bottom accents, there is the new primitive `\Umathbotaccent`. For combined top and bottom accents, there is `\Umathaccents`. The latter takes two math accent specifications. Like all the new primitives that actively scan for mathchars or delimititers, these use the XƎTEX-style syntax:

```
$$
\Umathbotaccent"0"0"323 A
\Umathaccents "0"0"20D7 "0"0"323 A
$$
```

$$\underset{.}{A}\overrightarrow{\underset{.}{A}}$$

Horizontal extenders

On top of the normal vertical extensibles, luaTEX also has support for horizontal extensibles. This is particularly useful for wide accents, as the following example shows:

```
\def\overarrow{\Umathaccent"0"0"20D7}
$$
\overarrow{a+b+c+d+e}
$$
```

$$\overrightarrow{a+b+c+d+e}$$

Note that this feature depends on support from the math font that is being used. This article is typeset using MicroSoft's Cambria Math font and that actually has this support built in, but so far none of the standard TEX fonts provide the needed information.

Math parameters

In luaTEX, the font dimension parameters that TEX82 uses in math typesetting are now accessible via primitive commands. These parameters are initialized from the math fonts, or can be set by the user via explicit commands. Each math parameter exists in eight versions that match the math styles. Re-factoring of the math engine has resulted in more parameters than were accessible before, even when taking the font dimensions of the math fonts into account.

Math parameter commands

Each of the math parameters (the full list is given in table 1 at the end of this article) can be set by an explicit command, like this:

```
\Umathquad\displaystyle=1em
```

Such settings obey grouping, but only one value can be in effect for a single formula, and that is decided upon when the closing dollar sign is read in. Here is an example:

```
\centerline{
$
\Ustack{a \over b} ×  b
$ \kern 50pt $
\Umathfractiondenomvgap \textstyle = 8pt
\Ustack{a \over b} ×  b
$}
```

$$\frac{a}{b} \times b \qquad\qquad \frac{a}{\vphantom{x}\\b} \times b$$

You can use `\the\Umathquad\displaystyle` if the current value is needed (for example inside a space fine-tuning macro).

Font-based Math Parameters

While it is nice to have these math parameters available for tweaking, it would be tedious to have to set each of them by hand. For this reason, luaTEX initializes (almost) all these parameters whenever you assign a font identifier to a math family. This is based either on the traditional math font dimensions in the font (for assignments to math family 2 and 3 using TFM-based fonts like `cmsy` and `cmex`), or on the named values in a 'MathConstants' table (when an OpenType math font is loaded via Lua). If there is a 'MathConstants' table, this takes precedence over font dimensions, and in that case no attention is paid to which family is being assigned to: the 'MathConstants' tables in the last assigned family sets all parameters.

The eight math parameters are typically set by using the `\textfont` value for the display and text styles (cramped and normal), `\scriptfont` for the script styles, and `\scriptscriptfont` for the scripscript styles. In table 2 these automatic mappings are shown. Besides the parameters listed in that table, luaTEX also looks at the 'space' font dimension parameter. For math fonts, this should be set to zero.

Math spacing parameters

Inter-element math spacing in TEX82 is controlled by the 8 × 8 table of spacing values that is given in Chapter 18 of the TEXbook. In luaTEX, this table has been converted into 64 primitives of the form `\Umath...spacing`, for all the paired combinations of `bin`, `rel`, `ord`, `open`, `close`, `punct`, `inner`, and `op`. Here is an example:

```
\centerline{$
a × b
$ \kern 50pt $
\Umathordbinspacing \textstyle = 18mu
\Umathbinordspacing \textstyle = \thickmuskip
\thickmuskip = 10mu
a × b
$}
```

$a \times b$ $\qquad\qquad$ $a \quad \times \quad b$

Normally, one would assign explicit `mu` dimensions to these parameters, but a special case arises when the predefined muskip registers are used. When the assignment uses `\thinmuskip`, `\medmuskip`, or `\thickmuskip`, late binding is used, so that later (re)assignments to one of these registers is taken into account.

Verbose versions of character commands

luaTEX defines six new primitives that have the same function as ^, _, $, and $$.

primitive	explanation
`\Usuperscript`	for the functionality of ^
`\Usubscript`	for the functionality of _
`\Ustartmath`	for $, outside math.
`\Ustopmath`	for $, inside inline math.
`\Ustartdisplaymath`	for $$, outside math.
`\Ustopdisplaymath`	for $$, inside display math.

The `\Ustopmath` and `\Ustopdisplaymath` primitives check if the current math mode is the correct one (inline vs. displayed), but you can freely intermix the four mathon/mathoff commands with explicit dollar sign(s).

Lua math extensions

Setting and getting math parameters

The lua functions `tex.setmath()` and `tex.getmath()` can be used to get or set the internal math parameters.

To set a math parameter, use `tex.setmath()`:

```
tex.setmath(<string> n, <string> t, <number> n)
```

or

```
tex.setmath('global',
           <string> n, <string> t, <number> n)
```

In an attempt to cut down the verbosity level, the first string is the parameter name minus the leading 'Umath', and the second string is the style name minus the trailing 'style', for example:

```
tex.setmath('fractiondenomvgap','text',8*65536)
```

An optional first parameter can be given with the explicit string `'global'`, which indicates a global assignment. For now, you cannot use Lua for the math object spacing parameters (because there is no read interface for 'mu' lengths defined yet).

Querying a math parameter uses the inverse function `tex.getmath()`:

```
<number> n = tex.getmath(<string> n, <string> t)
```

which should not need further explanation.

Attributes in math mode

Starting with luaTEX 0.40, node attributes are now remembered in math mode even after the conversion from math back to the horizontal list that is eventually added to the typeset paragraph. New nodes that are created in this process (like the horizontal rule in a fraction) inherit their attributes from the most logical parent node.

The 'mlist_to_hlist' callback

A simple callback is offered that can be used to alter last-minute things in the math node list. When you use this callback, you have to run the math to hlist conversion process yourself. To make this easier, there is a builtin function that does exactly what luaTEX would have done if there was no callback set.

First, here is the syntax diagram for the callback:

```
function(<node> head,
         <string> displaytype,
         <boolean> need_penalties)
    return <node> newhead
end
```

The returned node has to be the head of the list that will be added to the vertical or horizontal list, the string `displaytype` argument is either 'text' or 'display' depending on the current math mode, the boolean `need_penalties` argument is `true` if penalties have to be inserted into the generated hlist, `false` otherwise.

If all you want to do is alter a few small things, than the easiest approach is to make those alterations first, and then call the following helper function:

```
<node> h = node.mlist_to_hlist(
              <node> n,
              <string> displaytype,
              <boolean> penalties )
```

This runs the internal mlist to hlist conversion, converting the math list in `n` into the horizontal list `h`. The interface is exactly the same as for the callback `mlist_to_hlist`, so that a simple working callback is this one:

```
callback.register ('mlist_to_hlist',
  function (h,d,n)
    return node.mlist_to_hlist(h,d,n)
  end )
```

OpenType Math features

As explained by Ulrik Vieth's articles on the subject, there is much more to 'OpenType Math' than just being able to handle Unicode input characters. A number of extensions have been made to luaTEX to handle specific features of OpenType Math.

OpenType font metrics

First, lets talk about OpenType font metrics. OpenType fonts in luaTEX are always loaded via lua code in the `define_font` callback, and OpenType Math fonts are no exception.

The lua function `fontloader.to_table()` outputs the OpenType Math information in two parts: there is a global part where the Math Constants are listed, and a per-glyph local part with data like italic correction and extensible recipe structures.

The global part looks like this:

```
["math"]={
 ["AxisHeight"]=585,
 ...
 ["FractionDenominatorDisplayStyleGapMin"]=260,
 ["FractionDenominatorGapMin"]=133,
 ["FractionDenominatorShiftDown"]=1030,
 ["FractionNumeratorDisplayStyleGapMin"]=260,
 ["FractionNumeratorDisplayStyleShiftUp"]=1550,
 ["FractionNumeratorGapMin"]=133,
 ...
 ["ScriptPercentScaleDown"]=73,
 ["ScriptScriptPercentScaleDown"]=60,
 ...
```

The full list is longer than this of course; all the math constants are listed in that table. Except for a few cases with the word 'Percent' in the name, values are expressed in design units, and these have to be converted by Lua code into scaled points before being passed back to luaTEX (as is the case for all font dimensions).

The example above is taken from Cambria Math which is a Truetype format font with 2048 design units per em, so the actual value of 'AxisHeight' if the font is loaded at 10pt would be

$$585/2048 * 10\text{pt} = 187\,200\text{sp}$$

The local part is easier to explain in two steps, because not all glyphs have the same set of extended information. The first example shows the relevant part of the data for the Unicode character 'MATHEMATICAL ITALIC SMALL F', f:

```
 {
   ["name"]="u1D453",
   ["italic_correction"]=60,
   ["mathkern"]={
    ["bottom_right"]={
     {
       ["height"]=420,
       ["kern"]=-400,
     },
     {
       ["height"]=720,
       ["kern"]=-320,
     },
     {
       ["height"]=1020,
       ["kern"]=0,
     },
    },
    ["bottom_left"]={ ... }
    ["top_right"]={ ... }
   },
   ["top_accent"]=840,
   ...
 },
```

As you can see, it has an italic correction of 60 design units, it has an entry `top_accent` that is used for the placement of math accents on top of the glyph, and it has a `mathkern` subtable. The `mathkern` table is used for super- and subscript placement: it can define kerning corrections for each of the four corners of the glyph. Any of those can be missing, in which case no correction is needed (this is the case for the `top_left` side of this glyph).

The next example shows part of the metric data for SQUARE ROOT, the extensible character that represents the root sign, √:

```
{
  ["name"]="radical",
  ["vert_variants"]={
   ["italic_correction"]=0,
   ["parts"]={
    {
      ["component"]="uni23B7",
      ["advance"]=2743,
      ["end"]=2500,
    },
    {
      ["component"]="uni20D3",
      ["advance"]=1211,
      ["end"]=1150,
      ["extender"]=1,
      ["start"]=1150,
    },
    {
      ["component"]="radical.top",
      ["advance"]=1211,
      ["start"]=600,
     }
    },
   ["variants"]="radical radical.vsize1 \
                radical.vsize2 radical.vsize3\
                radical.vsize4 radical.vsize5"
  }
  ...
},
```

This glyph does not have either of the mathkern and top_accent extended information items that were present in the previous example, but it does have a vert_variants subtable. This table contains information for extensible recipes, split into three parts:

- the variants string gives a sequence of versions of this glyph with ever increasing size,
- the parts table lists the extensible parts,
- the italic_correction gives the italic correction that is needed for a glyph constructed from such parts.

The actual metric details will be explained later.

OpenType Math family sizes and ssty

When using OpenType Math fonts, it is important to load the \scriptfont and \scriptscriptfont at the sizes that are requested by the font designer (via ScriptPercentScaleDown and ScriptScriptPercentScaleDown).

If the font provides an ssty feature, then it is advisable to enable that feature (with values ssty=1 for script size and ssty=2 for script script size). The difference between doing so and simply loading the Cambria Math font at the 'normal' values of 70% and 50% can be seen in this example:

x^{x^x}	x^{x^x}
cambria 7pt/5pt	cambria 73%/60%+ssty

Extra Math Parameters

OpenType Math fonts have a few extra parameters compared to traditional TFM-based math fonts, and the effects of those extra parameters can be quite noticeable. The example below shows the difference between using the actual \Umathfractiondenomvgap and \Umathfractionnumvgap from Cambria Math versus the hardwired TEX82 value of three times default_rule_thickness:

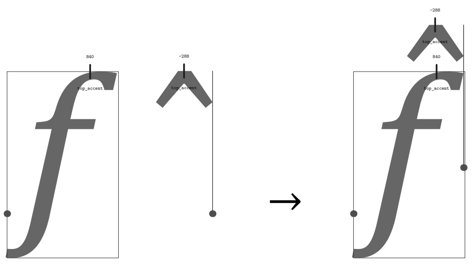
$\dfrac{p_p}{b^b}$	$\dfrac{p_p}{b^b}$
TEX82, 3*rule thickness	luaTEX, from font

Math accent placements

When a math (top) accent has to be placed and the accentee is a character that has a non-zero top_accent value, then this value will be used to place the accent instead of the \skewchar kern used by TEX82.

The top_accent value represents a vertical line somewhere in the accentee. The accent will be shifted horizontally such that its own top_accent line coincides with the one from the accentee. If the top_accent value of the accent is zero, then half the width of the accent followed by its italic correction is used instead.

In a picture, it looks like this:



The vertical placement of a top accent depends on the x_height of the font of the accentee (as explained in the TEXbook), but if that value turns out to be zero and the font has a ‘MathConstants’ table, then `AccentBaseHeight` is used instead.

If a math bottom accent has to be placed, the `bot_accent` value is checked instead of `top_accent`. Because bottom accents do not exist in TEX82, the `\skewchar` kern is ignored.

The vertical placement of a bottom accent is straight below the accentee, no correction takes place.

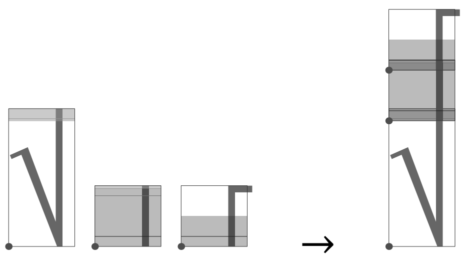
Also, remember that luaTEX has horizontal extensibles, and when present, these will be used by the accent placement primitives to build up longer versions when that is needed.

Overlapping extensibles

In TFM based fonts, extensible bits of glyphs are placed butt to butt, which normally works fine when printing, but often creates problems with PDF previewers. If you are looking at this article in PDF format, you will likely see small gaps appearing in the left side of the following example:

... but not on the right side, because the right side uses OpenType extensible recipes where a certain amount of overlap is built in.

Recall the `vert_variants` metrics representation that was listed earlier. Each of the `parts` had an `advance` key, but also type `start` and/or `end`. These latter two are combined with the ‘MathConstants’ value `MinConnectorOverlap` to define overlap zones. Once again here is an image to demonstrate the effect (the actual algorithm is documented in the OpenType Math specification, which is followed by luaTEX).



Extensible big operators

In OpenType Math (and therefore also in luaTEX), big operators can come in more than just the two sizes provided by TEX82. Big operators can even be build up from extensible parts.

Normally, the OpenType font designer decides the size that is used in display mode via the ‘MathConstants’ table, but it can be fun to change the used value manually. For example:

```
\Umathoperatorsize\displaystyle = 15pt
$$\sum_{k=2}^4 k^2 = 2^2 + 3^2 + 4^2 = 29$$
\Umathoperatorsize\displaystyle = 55pt
$$\sum_{k=2}^4 k^2 = 2^2 + 3^2 + 4^2 = 29$$
```

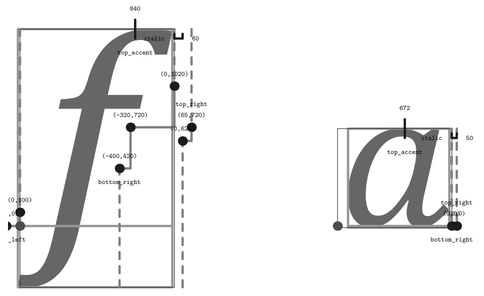
$$\sum_{k=2}^4 k^2 = 2^2 + 3^2 + 4^2 = 29$$

$$\sum_{k=2}^4 k^2 = 2^2 + 3^2 + 4^2 = 29$$

Script placements

As seen earlier, the character entries in an OpenType Math font can have a ‘mathkern’ table.

The ‘mathkern value’ at a specific height is the kern value that is specified by the next higher height and kern pair, or the highest one in the character (if there is no value high enough in the character), or simply zero (if the character has no mathkern pairs at all).



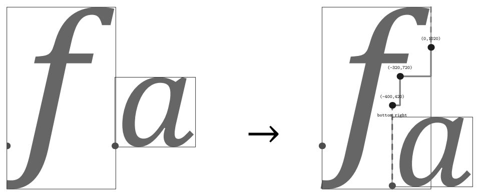
When a super- or subscript has to be placed next to a math item, luaTEX checks whether the super- or subscript and the nucleus are both simple character items. If they are, and if the fonts of both character items are OpenType fonts (as opposed to legacy TEX fonts), then luaTEX will use the OpenTypeMath algorithm for deciding on the horizontal placement of the super- or subscript. This works as follows:

- The vertical position of the script is calculated.
- The default horizontal position is flat next to the base character.
- For superscripts, the italic correction of the base character is added.
- For a superscript, two vertical values are calculated: the bottom of the script (after shifting up),

and the top of the base. For a subscript, the two values are the top of the (shifted down) script, and the bottom of the base.

- For each of these two locations, luaTEX:
  - finds out the 'mathkern value' at this height for the base (for a subscript placement, this is the `bottom_right` corner, for a superscript placement the `top_right` corner)
  - finds out the 'mathkern value' at this height for the script (for a subscript placement, this is the `top_left` corner, for a superscript placement the `bottom_left` corner)
- The horizontal kern to be applied is the smallest of the two results from previous step.

A picture should help make this clearer:



Legends on extensible items

The new `\Uunderdelimiter` and `\Uoverdelimiter` primitives allow the placement of a subscript or superscript on an extensible item and the complementary `\Udelimiterunder` and `\Udelimiterover` primitives allow the placement of an extensible item as a subscript or superscript on a nucleus.

In all four primitives, the vertical placements are controlled by `\...bgap` and `\...vgap` parameters using a similar method as for limit placements on large operators. The superscript in `\Uoverdelimiter` is typeset in a suitable scripted style, the subscript in `\Uunderdelimiter` is cramped as well.

```
$$
A \mathrel{\Uoverdelimiter 0 "2192 {a+b}}
B \mathrel{\Uunderdelimiter 0 "2192 {a+b}} C
$$
```

$$A \xrightarrow{a+b} B \xrightarrow[a+b]{} C$$

```
$$\Udelimiterover 0 "23DE {a+b}
+ \Udelimiterunder 0 "23DF {a+b} = C $$
```

$$\overbrace{a+b} + \underbrace{a+b} = C$$

Here it is the delimiter that is typeset in a script style.

Radicals with degrees

The new primitive `\Uroot` allows the direct construction of a radicals including a degree. Its syntax is a straightforward extension of `\Uradical`:

```
\Uradical <fam> <char> <radicand>
\Uroot    <fam> <char> <degree> <radicand>
```

The placement of the degree is controlled by the math parameters `\Umathradicaldegree...`, and the degree is typeset in `\scriptscriptstyle`.

```
$$
\Uroot 0 "221A {3}{x^3+y^3}
$$
```

$$\sqrt[3]{x^3+y^3}$$

This bit of the OpenType Math specification is a literal conversion of the plain TEX macro `\root \of`, with the important difference being that it shifts the adhoc values in the plain macro to the font, so that the font designer can come up with nice looking values.

In luaTEX, this functionality could have been implemented by a macro. However, doing so felt clumsy because of the need to take care of the different sizes.

Open OpenType issues

There are a few remaining problems that have not been dealt with at the time of the writing of this article:

- It is not clear how `\atopwithdelims` and `\overwithdelims` should be implemented with OpenType Math fonts. For the moment, it is best to avoid using these primitives.
- It is unclear whether stretch stacks (the four primitives explained in the 'Legends on extensible items' section) should be centered on the math axis or not.
- Some confusion remains on what the Math constant 'DelimitedSubFormulaMinHeight' is meant to represent.

Two features of OpenType Math have not been implemented yet:

- Skewed (text-style) fractions.
- Flattened accents for high characters.

These, as well as some other math extensions, are planned for the luaTEX 0.50 release.

Taco Hoekwater

Primitive name	**Description**
`\Umathquad`	the width of 18mu's
`\Umathaxis`	height of the vertical center axis of the math formula above the baseline
`\Umathoperatorsize`	minimum size of large operators in display mode
`\Umathoverbarkern`	vertical clearance above the rule
`\Umathoverbarrule`	the width of the rule
`\Umathoverbarvgap`	vertical clearance below the rule
`\Umathunderbarkern`	vertical clearance below the rule
`\Umathunderbarrule`	the width of the rule
`\Umathunderbarvgap`	vertical clearance above the rule
`\Umathradicalkern`	vertical clearance above the rule
`\Umathradicalrule`	the width of the rule
`\Umathradicalvgap`	vertical clearance below the rule
`\Umathradicaldegreebefore`	the forward kern that takes place before placement of the radical degree
`\Umathradicaldegreeafter`	the backward kern that takes place after placement of the radical degree
`\Umathradicaldegreeraise`	this is the percentage of the total height and depth of the radical sign that the degree is raised by. It is expressed in `percents`, so 60% is expressed as the integer 60.
`\Umathstackvgap`	vertical clearance between the two elements in a `\atop` stack
`\Umathstacknumup`	numerator shift upward in `\atop` stack
`\Umathstackdenomdown`	denominator shift downward in `\atop` stack
`\Umathfractionrule`	the width of the rule in a `\over`
`\Umathfractionnumvgap`	vertical clearance between the numerator and the rule
`\Umathfractionnumup`	numerator shift upward in `\over`
`\Umathfractiondenomvgap`	vertical clearance between the denominator and the rule
`\Umathfractiondenomdown`	denominator shift downward in `\over`
`\Umathfractiondelsize`	minimum delimiter size for `\...withdelims`
`\Umathlimitabovevgap`	vertical clearance for limits above operators
`\Umathlimitabovebgap`	vertical baseline clearance for limits above operators
`\Umathlimitabovekern`	space reserved at the top of the limit
`\Umathlimitbelowvgap`	vertical clearance for limits below operators
`\Umathlimitbelowbgap`	vertical baseline clearance for limits below operators
`\Umathlimitbelowkern`	space reserved at the bottom of the limit
`\Umathoverdelimitervgap`	vertical clearance for limits above delimiters
`\Umathoverdelimiterbgap`	vertical baseline clearance for limits above delimiters
`\Umathunderdelimitervgap`	vertical clearance for limits below delimiters
`\Umathunderdelimiterbgap`	vertical baseline clearance for limits below delimiters
`\Umathsubshiftdrop`	subscript drop for boxes and sub-formulas
`\Umathsubshiftdown`	subscript drop for characters
`\Umathsupshiftdrop`	superscript drop (raise, actually) for boxes and sub-formulas
`\Umathsupshiftup`	superscript raise for characters
`\Umathsubsupshiftdown`	subscript drop in the presence of a superscript
`\Umathsubtopmax`	the top of standalone subscripts cannot be higher than this above the baseline
`\Umathsupbottommin`	the bottom of standalone superscripts cannot be less than this above the baseline
`\Umathsupsubbottommax`	the bottom of the superscript of a combined super- and subscript be at least as high as this above the baseline
`\Umathsubsupvgap`	vertical clearance between super- and subscript
`\Umathspaceafterscript`	additional space added after a super- or subscript
`\Umathconnectoroverlapmin`	minimum overlap between parts in an extensible recipe

Table 1 Named math parameters.

Variable	Default value (OpenType)	Default value (TFM)
\Umathaxis	AxisHeight	axis_height
\Umathoperatorsize	MinimumDisplayOperatorHeight	<not set>
\Umathfractiondelsize	0	delim1, delim2
\Umathfractiondenomdown	FractionDenominator[DisplayStyle]ShiftDown	denom1, denom2
\Umathfractiondenomvgap	FractionDenominator[DisplayStyle]GapMin	3*rule, rule
\Umathfractionnumup	FractionNumerator[DisplayStyle]ShiftUp	num1, num2
\Umathfractionnumvgap	FractionNumerator[DisplayStyle]GapMin	3*rule, rule
\Umathfractionrule	FractionRuleThickness	rule
\Umathlimitabovebgap	UpperLimitBaselineRiseMin	big_op_spacing3
\Umathlimitabovekern	0	big_op_spacing5
\Umathlimitabovevgap	UpperLimitGapMin	big_op_spacing1
\Umathlimitbelowbgap	LowerLimitBaselineDropMin	big_op_spacing4
\Umathlimitbelowkern	0	big_op_spacing5
\Umathlimitbelowvgap	LowerLimitGapMin	big_op_spacing2
\Umathoverdelimitervgap	StretchStackGapBelowMin	big_op_spacing1
\Umathoverdelimiterbgap	StretchStackTopShiftUp	big_op_spacing3
\Umathunderdelimitervgap	StretchStackGapAboveMin	big_op_spacing2
\Umathunderdelimiterbgap	StretchStackBottomShiftDown	big_op_spacing4
\Umathoverbarkern	OverbarExtraAscender	rule
\Umathoverbarrule	OverbarRuleThickness	rule
\Umathoverbarvgap	OverbarVerticalGap	3*rule
\Umathquad	<font_size(f)>	math_quad
\Umathradicalkern	RadicalExtraAscender	rule
\Umathradicalrule	RadicalRuleThickness	<not set>
\Umathradicalvgap	Radical[DisplayStyle]VerticalGap	(rule+(abs(math_x)/4)), (rule+(abs(rule)/4))
\Umathradicaldegreebefore	RadicalKernBeforeDegree	<not set>
\Umathradicaldegreeafter	RadicalKernAfterDegree	<not set>
\Umathradicaldegreeraise	RadicalDegreeBottomRaisePercent	<not set>
\Umathspaceafterscript	SpaceAfterScript	script_space
\Umathstackdenomdown	StackBottom[DisplayStyle]ShiftDown	denom1, denom2
\Umathstacknumup	StackTop[DisplayStyle]ShiftUp	num1, num3
\Umathstackvgap	Stack[DisplayStyle]GapMin	7*rule, 3*rule
\Umathsubshiftdown	SubscriptShiftDown	sub1
\Umathsubshiftdrop	SubscriptBaselineDropMin	sub_drop
\Umathsubsupshiftdown	SubscriptShiftDown[WithSuperscript]	sub2
\Umathsubtopmax	SubscriptTopMax	(abs(math_x*4)/5)
\Umathsubsupvgap	SubSuperscriptGapMin	4*rule
\Umathsupbottommin	SuperscriptBottomMin	(abs(math_x)/4)
\Umathsupshiftdrop	SuperscriptBaselineDropMax	sup_drop
\Umathsupshiftup	SuperscriptShiftUp[Cramped]	sup1, sup2, sup3
\Umathsupsubbottommax	SuperscriptBottomMaxWithSubscript	(abs(math_x*4)/5)
\Umathunderbarkern	UnderbarExtraDescender	rule
\Umathunderbarrule	UnderbarRuleThickness	rule
\Umathunderbarvgap	UnderbarVerticalGap	3*rule
\Umathconnectoroverlapmin	MinConnectorOverlap	0

Table 2 Initialization of math parameters from font information. In the last column, 'rule' stands for default_rule_thickness, and 'math_x' stands in for math_x_height. Where multiple values or square brackets are present some of the eight parameter instances are based on some values and others on other values. See the luaTEX reference manual for more detailed information.

Unicode Math in ConTEXt

Abstract
This article is complementary to Taco Hoekwater's article about the upgrade of the math subsystem in LuaTEX. In parallel (also because we needed a testbed) the math subsystem of ConTEXt has been upgraded. In this article I will describe how we deal with Unicode math using the regular Latin Modern and TEXGyre fonts and how we were able to clean up some of the more nasty aspects of math.

Introduction

The LuaTEX project entered a new stage when end of 2008 and beginning of 2009 math got opened up. Although TEX can handle math pretty well we had a few wishes that we hoped to fulfill in the process. TEX's math machinery is a rather independent subsystem. This is reflected in the fact that after parsing there is an intermediate list of so called noads (math elements), which then gets converted into a node list (glyphs, kerns, penalties, glue and more). This conversion can be intercepted by a callback and a macro package can do whatever it likes with the list of noads as long as it returns a proper list.

Of course ConTEXt does support math and that is visible in its code base:

- Due to the fact that we need to be able to switch to alternative styles the font system is quite complex and in ConTEXt MkII math font definitions (and changes) are good for 50% of the time involved. In MkIV we can use a more efficient model.
- Because some usage of ConTEXt demands the mix of several completely different encoded math fonts, there is a dedicated math encoding subsystem in MkII. In MkIV we will use Unicode exclusively.
- Some constructs (and symbols) are implemented in a way that we find suboptimal. In the perspective of Unicode in MkIV we aim at all symbols being real characters. This is possible because all important constructs (like roots, accents and delimiters) are supported by the engine.
- In order to fit vertical spacing around math (think for instance of typesetting on a grid) in MkII we have ended up with rather messy and suboptimal code. (This is because spacing before and after formulas has to cooperate with spacing of structural components that surround it.) The expectation is that we can improve that.

In the following sections I will discuss a few of the implementation details of the font related issues in MkIV. Of course a few years from now the actual solutions we implemented might look different but the principles remain the same. Also, as with other components of LuaTEX Taco and I worked in parallel on the code and its usage, which made the tasks easier for both of us.

Transition

In TEX, math typesetting uses a special concept called families. Each math component (number, letter, symbol, et cetera) is member of a family. Because we have three sizes (text, script and scriptscript) this results in a family–size matrix of defined fonts. The number of glyphs in a font was limited to 256, which meant that we had quite some font definitions. The minimum number of families was 4 (roman, italic, symbol, and extension) but in practice several more could be active (sans, bold, mono-spaced, more symbols, et cetera) for specific alphabets or extra symbols (for instance ams set A and B). The total number of families in traditional TEX is limited to 16, and one easily hits this maximum. In that case, some 16 times 3 fonts are defined for one size of which in practice only a few are really used in the typesetting.

A potential source of confusion is bold math. Bold in math can either mean having some bold letters, or having the whole formula in bold. In practice this means that for a complete bold formula one has to define the whole lot using bold fonts. A complication is that the math symbols are kind of bound to families and so we end up with either redefining symbols, or reusing the families (which is easier and faster). In any case there is a performance issue involved due to the rather massive switch from normal to bold.

In Unicode all alphabets that make sense, as well as all math symbols are part of the definition, although unfortunately some alphabets have their letters spread over the Unicode vector and not in a range (like blackboard). This forces all applications that want to support math to implement similar hacks to deal with it.

In MkIV we will assume that we have Unicode aware math fonts, like OpenType. The font that sets the standard is Microsoft Cambria. The upcoming (I'm writing this in January 2009) TEXGyre fonts will be compliant to this standard but they're not yet there and so we have a problem. The way out is to define virtual fonts and now that LuaTEX math is extended to cover all of Unicode, as well as provides access to the (intermediate) math lists, this has become feasible. This also permits us to test LuaTEX with both Cambria and Latin Modern Virtual Math.

The advantage is that we can stick to just one family for all shapes which simplifies the underlying TEX code enormously. First of all we need to define way less fonts (which is partially compensated by loading them as part of the virtual font) and all math aspects can now be dealt with using the character data tables.

One tricky aspect of the new approach is that the Latin Modern fonts have design sizes, so we have to define several virtual fonts. On the other hand, fonts like Cambria have alternative script and scriptscript shapes which is controlled by the `ssty` feature, a gsub alternate that provides some alternative sizes for a couple of hundred characters that matter.

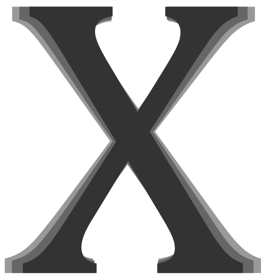
text	`lmmi12 at 12pt`	`cambria at 12pt with ssty=no`
script	`lmmi8 at 8pt`	`cambria at 8pt with ssty=1`
scriptscript	`lmmi6 at 6pt`	`cambria at 6pt with ssty=2`

So Cambria not so much has design sizes but shapes optimized relative to the text variant: in the following example we see text in red, script in green and scriptscript in blue.

```
\definefontfeature[math][analyze=false,script=math,language=dflt]

\definefontfeature[text]        [math][ssty=no]
\definefontfeature[script]      [math][ssty=1]
\definefontfeature[scriptscript][math][ssty=2]
```

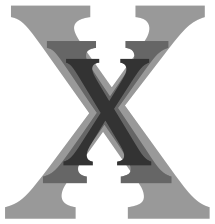
Let us first look at Cambria:

```
\startoverlay
    {\definedfont[name:cambriamath*scriptscript at 150pt]\mkblue  X}
    {\definedfont[name:cambriamath*script       at 150pt]\mkgreen X}
    {\definedfont[name:cambriamath*text         at 150pt]\mkred   X}
\stopoverlay
```



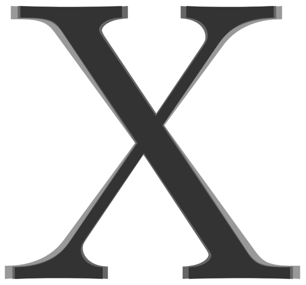
When we compare them scaled down as happens in real script and scriptscript we get:

```
\startoverlay
    {\definedfont[name:cambriamath*scriptscript at 120pt]\mkblue  X}
    {\definedfont[name:cambriamath*script       at  80pt]\mkgreen X}
    {\definedfont[name:cambriamath*text         at  60pt]\mkred   X}
\stopoverlay
```



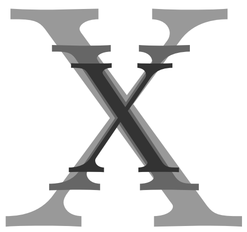
Next we see (scaled) Latin Modern:

```
\startoverlay
    {\definedfont[LMRoman8-Regular  at 150pt]\mkblue  X}
    {\definedfont[LMRoman10-Regular at 150pt]\mkgreen X}
    {\definedfont[LMRoman12-Regular at 150pt]\mkred   X}
\stopoverlay
```



In practice we will see:

```
\startoverlay
    {\definedfont[LMRoman8-Regular  at 120pt]\mkblue  X}
    {\definedfont[LMRoman10-Regular at  80pt]\mkgreen X}
    {\definedfont[LMRoman12-Regular at  60pt]\mkred   X}
\stopoverlay
```



Both methods probably work out well, although you need to keep in mind that the OpenType `ssty` feature is not so much a design size related feature.

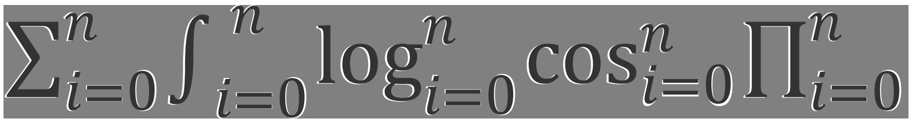
An OpenType font can have a specification for the script and scriptscript size. By default we listen to this specification instead of the one imposed by the bodyfont environment. When you turn on tracing

```
\enabletrackers[otf.math]
```

you will get messages like:

```
asked scriptscript size: 458752, used: 471859.2 (102.86 %)
asked script size: 589824, used: 574095.36 (97.33 %)
```

The differences between the defaults and the font recommendations are not that large so by default we listen to the font specification.



In this overlay the white text is scaled according to the specification in the font, while the black text is scaled according to the bodyfont environment (12/7/5 points).

Going virtual

The number of math fonts (used) in the TEX community is relatively small and of those only Latin Modern (which builds upon Computer Modern) has design sizes. This means that the amount of Unicode compliant virtual math fonts that we have to make is not that large. We could have used an already present virtual composition mechanism but instead we made a handy helper function that does a more efficient job. This means that a definition looks (a bit simplified) as follows:

```
mathematics.make_font ( "lmroman10-math", {
  { name="lmroman10-regular", features="virtualmath", main=true },
  { name="lmmi10", vector="tex-mi", skewchar=0x7F },
  { name="lmsy10", vector="tex-sy", skewchar=0x30, parameters=true } ,
  { name="lmex10", vector="tex-ex", extension=true } ,
  { name="msam10", vector="tex-ma" },
  { name="msbm10", vector="tex-mb" },
  { name="lmroman10-bold", "tex-bf" } ,
  { name="lmmib10", vector="tex-bi", skewchar=0x7F } ,
  { name="lmsans10-regular", vector="tex-ss", optional=true },
  { name="lmmono10-regular", vector="tex-tt", optional=true },
} )
```

For the TEXGyre Pagella it looks this way:

```
mathematics.make_font ( "px-math", {
  { name="texgyrepagella-regular", features="virtualmath", main=true },
  { name="pxr", vector="tex-mr" } ,
  { name="pxmi", vector="tex-mi", skewchar=0x7F },
  { name="pxsy", vector="tex-sy", skewchar=0x30, parameters=true } ,
  { name="pxex", vector="tex-ex", extension=true } ,
  { name="pxsya", vector="tex-ma" },
  { name="pxsyb", vector="tex-mb" },
} )
```

As you can see, it is possible to add alphabets, given that there is a suitable vector that maps glyph indices onto Unicodes. It is good to know that this function only defines the way such a font is constructed. The actual construction is delayed till the font is needed.

Such a virtual font is used in typescripts (the building blocks of typeface definitions in ConTEXt) as follows:

```
\starttypescript [math] [palatino] [name]
  \definefontsynonym [MathRoman] [pxmath@px-math]
  \loadmapfile[original-youngryu-px.map]
\stoptypescript
```

If you are familiar with the way fonts are defined in ConTEXt, you will notice that we no longer need to define MathItalic, MathSymbol and additional symbol fonts. Of course users don't have to deal with these issues themselves. The @ triggers the virtual font builder.

You can imagine that in MkII switching to another font style or size involves initializing (or at least checking) some 30 to 40 font definitions when it comes to math (the number of used families times 3, the number of math sizes.). And even if we take into account that fonts are loaded only once, this checking and enabling takes time. Keep in mind that in ConTEXt we can have several math font sets active in one document which comes at a price.

In MkIV we use one family (at three sizes). Of course we need to load the font (and more than one in the case of virtual variants) but when switching bodyfont sizes we only need to enable one (already defined) math font. And that really saves time. This is one of the areas where we gain back time that we loose elsewhere by extending core functionality using Lua (like OpenType support).

Dimensions

By setting font related dimensions you can control the way TEX positions math elements relative to each other. Math fonts have a few more dimensions than regular text fonts. But OpenType math fonts like Cambria have quite some more. There is a nice booklet published by Microsoft, 'Mathematical Typesetting', where dealing with math is discussed in the perspective of their word processor and TEX. In the booklet some of the parameters are discussed and since many of them are rather special it makes no sense (yet) to elaborate on them here. Figuring out their meaning was quite a challenge.

I am the first to admit that the current code in MkIV that deals with math parameters is somewhat messy. There are several reasons for this:

- We can pass parameters as a `MathConstants` table in the tfm table that we pass to the core engine.
- We can use some named parameters, like `x_height` and pass those in the `parameters` table.
- We can use the traditional font dimension numbers in the `parameters` table, but since they overlap for symbol and extensible fonts, that is asking for troubles.

Because in MkIV we create virtual fonts at run-time and use just one family, we fill the `MathConstants` table for traditional fonts as well. Future versions may use the upcoming mechanisms of font parameter sets at the macro level. These can be defined for each of the sizes (display, text, script and scriptscript, and the last three in cramped form as well) but since a font only carries one set, we currently use a compromise.

Tracing

One of the nice aspects of the opened up math machinery is that it permits us to get a more detailed look at what happens. It also fits nicely in the way we always want to visualize things in ConTEXt using color, although most users are probably unaware of many such features because they don't need them as I do.

```
\enabletrackers[math.analyzing]
\ruledhbox{$a = \sqrt{b^2 + \sin{c} - {1 \over \gamma}}$}
\disabletrackers[math.analyzing]
```

$\boxed{a = \sqrt{b^2 + \sin c - \frac{1}{\gamma}}}$

This tracker option colors characters depending on their nature and the fact that they are remapped. The tracker also was handy during development of LuaTEX especially for checking if attributes migrated right in constructed symbols.

For over a year I had been using a partial Unicode math implementation in some projects but for serious math the vectors needed to be completed. In order to help the 'math department' of the ConTEXt development team (Aditya Mahajan, Mojca Miklavec, Taco Hoekwater and myself) we have some extra tracing options, like

```
\showmathfontcharacters[][0x0007B]
```

U+0007B: { left curly bracket
width: 253760, height: 463680, depth: 146560, italic: 0
mathclass: open, mathname: lbrace
next: U+F03B0 { => U+F04CE { => U+F03B1 { => U+F04D4 { => U+F03B2 { => U+F04DA { => U+F03B3 { => variants: U+023A9 ⎩ => U+023AA ⎪ => U+023A8 ⎨ => U+023AA ⎪ => U+023A7 ⎧

The simple variant with no arguments would have extended this document with many pages of such descriptions.

Another handy command (defined in module `fnt-25`) is the following:

```
\ShowCompleteFont{name:cambria}{9pt}{1}
\ShowCompleteFont{dummy@lmroman10-math}{10pt}{1}
```

For Cambria this will generate between 50 and 100 pages of character tables.

If you look at the following samples you can imagine how coloring the characters and replacements helped figuring out the alphabets. We use the following input (stored in a buffer):

```
$abc \bf abc \bi abc$
$\mathscript abcdefghijklmnopqrstuvwxyz $
$\mathscript 1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ$
$\mathfraktur abcdefghijklmnopqrstuvwxyz$
$\mathfraktur 1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ$
$\mathblackboard abcdefghijklmnopqrstuvwxyz $
$\mathblackboard 1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ$
```

```
$\mathscript abc IRZ \mathfraktur abc IRZ $
$\mathblackboard abc IRZ \ss abc IRZ 123$
```

For testing Cambria we say:

```
\usetypescript[cambria]
\switchtobodyfont[cambria,11pt]
\enabletrackers[math.analyzing]
\getbuffer[mathtest] % the input shown before
\disabletrackers[math.analyzing]
```

And we get:

*abc***abc*****abc***
abcdefghijklmnopqrstuvwxyz
1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcIRZabcIRZ
abcIRZabcIRZ123

For the virtualized Latin Modern we say:

```
\usetypescript[modern]
\switchtobodyfont[modern,11pt]
\enabletrackers[math.analyzing]
\getbuffer[mathtest] % the input shown before
\disabletrackers[math.analyzing]
```

This gives:

*abc*abc***abc***
abcdefghijklmnopqrstuvwxyz
1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcIRZabcIRZ
abcIRZabcIRZ123

These two samples demonstrate that Cambria has a rather complete repertoire of shapes which is no surprise because it is a recent font that also serves as a showcase for Unicode and OpenType driven math.

Commands like `\mathscript` set an attribute. When we post-process the noad list and encounter this attribute, we remap the characters to the desired variant. Of course this happens selectively. So, a capital A (`0x0041`) becomes a capital script A (`0x1D49C`). Of course this solution is rather ConTEXt-specific and there are other ways to achieve the same goal (like using more families and switching family.)

Special cases

Because we now are operating in the Unicode domain, we run into problems if we keep defining some of the math symbols in the traditional TEX way. Even with the ams fonts available, we still end up with some characters that are represented by combining others. Take for instance ≠ which is composed of two characters. Because in MkIV we want to have all characters in their pure form, we use a virtual replacement for them. In MkIV speak it looks like this:

```
local function negate(main,unicode,basecode)
    local characters = main.characters
    local basechar = characters[basecode]
    local ht, wd = basechar.height, basechar.width
    characters[unicode] = {
        width    = wd,
        height   = ht,
        depth    = basechar.depth,
        italic   = basechar.italic,
        kerns    = basechar.kerns,
        commands = {
            { "slot", 1, basecode },
            { "push" },
            { "down",    ht/5},
            { "right", - wd/2},
            { "slot", 1, 0x2215 },
            { "pop" },
        }
    }
end
```

In case you're curious, there are indeed kerns, in this case the kerns with the Greek Delta.

Another thing we need to handle is positioning of accents on top of slanted (italic) shapes. For this TEX uses a special character in its fonts (set with \skewchar). In its kerning table, any character can have a kern towards this special character. From this kern we can calculate the top_accent variable that we can pass for each character. This variable lives at the same level as width, height, depth and italic and is calculated as: $w/2 + k$, so it defines the horizontal anchor. A nice side effect is that (in the ConTEXt font management subsystem) this saves us passing information associated with specific fonts such as the skew character.

A couple of concepts are unique to TEX, like having \hat and \widehat where the wide one has sizes. In OpenType and Unicode we don't have this distinction so we need special trickery to simulate this. We do so by adding extra code points in a private Unicode space which in return results in them being defined automatically and the relevant first size variant being used for \hat. For some users this might still be too wide but at least it's better than a wrongly positioned ascii variant. In the future we might use this private space for similar cases.

Arrows, horizontal extenders and radicals also fall in the category ‘troublesome’ if only because they use special dimensions to get the desired effect. Fortunately OpenType math is modeled after TEX, so in LuaTEX we introduce a couple of new constructs to deal with this. One such simplification at the macro level is in the definition of \root. Here we use the new \Uroot primitive. The placement related parameters are those used by traditional TEX, but when they are available the OpenType variants are applied. The simplified plain definitions are now:

```
\def\rootradical{\Uroot 0 "221A }

\def\root#1\of{\rootradical{#1}}

\def\sqrt{\rootradical{}}
```

The successive sizes of the root will be taken from the font in the same way as traditional TEX does it. In that sense LuaTEX is not doing anything differently, it only has more parameters to control the process. The definition of \sqrt in ConTEXt permits an optional first argument that sets the degree.

U+0221A: √ square root
    width: 430400, height: 603520, depth: 27200, italic: 0
    mathclass: radical, mathname: surd

    next: U+F03F8 √ => U+F03F9 √ => U+F03FA √ => U+F03FB √

=> U+F03FC √ => variants: U+023B7 √ => U+020D3 □ => U+F04A1 □

Note that we have collected all characters in family 0 (simply because that is what TeX defaults characters to) and that we use the formal Unicode slots. When we use the Latin Modern fonts we just remap traditional slots to the right ones.

Another neat trick is used when users choose among the bigger variants of some characters. The traditional approach is to create a box of a certain size and create a fake delimited variant which is then used.

```
\definemathcommand [big]  {\choosemathbig\plusone  }
\definemathcommand [Big]  {\choosemathbig\plustwo  }
\definemathcommand [bigg] {\choosemathbig\plusthree}
\definemathcommand [Bigg] {\choosemathbig\plusfour }
```

Of course this can become a primitive operation and we might decide to add such a primitive later on so we won't bother you with more details.

Attributes are also used to make live easier for authors who have to enter lots of pairs. Compare:

```
\setupmathematics[autopunctuation=no]

$ (a,b) = (1.20,3.40) $
```

$(a, b) = (1.20, 3.40)$

with:

```
\setupmathematics[autopunctuation=yes]

$ (a,b) = (1.20,3.40) $
```

$(a{,}b) = (1.20{,}3.40)$

So we don't need to use this any more:

```
$ (a{,}b) = (1{.}20{,}3{.}40) $
```

Features like this are implemented on top of an experimental math manipulation framework that is part of MkIV. When the math font system is stable we will rework the rest of math support and implement additional manipulating frameworks.

Control

As with all other character related issues, in MkIV everything is driven by a character table (consider it a database). Quite some effort went into getting that one right and although by now math is represented well, more data will be added in due time.

In MkIV we no longer have huge lists of TEX definitions for math related symbols. Everything is initialized using the mentioned table: normal symbols, delimiters, radicals, with or without name. Take for instance the square root:

U+0221A: √ square root
width: 430400, height: 603520, depth: 27200, italic: 0
mathclass: radical, mathname: surd

next: U+F03F8 √ => U+F03F9 √ => U+F03FA √ => U+F03FB √

=> U+F03FC √ => variants: U+023B7 √ => U+020D3 □ => U+F04A1 □

Its entry is:

```
[0x221A] = {
    adobename = "radical",
    category = "sm",
    cjkwd = "a",
    description = "SQUARE ROOT",
    direction = "on",
    linebreak = "ai",
    mathclass = "radical",
    mathname = "surd",
    unicodeslot = 0x221A,
}
```

The fraction symbol also comes in sizes (this symbol is not to be confused with the negation symbol `0x2215` – which is known as `\not` in TEX terminology):

U+02044: ⁄ fraction slash
width: 362880, height: 457920, depth: 137600, italic: 0
mathclass: binary, mathname: slash
mathclass: close, mathname: solidus

next: U+F03AC ⁄ => U+F03AD ⁄ => U+F03AE ⁄ => U+F03AF ⁄

```
[0x2044] = {
    adobename = "fraction",
    category = "sm",
    contextname = "textfraction",
    description = "FRACTION SLASH",
    direction = "cs",
    linebreak = "is",
    mathspec = {
        { class = "binary", name = "slash" },
        { class = "close", name = "solidus" },
    },
    unicodeslot = 0x2044,
}
```

However, since most users don't have this symbol visualized in their word processor, they expect the same behavior from the regular slash. This is why we find a reference to the real symbol in its definition.

U+0002F: / solidus
  width: 321280, height: 457920, depth: 137600, italic: 0
  mathsymbol: U+02044 ⁄

The definition is:

```
[0x002F] = {
    adobename = "slash",
    category = "po",
    cjkwd = "na",
    contextname = "textslash",
    description = "SOLIDUS",
    direction = "cs",
    linebreak = "sy",
    mathsymbol = 0x2044,
    unicodeslot = 0x002F,
}
```

One problem left is that currently we have only one class per character (apart from the delimiter and radical usage which have their own definitions). Future releases of ConTEXt will provide support for math dictionaries (as in OpenMath and MathML 3). At that point we will also have a `mathdict` entry.

There is another issue with character mappings, one that will seldom reveal itself to the user, but might confuse macro writers when they see an error message.

In traditional TEX, and therefore also in the Latin Modern fonts, a chain from small to large character goes in two steps: the normal size is taken from one family and the larger variants from another. The larger variant then has a pointer to an even larger one and so on, until there is no larger variant or an extensible recipe is found. The default family is number 0. It is for this reason that some of the definition primitives expect a small and large family part.

However, in order to support OpenType in LuaTEX, the alternative method no longer assumes this split. After all, we no longer have a situation where the 256 limit forces us to take the smaller variant from one font and the larger sequence from another (so we need two family–slot pairs where each family eventually resolves to a font).

It is for that reason that the new `\U...` primitives expect only one family specification: the small symbol, which then has a pointer to a larger variant when applicable. However deep down in the engine, there is still support for the multiple family solution (after all, we don't want to drop compatibility). As a result, in error messages you can still find references (defaulting to 0) to large specifications, even if you don't use them. In that case you can simply ignore the large symbol (0,0), since it is not used when the small symbol provides a link.

Extensibles

In TEX fences can be told to become larger automatically. In traditional TEX a character can have a linked list of next larger shapes ending in a description of how to compose even larger variants.

A parenthesis in Cambria has the following list:

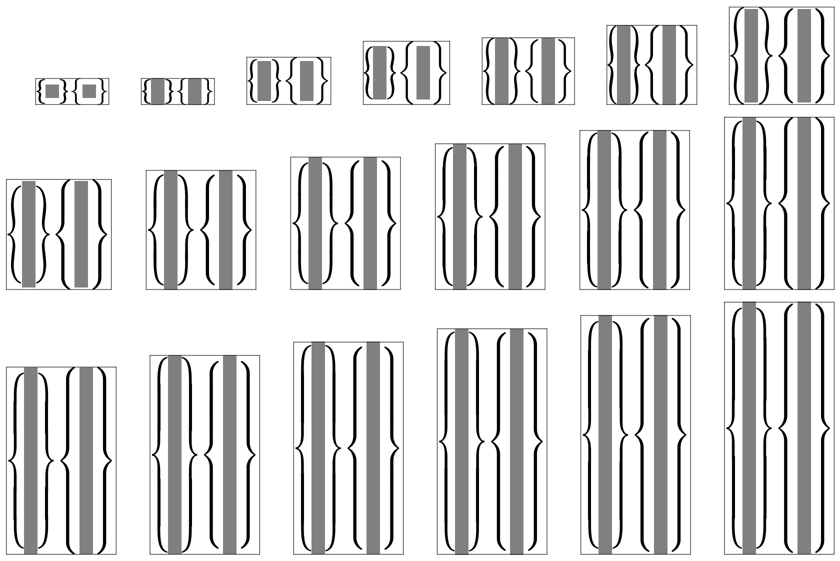
U+00028: (left parenthesis
  width: 272000, height: 462400, depth: 144640, italic: 0
  mathclass: open, mathname: lparent

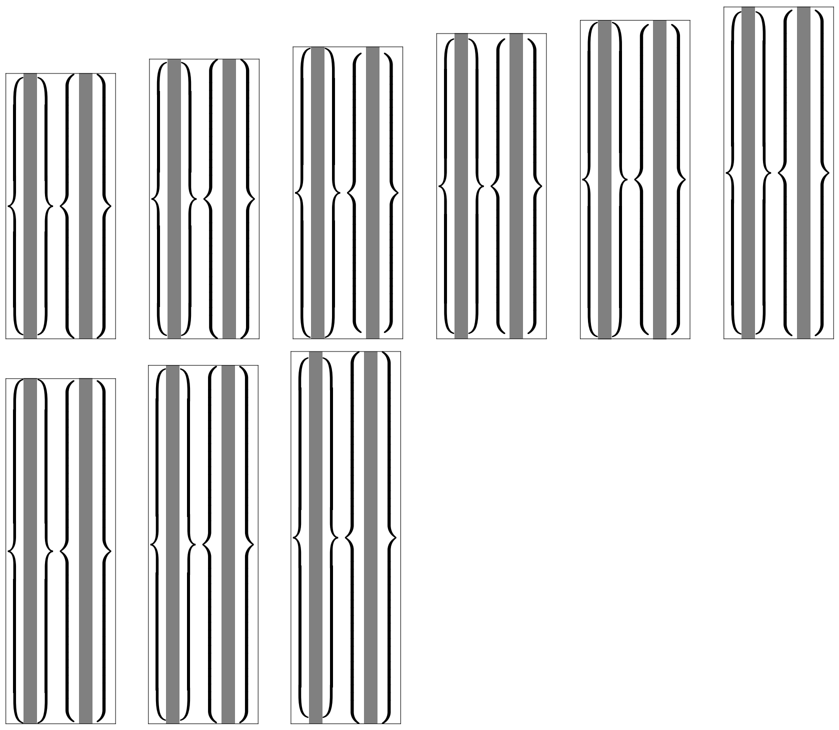
next: U+F03C0 => U+F04CA => U+F03C1 => U+F04D0 =>
U+F03C2 => U+F04D6 => U+F03C3 => variants: U+0239D
=> U+0239C => U+0239B

In Latin Modern we have:

U+00028: left parenthesis
width: 254935.04, height: 491520, depth: 163840, italic: 0
mathclass: open, mathname: lparent
next: U+FF000 => U+FF010 => U+FF012 => U+FF020
=> U+FF030 => variants: U+FF040 => U+FF042 =>
U+FF030

Of course, LuaTEX is downward compatible with respect to this feature, but the internal representation is now closer to what OpenType math provides (which is not that far from how TEX works, simply because it is inspired by TEX). Because Cambria has different parameters we get slightly different results. In the following list of pairs, you see Cambria on the left, Latin Modern on the right. Both start with stepwise larger shapes, followed by a more gradual growth. The thresholds for a next step are driven by parameters set in the OpenType font or by TEX's default.





In traditional TeX horizontal extensibles are not really present. Accents are chosen from a linked list of variants and don't have an extensible specification. This is because most such accents grow in two dimensions and the only extensible accents are rules and braces. However, in Unicode we have a few more and also because of symmetry we decided to add horizontal extensibles too. Take:

```
$ \overbrace {a+1} \underbrace {b+2} \doublebrace {c+3} $ \par
$ \overparent{a+1} \underparent{b+2} \doubleparent{c+3} $ \par
```

This gives:

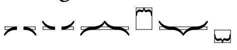
$\overbrace{a+1}\,\underbrace{b+2}\,\overbrace{\underbrace{c+3}}$

$\overparen{a+1}\,\underparen{b+2}\,\overparen{\underparen{c+3}}$

Contrary to Cambria, Latin Modern Math, which is just like Computer Modern Math, has no ready overbrace glyphs. Keep in mind that in the latter we are dealing with fonts that have only 256 slots and that the traditional font mechanism has the same limitation. For this reason, the (extensible) braces are traditionally made from snippets as is demonstrated below.

```
\hbox\bgroup
  \ruledhbox{\getglyph{lmex10}{\char"7A}}
  \ruledhbox{\getglyph{lmex10}{\char"7B}}
  \ruledhbox{\getglyph{lmex10}{\char"7C}}
  \ruledhbox{\getglyph{lmex10}{\char"7D}}
  \ruledhbox{\getglyph{lmex10}{\char"7A\char"7D\char"7C\char"7B}}
  \ruledhbox{\getglyph{name:cambriamath}{\char"23DE}}
  \ruledhbox{\getglyph{lmex10}{\char"7C\char"7B\char"7A\char"7D}}
  \ruledhbox{\getglyph{name:cambriamath}{\char"23DF}}
\egroup
```

This gives:



The four snippets have the height and depth of the rule that will connect them. Since we want a single interface for all fonts we no longer will use macro based solutions. First of all fonts like Cambria don't have the snippets, and using active character trickery (so that we can adapt the meaning to the font) has no preference either. This confines us to virtual glyphs.

It took us a bit of experimenting to get the right virtual definition because it is a multi–step process:

- The right Unicode character (0x23DE) points to a character that has no glyph itself but only horizontal extensibles.
- The snippets that make up the extensible don't have the right dimensions (as they define the size of the connecting rule), so we need to make them virtual themselves and give them a size that matches LuaTEX's expectations.
- Each virtual snippet contains a reference to the physical snippet and moves it up or down as well as fixes its size.
- The second and fifth snippet are actually not real glyphs but rules. The dimensions are derived from the snippets and it is shifted up or down too.

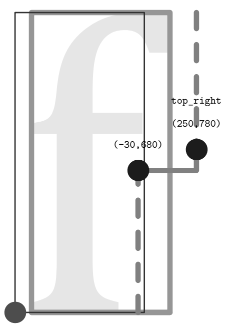
You might wonder if this is worth the trouble. Well, it is if you take into account that all upcoming math fonts will be organized like Cambria.

Math kerning

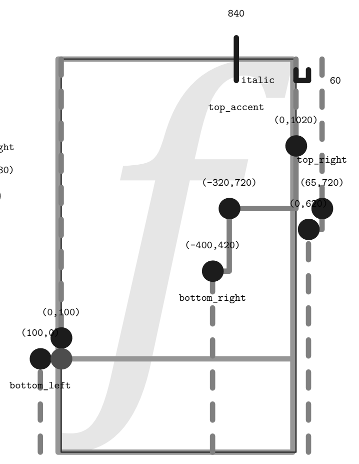
While reading Microsofts orange booklet, it became clear that OpenType provides advanced kerning possibilities and we decided to put it on the agenda for LuaTEX.

It is possible to define a ladder–like boundary for each corner of a character where the ladder more or less follows the shape of a character. In theory this means that when we attach a superscript to a base character we can use two such ladders to determine the optimal spacing between them.

Let's have a look at a few characters, the upright f and its italic cousin.



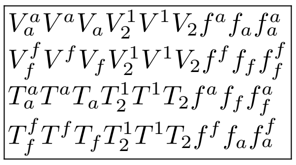
U+00066

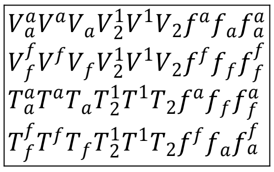


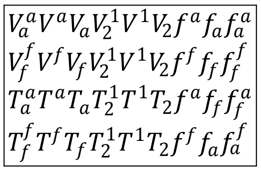
0x1D453

The ladders on the right can be used to position a super- or subscript, that is, they are positioned in the normal way but the ladder, as well as the boundingbox and/or left ladders of the scripts, can be used to fine tune the positioning.

Should we use this information? I made this visualizer for checking some Arabic fonts anchoring and cursive features and then it made sense to add some of the information related to math as well. (Taco extended the visualizer for his presentation at Bachotek 2009 so you might run into variants.) The orange booklet shows quite advanced ladders, and when looking at the 3500 shapes in Cambria, it quickly becomes clear that in practice there is not that much detail in the specification. Nevertheless, because without this feature the result is not acceptable, LuaTEX gracefully supports it.


latin modern


cambria
without kerning


cambria with kerning

Faking glyphs

A previous section already discussed virtual shapes. In the process of replacing all shapes that lack in Latin Modern and are composed of snippets instead, we ran into the dots. As they are a nice demonstration of something that, although somewhat of a hack, survived 30 years without problems we show the definition used in ConTEXt MkII:

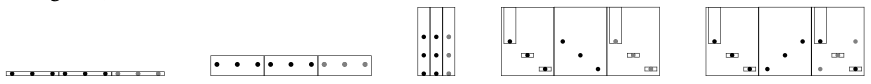
```
\def\PLAINldots{\ldotp\ldotp\ldotp}
\def\PLAINcdots{\cdotp\cdotp\cdotp}

\def\PLAINvdots
  {\vbox{\forgetall\baselineskip.4\bodyfontsize\lineskiplimit\zeropoint
        \kern.6\bodyfontsize\hbox{.}\hbox{.}\hbox{.}}}

\def\PLAINddots
  {\mkern1mu%
   \raise.7\bodyfontsize\ruledvbox{\kern.7\bodyfontsize\hbox{.}}%
   \mkern2mu%
   \raise.4\bodyfontsize\relax\ruledhbox{.}%
   \mkern2mu%
   \raise.1\bodyfontsize\ruledhbox{.}%
   \mkern1mu}
```

This permitted us to say:

```
\definemathcommand [ldots] [inner]   {\PLAINldots}
\definemathcommand [cdots] [inner]   {\PLAINcdots}
\definemathcommand [vdots] [nothing] {\PLAINvdots}
\definemathcommand [ddots] [inner]   {\PLAINddots}
```

However, in MkIV we use virtual shapes instead.

The following lines show the virtual shapes in greyscale. In each triplet we see the original, the virtual and the overlaid character.



As you can see here, the virtual variants are rather close to the originals. At 12pt there are no real differences but (somehow) at other sizes we get slightly different results but it is hardly visible. Watch the special spacing above the shapes. It is probably needed for getting the spacing right in matrices (where they are used).

Hans Hagen

LuaTEX – Halfway

Introduction

We are about halfway the LuaTEX project now. At the time of writing this document we are only a few days away from version 0.40 (the BachoTEX cq. TEXlive version) and around EuroTEX 2009 we will release version 0.50. Starting with version 0.30 (which we released around the conference of the Korean TEX User group meeting) all one-decimal releases are supported and usable for (controlled) production work. We have always stated that all interfaces may change until they are documented to be stable, and we expect to document the first stable parts in version 0.50. Currently we plan to release version 1.00 sometime in 2012, 30 years after TEX82, with 0.60 and 0.70 in 2010, 0.80 and 0.90 in 2011. But of course it might turn out different.

In this update we assume that the reader knows what LuaTEX is and what it does.

Design principles

We started this project because we wanted an extensible engine and have chosen Lua as glue language. We do not regret this choice as it permitted us to open up TEX's internals pretty well. There have been a few extensions to TEX itself and there will be a few more but none of them are fundamental in the sense that they influence typesetting. Extending TEX in that area is up to the macro package writer who can use the Lua language combined with TEX macros. In a similar fashion we made some decisions about Lua libraries that are included. What we have now is what you will get. Future versions of LuaTEX will have the ability to load additional libraries but these will not be part of the core distribution. There is simply too much choice and we do not want to enter endless discussions about what is best. More flexibility would also add a burden on maintenance that we do not want. Portability has always been a virtue of TEX and we want to keep it that way.

Lua scripting

Before 0.40 there could be multiple instances of the Lua interpreter active at the same time, but we decided to limit the number of instances to just one. The reason is simple: sharing all functionality among multiple Lua interpreter instances does more bad than good and Lua has enough possibilities to create namespaces anyway. The new limit also simplifies the internal source code, which is a good thing. While the `\directlua` command is now sort of frozen, we might extend the functionality of `\latelua` especially in relation to what is possible in the backend. Both commands still accept a number but this now refers to an index in a user–definable name table that will be shown when an error occurs.

Input and output

The current LuaTEX release permits multiple instances of kpse which can be handy if you mix for instance a macro package and mplib, as both have there own ‘progname’ (and engine) namespace. However, right from the start it has been possible to bring most input under Lua control and one can overload the usual kpse mechanisms. This is what we do in ConTEXt (and probably only there).

Logging et cetera is also under Lua control. There is no support for writing to TEX's opened output channels except for the log and the terminal. We are thinking about limited write control to numbered channels but this has a very low priority.

Reading from zip files and sockets has been available for a while now.

Among the first things that have been implemented is a mechanism for managing category codes (`\catcode`) although this is not really needed for practical usage as we aim at full compatibility. It just makes printing back to TEX from Lua a bit more comfortable.

Interface to TEX

Registers can always be accessed from Lua by number and (when defined at the TEX end) also by name. When writing to a register, grouping is honored. Most internal registers can be accessed (mostly read-only). Box registers can be manipulated but users need to be aware of potential memory management issues.

There will be provisions to use the primitives related to setting codes (lowercase codes and such). Some of this functionality will be available in version 0.50.

Fonts

The internal font model has been extended to the full Unicode range. There are readers for OpenType, Type1, and traditional TEX fonts. Users can create virtual fonts on the fly and have complete control over what goes into TEX. Font specific features can either be mapped onto the traditional ligature and kerning mechanisms or be implemented in Lua.

We use code from FontForge that has been stripped to get a smaller code base. Using the FontForge code has the advantage that we get a similar view on the fonts in LuaTEX as in this editor which makes debugging easier and developing fonts more convenient.

The interface is already rather stable but some of the keys in loaded tables might change. Almost all of the font interface will be stable in version 0.50.

Tokens

It is possible to intercept tokenization. Once intercepted, a token table can be manipulated before being piped back into LuaTEX. We still support Omega's translation processors but that might become obsolete at some point.

Future versions of LuaTEX might use Lua's so called user data concept but the interface will mostly be the same. Therefore this subsystem will not be frozen yet in version 0.50.

Nodes

Users have access to the node lists in various stages. This interface has already been quite stable for some time but some cleanup might still take place. Currently the node memory maintenance is still explicit, but we will eventually make releasing unused nodes automatic.

We have plans for keeping more extensive information within a paragraph (initial whatsit) so that one can build alternative paragraph builders in Lua. There will be a vertical packer (in addition to the horizontal packer) and we will open up the page builder (inserts et cetera). The basic interface will be stable in 0.50.

Attributes

This new kid on the block is now available for most subsystems but we might change some of its default behavior. As of 0.40 you can also use negative values for attributes. The original idea of using negative values for special purposes has been abandoned as we consider a secondary (faster and more efficient) limited variant. The basic principles will be stable around version 0.50, but we reserve the freedom to change some aspects of attributes until we reach version 1.00.

Hyphenation

In LuaTEX we have clearly separated hyphenation, ligature building and kerning. Managing patterns as well as hyphenation is reimplemented from scratch but uses the same principles as traditional TEX. Patterns can be loaded at run time and exceptions are quite efficient now. There are a few extensions, like embedded discretionaries in exceptions and pre- as well as posthyphens.

On the agenda is fixing some ‘hyphenchar’ related issues and future releases might deal with compound words as well. There are some known limitations that we hope to have solved in version 0.50.

Images

Image handling is part of the backend. This part of the pdfTEX code has been rewritten and can now be controlled from Lua. There are already a few more options than in pdfTEX (simple transformations). The image code will also be integrated in the virtual font handler.

Paragraph building

The paragraph builder has been rewritten in C (soon to be converted back to cweb). There is a callback related to the builder so it is possible to overload the default line breaker by one written in Lua.

There are no further short-term revisions on the agenda, apart from writing an advanced (third order) Arabic routine for the Oriental TEX project.

Future releases may provide a bit more control over `\parshapes` and multiple paragraph shapes.

Metapost

The closely related mplib project has resulted in a MetaPost library that is included in LuaTEX. Multiple instances can be active at the same time and MetaPost processing is very fast. Conversion to pdf is to be done with Lua.

On the todo list is a bit more interoperability (pre- and postscript tables) and this will make it into release 0.50 (maybe even in version 0.40 already).

Mathematics

Version 0.50 will have a stable version of Unicode math support. Math is backward compatible but provides solutions for dealing with OpenType math fonts. We provide math lists in their intermediate form (noads) so that it is possible to manipulate math in great detail.

The relevant math parameters are reorganized according to what OpenType math provides (we use Cambria as reference). Parameters are grouped by style. Future versions of LuaTEX will build upon this base to provide a simple mechanism for switching style sets and font families in-formula.

There are new primitives for placing accents (top and bottom variants and extensible characters), creating radicals, and making delimiters. Math characters are permitted in text mode.

There will be an additional alignment mechanism analogous to what MathML provides. Expect more.

Page building

Not much work has been done on opening up the page builder although we do have access to the intermediate lists. This is unlikely to happen before 0.50.

Going cweb

After releasing version 0.50 around EuroTEX 2009 there will be a period of relative silence. Apart from bug fixes and (private) experiments there will be no release for a while. At the time of the 0.50 release the LuaTEX source code will probably be in plain C completely. After that is done, we will concentrate strongly on consolidating and upgrading the code base back into cweb.

Cleanup

Cleanup of code is a continuous process. Cleanup is needed because we deal with a merge of traditional TEX, ε-TEX extensions, pdfTEX functionality and some Omega (Aleph) code.

Compatibility is a prerequisite, with the exception of logging and rather special ligature reconstruction code.

We also use the opportunity to slowly move away from all the global variables that are used in the Pascal version.

Alignments

We do have some ideas about opening up alignments, but it has a low priority and it will not happen before the 0.50 release.

Error handling

Once all code is converted to cweb, we will look into error handling and recovery. It has no high priority as it is easier to deal with after the conversion to cweb.

Backend

The backend code will be rewritten stepwise. The image related code has already been redone, and currently everything related to positioning and directions is redesigned and made more consistent. Some bugs in the Aleph code (inherited from Omega) have been removed and we are trying to come up with a consistent way of dealing with directions. Conceptually this is somewhat messy because much directionality is delegated to the backend.

We are experimenting with positioning (preroll) and better literal injection. Currently we still use the somewhat fuzzy pdfTEX methods that evolved over time (direct, page and normal injection) but we will come up with a clearer model.

Accuracy of the output (pdf) will be improved and character extension (hz) will be done more efficient. Experimental code seems to work okay. This will become available from release 0.40 and onwards and further cleanup will take place when the cweb code is there as much of the pdf backend code is already C.

Context MkIV

When we started with LuaTEX we decided to use a branch of ConTEXt for testing as it involves quite drastic changes, many rewrites, a tight connection with binary versions, et cetera.

As a result for some time we now have two versions of ConTEXt: MkII and MkIV, where the first one targets at pdfTEX and XƎTEX, and the second one is exclusively using LuaTEX. Although the user interface is downward compatible, the code base starts to diverge more and more. Therefore at the last ConTEXt meeting it was decided to freeze the current version of MkII and only apply bug fixes and an occasional simple extension.

This policy change opened the road to rather drastic splitting of the code, also because full compatibility between MkII and MkIV is not required. Around LuaTEX version 0.40 the new, currently still experimental, document structure related code will be merged into the regular MkIV version. This might have some impact as it opens up new possibilities.

The future

In the future, MkIV will try to create (more) clearly separated layers of functionality so that it will become possible to make subsets of ConTEXt for special purposes. This is done under the name MetaTEX. Think of layering like:

- io, catcodes, callback management, helpers
- input regimes, characters, filtering
- nodes, attributes and noads
- user interface
- languages, scripts, fonts and math
- spacing, par building and page construction
- xml, graphics, MetaPost, job management, structure (huge impact)
- modules, styles, specific features
- tools

Fonts

At this moment MkIV is already quite capable of dealing with OpenType fonts. The driving force behind this is the Oriental TEX project which brings along some very complex and feature–rich Arabic font technology. Much time has gone into reverse engineering the specification and behavior of these fonts in Uniscribe (which we use as reference for Arabic).

Dealing with the huge cjk fonts is less a font issue and more a matter of node list processing. Around the annual meeting of the Korean User Group we got much of the machinery working, thanks to discussions on the spot and on the mailing list.

Math

Between LuaTEX versions 0.30 and 0.40 the math machinery was opened up (stage one). In order to test this new functionality, MkIV's math subsystem (that was then already partially Unicode aware) had to be adapted.

First of all Unicode permits us to use only one math family and so MkIV now does that. The implementation uses Microsoft's Cambria Math font as a benchmark. It creates virtual fonts from the other (old and new) math fonts so they appear to match up to Cambria Math. Because the TEXGyre math project is not yet up to speed, MkIV currently uses virtual variants of these fonts that are created at run time. The missing pieces in for instance Latin Modern and friends are compensated for by means of virtual characters.

Because it is now possible to parse the intermediate noad lists MkIV can do some manipulations before the formula is typeset. This is for instance used for alphabet remapping, forcing sizes, and spacing around punctuation.

Although MkIV already supports most of the math that users expect, there is still room for improvement once there is even more control over the machinery. This is possible because MkIV is not bound to downward compatibility.

As with all other LuaTEX related MkIV code, it is expected that we will have to rewrite most of the current code a few times as we proceed, so MkIV math support is not yet stable either. We can take such drastic measures because MkIV is still experimental and because users are willing to do frequent synchronous updating of macros and engine. In the process we hope to get away from all ad-hoc boxing and kerning and whatever solutions for creating constructs, by using the new accent, delimiter, and radical primitives.

Tracing and testing

Whenever possible we add tracing and visualization features to ConTEXt because the progress reports and articles need them. Recent extensions concerned tracing math and tracing OpenType processing.

The OpenType tracing options are a great help in stepwise reaching the goals of the Oriental TEX project. This project gave the LuaTEX project its initial boost and aims at high quality right to left typesetting. In the process complex (test)fonts are made which, combined with the tracing mentioned, helps us to reveal the secrets of OpenType.

Hans Hagen
Taco Hoekwater
Hartmut Henkel

TEX Programming: The past, the present, and the future

Abstract
This article summarizes a recent thread on the ConTEXt mailing list.
(http://archive.contextgarden.net/thread/20090304.193503.1c42e4d5.en.html/)
To make the article interesting, I have changed the question and correspondingly modified the solutions.

Keywords
ConTEXt, luaTEX, TEX Programming

Suppose you want to typeset (in ConTEXt) all possible sums of roll of two dies, like this:

(+)	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

The mundane way to do this, *especially if you do not have too much time at hand,* is to type the whole thing by hand:

```
\bTABLE
  \bTR \bTD $(+)$ \eTD \bTD 1 \eTD \bTD 2 \eTD
       \bTD 3 \eTD \bTD 4  \eTD \bTD 5  \eTD \bTD 6  \eTD \eTR
  \bTR \bTD 1     \eTD \bTD 2 \eTD \bTD 3 \eTD
       \bTD 4 \eTD \bTD 5  \eTD \bTD 6  \eTD \bTD 7  \eTD \eTR
  \bTR \bTD 2     \eTD \bTD 3 \eTD \bTD 4 \eTD
       \bTD 5 \eTD \bTD 6  \eTD \bTD 7  \eTD \bTD 8  \eTD \eTR
  \bTR \bTD 3     \eTD \bTD 4 \eTD \bTD 5 \eTD
       \bTD 6 \eTD \bTD 7  \eTD \bTD 8  \eTD \bTD 9  \eTD \eTR
  \bTR \bTD 4     \eTD \bTD 5 \eTD \bTD 6 \eTD
       \bTD 7 \eTD \bTD 8  \eTD \bTD 9  \eTD \bTD 10 \eTD \eTR
  \bTR \bTD 5     \eTD \bTD 6 \eTD \bTD 7 \eTD
       \bTD 8 \eTD \bTD 9  \eTD \bTD 10 \eTD \bTD 11 \eTD \eTR
  \bTR \bTD 6     \eTD \bTD 7 \eTD \bTD 8 \eTD
       \bTD 9 \eTD \bTD 10 \eTD \bTD 11 \eTD \bTD 12 \eTD \eTR
\eTABLE
```

I am using Natural Tables since it is easy to configure its output (see `http://www.pragma-ade.com/general/manuals/enattab.pdf/`). For example, to get the effect shown above, I use the following setup:

```
\setupTABLE[each][each][width=2em,height=2em,align={middle,middle}]
\setupTABLE[r][1][background=color,backgroundcolor=gray]
\setupTABLE[c][1][background=color,backgroundcolor=gray]
```

Natural tables, however, are not the focus of this article. It is rather, what would you do if you are adventurous and have time at hand. The above is a repetitive task, so it should be possible to automate it. That will save typing errors (unless you make a mistake in your algorithm) and make the code reusable. In any ordinary programming language you could easily write something like the following pseudo code

```
  start_table
  start_table_row
     table_element("(+)")
      for y in [1..6] do
        table_elemnt(y)
  stop_table_row
  for x in [1..6] do
    start_table_row
      table_element(x)
      for y in [1..6] do
        table_element(x+y)
      end
    stop_table_row
  end
stop_table
```

But TEX is no ordinary programming language! Lets try to do this using ConTEXt's equivalent of a for-loop—`\dorecurse`

```
\bTABLE
 \bTR
  \bTD $(+)$ \eTD
  \dorecurse{6}
   {\bTD \recurselevel \eTD}
  \eTR
\dorecurse{6}
 {\bTR
     \bTD \recurselevel \eTD
     \edef\firstrecurselevel{\recurselevel}
  \dorecurse{6}
    {\bTD \the\numexpr\firstrecurselevel+\recurselevel \eTD}
  \eTR}
\eTABLE
```

This, however, does not work as expected because `\dorecurse` is not fully expandable. One way to get around this problem is to *expand* the appropriate parts of the body of `\dorecurse`

```
\bTABLE
 \bTR
  \bTD $(+)$ \eTD
  \dorecurse{6}
   {\expandafter \bTD \recurselevel \eTD}
  \eTR
\dorecurse{6}
 {\bTR
     \edef\firstrecurselevel{\recurselevel}
     \expandafter\bTD \recurselevel \eTD
  \dorecurse{6}
    {\expandafter\bTD
       \the\numexpr\firstrecurselevel+\recurselevel\relax
     \eTD}
  \eTR}
\eTABLE
```

Behold, the `\expandafter`! So, what is this expansion stuff, and why do we need `\expandafter`. TEX has a esoteric executing model, which was succinctly explained by David Kastrup in his TEX interview (`http://www.tug.org/interviews/interview-files/david-kastrup.html/`)

> "Instead, macros are used as a substitute for programming. TEX's macro expansion language is the only way to implement conditionals and loops, but the corresponding control variables can't be influenced by macro expansion (TEX's "mouth" in Knuth's terminology). Instead assignments must be executed by the back end (TEX's "stomach"). Stomach and mouth execute at different times and independently from one another. But it is not possible to solve nontrivial programming tasks with either: only the unholy chimera made from both can solve serious problems. ε-TEX gives the mouth a few more teeth and changes some of that, but the changes are not really fundamental: expansion still makes no assignments."

So, where do we add the `\expandafters`? It's simple, once you get the hang of it (Taco Hoekwater in a ConTEXt mailing list thread (`http://archive.contextgarden.net/message/20060702.141636.a57e6b68.en.html/`))

> "The trick to `\expandafter` is that you (normally) write it backwards until reaching a moment in time where TeX is not scanning an argument.
>
> Say you have a macro that contains some stuff in it to be typeset by `\type`:
>
> ```
> \def\mystuff{Some literal stuff}
> ```
>
> Then you begin with
>
> ```
> \type{\mystuff}
> ```
>
> but that obviously doesn't work, you want the final input to look like
>
> ```
> \type{Some literal stuff}
> ```
>
> Since `\expandafter` expands the token that follows after next token—whatever the next token is—you have to insert it backwards across the opening brace of the argument, like so:
>
> ```
> \type\expandafter{\mystuff}
> ```
>
> But this wouldn't work, yet: you are still in the middle of an expression (the `\type` expects an argument, and it gets `\expandafter` as it stands).
>
> Luckily, `\expandafter` itself is an expandable command, so you jump back once more and insert another one:
>
> ```
> \expandafter\type\expandafter{\mystuff}
> ```
>
> Now you are on 'neutral ground', and can stop backtracking. *Easy, once you get the hang of it.*"

If you do not get the hang of it, relax. ConTEXt provides a command \expanded that expands its arguments.

```
\bTABLE
 \bTR
  \bTD $(+)$ \eTD
  \dorecurse{6}
    {\expanded{\bTD \recurselevel \eTD}}
  \eTR
  \dorecurse{6}
    {\bTR
       \expanded{\bTD \recurselevel \eTD}
       \edef\firstrecurselevel{\recurselevel}
     \dorecurse{6}
       {\expanded{\bTD
        \the\numexpr\firstrecurselevel+\recurselevel\relax \eTD}}
    \eTR}
\eTABLE
```

Using \expanded is easier than using \expandafter, but you still need to understand TEX's expansion mechanism to get it right. For example, if you try

```
  ...
  \dorecurse{6}
    {\expaned{\bTR
      \bTD \recurselevel \eTD
       \edef\firstrecurselevel{\recurselevel}
     \dorecurse{6}
       {\expanded{\bTD
        \the\numexpr\firstrecurselevel+\recurselevel\relax \eTD}}
    \eTR}}
  ...
```

you will get all sorts of TEX errors, and you need to sprinkle \noexpand at correct places to get it to work. So, \expanded is not a silver bullet.

In the above mention mailing list thread, Wolfgang Schuster posted a much neater solution.

```
\bTABLE
 \bTR
  \bTD $(+)$ \eTD
  \dorecurse{6}
   {\bTD #1 \eTD}
  \eTR
\dorecurse{6}
 {\bTR
     \bTD #1 \eTD
  \dorecurse{6}
    {\bTD \the\numexpr#1+##1 \eTD}
  \eTR}
\eTABLE
```

This makes TEX disguise as a **normal** programming language. But only TEX wizards like Wolfgang can discover such solutions. You need to know the TeX digestive system inside out to even attempt something like this. Inspired by Wolfgang's solution, I tried the same thing with ConTEXt's lesser known for loops

```
\bTABLE
  \bTR
    \bTD $(+)$ \eTD
    \for \y=1 \to 6 \step 1 \do
      {\bTD #1 \eTD}
  \eTR
  \for \x=1 \to 6 \step 1 \do
  {\bTR
     \bTD #1 \eTD
    \for \y=1 \to 6 \step 1 \do
    {\bTD \the\numexpr#1+##1 \eTD}
  \eTR}
\eTABLE
```

Is your head hurting. Don't worry. luaTEX provides hope that normal users can do simple programming tasks. Luigi Scarso posted the following code:

```
\startluacode
    tprint = function(s) tex.sprint(tex.ctxcatcodes,s) end
    tprint('\\bTABLE')
    tprint('\\bTR')
    tprint('\\bTD $(+)$ \\eTD')
    for y = 1,6 do
      tprint('\\bTD ' .. y .. '\\eTD')
    end
    tprint('\\eTR')
    for x = 1,6 do
      tprint('\\bTR')
      tprint('\\bTD ' .. x .. '\\eTD')
      for y = 1,6 do
        tprint('\\bTD' .. x+y .. '\\eTD')
      end
      tprint('\\eTR')
    end
    tprint('\\eTABLE')
\stopluacode
```

Finally luaTEX offers a simple way of implementing simple algorithms inside TEX. There is no need to know TEX's digestive system. Write code as you would write in any other programing language!

If you are a TEX programming guru who can keep track of TEX's expansion mechanism, don't fear luaTEX. There are other options for you: mix TEX and MetaPost.

```
\let\normalbTABLE\bTABLE
\let\normaleTABLE\eTABLE

\unexpanded\def\bTABLE{\normalbTABLE}
\unexpanded\def\eTABLE{\normaleTABLE}

\unexpanded\def\dobTR{\dodoubleempty\parseTR}
\unexpanded\def\dobTD{\dodoubleempty\parseTD}
\unexpanded\def\dobTH{\dodoubleempty\parseTH}
\unexpanded\def\dobTN{\dodoubleempty\parseTN}

\let\bTR\dobTR
\let\bTD\dobTD
\let\bTH\dobTH
```

```
\let\bTN\dobTN

\startMPcode
  string table ;
  table = "\bTABLE \bTR \bTD $(+)$ \eTD" &
  for y = 1 upto 6 :
    "\bTD " & decimal y & "\eTD " &
  endfor
  "\eTR " &
  for x = 1 upto 6 :
    "\bTR \bTD " & decimal x & "\eTD " &
    for y = 1 upto 6 :
      "\bTD " & decimal (x+y) & "\eTD " &
    endfor
    "\eTR" &
  endfor
  "\eTABLE" ;
  label(textext(table), origin) ;
\stopMPcode
```

Aditya Mahajan
adityam@umich.edu

TEX beauties and oddities

Abstract
The BachoTEX 2009 conference continued the Pearls of TEX Programming open session introduced in 2005 during which volunteers present TEX-related tricks and shorties.

A permanent call for TEX pearls

What is wanted:

- short TEX or MetaPost macro/macros (half A4 page or half a screen at most),
- the code should be generic; potentially understandable by plain-oriented users,
- results need not be useful or serious, but language-specific, tricky, preferably non-obvious,
- obscure oddities, weird TEX behaviour, dirty and risky tricks and traps are also welcome,
- the code should be explainable in a couple of minutes.

The already collected pearls can be found at http://www.gust.org.pl/pearls. All pearl-divers and pearlgrowers are kindly asked to send the pearl-candidates to pearls@gust.org.pl, where Paweł Jackowski, our pearl-collector, is waiting impatiently. The pearls market-place is active during the entire year, not just before the annual BachoTEX Conference.

Note: The person submitting pearl proposals and/or participating in the BachoTEX pearls session does not need to be the inventor. Well known hits are also welcome, unless already presented at one of our sessions.

Since some seasoned TEX programmers felt indignant of calling ugly TEX constructs 'Pearls of TEX programming', we decided not to irritate them any longer. We hope they will accept 'TEX beauties and oddities' as the session title.

If you yourself have something that fits the bill, please consider. If you know somebody's work that does, please let us know, we will contact the person. We await your contributions even if you are unable to attend the conference. In such a case you are free either to elect one of the participants to present your work or 'leave the proof to the gentle reader' (cf. e.g. http://www.aurora.edu/mathematics/bhaskara.htm).

Needless to say that all contributions will be published in a separate section of the conference proceedings, possibly also reprinted in different TEX bulletins.

Scary space

Hans Hagen & Taco Hoekwater

In pure TeX

```
\show\
\show\ %
```

gives different logs on Hans' and Taco's machines (Hans' is on the left)

```
**space.tex                           **space.tex
(./space.tex                          (./space.tex
> \                                   > \^^M=macro:
=macro:                               ->\ .
->\ .                                 l.1 \show\
l.1 \show\
                                      ?
?                                     > \ =\ .
> \ =\ .                              l.2 \show \
l.2 \show \                                       %
            %                         ?
?                                      )
 )                                    No pages of output.
No pages of output.
```

The visualization of a `^^M` depends of the platform but since there's definitely a newline involved we need to take care of it.

When parsing the input the following happens (this is mentioned in one of the dangerous bends in the TEXbook):

```
 \let\x\ <newline> => \let\x\<endlinechar>
```

This means that when you want to store the meaning of this primitive, you need to make sure that TEX explicitly sees a space instead of a newline. So we get:

```
\let\normalspaceprimitive=\ % space-comment is really needed
```

In ConTeXt this is used for:

```
\unexpanded\def
   \ {\mathortext\normalspaceprimitive{\dontleavehmode\space}}
```

If you don't use the explicit space a simple

```
$\ $
```

will execute `\^^M`. In Plain TeX (and in ConTeXt) we have:

```
\def\^^M{\ } % control <return> = control <space>
```

So this will result in a loop.

Null control sequence

Hans Hagen & Taco Hoekwater

When you do:

```
\endlinechar=-1
\let\x\
```

the macro `\x` is undefined...

Actually `\x` becomes equal to the 'null control sequence' that you would get from

```
\expandafter\def\csname\endcsname{}
```

but that is usually undefined.

And you can even use this effect to assign to the null control sequence without needing `\expandafter`:

```
\endlinechar=-1
\gdef\
    {\message{NULL CS}}

\csname\endcsname
```

$$: empty formula or unmatched display

Hans Hagen & Taco Hoekwater

When you try the following under TEX'S normal catcode regime, you will get an error:

```
$$ $21-09$
```

The message is:

```
! Display math should end with $$.
 <to be read again>
                  2
 l.7 $$ $2
          1-09$
```

But how about this then:

```
\halign{#&#\cr $$ & $21-09$\cr}
```

It magically works! The actual effect is similar to

```
\hbox{$$} % or
\hbox{${}$}
```

In words of the TEXbook (chapter 25, page 287):

> “One consequence of these rules is that `$$' in restricted horizontal mode simply yields an empty math formula.”

<macro> macro

Philip Taylor

Typesetting a multi-lingual document, even something as simple as a Christmas letter, can be time-consuming and error-prone if the embedded languages make frequent use of diacritics. To eliminate both of these problems, I wrote a macro called `\macro` which enables me to encapsulate all of the tricky words and phrases into macros whose names are (normally) identical to the words or phrases but without the corresponding diacritics.

The following code implements the `\macro` macro, and is followed by some sample definitions and applied occurrences.

```
\catcode`\<=\active
\def<#1>{%
     % cf. Bernd Raichle: check if defined, no side effects (2006)
     \if \csname macro:#1\endcsname \relax
            {\bf {$\ll$}#1{$\gg$}}%
     \else
            \csname macro:#1\endcsname
     \fi}

\def\macro#1#2{\expandafter\def\csname macro:#1\endcsname{#2}}

\macro {Zhou Shang Zhi}{Zh\=ou Sh\`ang Zh\=\i}
\macro {Shangzhi}{Sh\`angzh\=\i}
\macro {Sai Gon}{S\`ai G\`on}
\macro {HCM}{H\raise 0.5ex \rlap {\` }\^o Ch\'\i{} Minh}
\macro {Mui Ne}{M\~ui N\'e}
%\macro {Le}{L\rlap {\d e}\^e}

On a~happier note, the year started with both Khanh \&~I~being
invited to spend time with one of my former Chinese teachers,
<Zhou Shang Zhi>, and his family in Kyoto, Japan. <Shangzhi>
was there for one year, teaching at a~local university, and
the last three months were effectively a~holiday for him with
very few formal duties. Knowing that we might like to visit
Kyoto, <Shangzhi> very kindly invited both of us, which we
accepted with great pleasure.

Khanh's journey commenced with a~flight to <Sai Gon>
(``<HCM> City''), from where she took a~'bus south to <Mui Ne>
(a~distance of some 100 miles or so), where her sister
<Le>~Hoa had booked her into a~very posh hotel by the
beach. Once in <Mui Ne>, Khanh hired a~moped driver.
```

On a happier note, the year started with both Khanh & I being invited to spend time with one of my former Chinese teachers, Zhōu Shàng Zhī, and his family in Kyoto, Japan. Shàngzhī was there for one year, teaching at a local university, and the last three months were effectively a holiday for him with very few formal duties. Knowing that we might like to visit Kyoto, Shàngzhī very kindly invited both of us, which we accepted with great pleasure.

Khanh's journey commenced with a flight to Sài Gòn ("Hồ Chí Minh City"), from where she took a 'bus south to Mũi Né (a distance of some 100 miles or so), where her sister $\ll$**Le**$\gg$ Hoa had booked her into a very posh hotel by the beach. Once in Mũi Né, Khanh hired a moped driver.

UTF-8 support detection

Arthur Reutenauer

When you need to detect if you are running an extension of TeX that supports UTF-8 input, you can use an extensive approach by making the list of engines that could be concerned, and check for particular control sequences like `\XeTeXversion` for XeTeX, or `\directlua` for luaTeX. But you can also simply check for UTF-8 directly, by counting the bytes:

Take Τ, the letter Tau from the Greek alphabet, not the Latin one that looks like it. In UTF-8, its encoding form uses two bytes, which means it is read as two characters by 8-bit TeX engines, but only one by UTF-8 engines. Hence, the following lines detect UTF-8 engines:

```
\def\testengine#1#2!{\def\secondarg{#2}}
```

That's Tau (as in TeX),

```
\testengine Τ!\relax
```

```
UTF-8
\ifx\secondarg\empty
       is % We're UTF-8
\else
       not % We're 8-bit
\fi
supported.
```

Abba Don

Grzegorz Murzynowski

What is and what is not a number for TeX? Adoremus magna et mirabilia opera pappæ Knuth!

```
\ifnum 666>'0888${}-222 = 2\times32\times37={}$DCLXVI
   (all the Roman digits except the largest)\fi

\ifnum 666>"000ecce Angelus Pulcherissimus regnavit!\fi

\ifnum 666>"0000ABBA Father call I from the deepest of my s***
\else Breke kekk, breke kekk!\fi

\ifnum 11254493="ABBADDON($\aleph_0$)\fi
```

Note that `A`, `B`, `c`, `D` and `e` are (in some contexts) hexadecimal digits and (in those contexts) `0xecce` $= 60622$ and (in some other contexts) Abbaddon is the name of the Angel of Extinction.

inlinedef: a general recursive token scanner with callbacks

Stephen Hicks

There have been several discussions about uses of `\expandafter` that border on the ridiculous, with as many as fifteen in a row found in actual TeX input files! Additionally, trying to expand past macro parameters #1 causes problems because there is no guarantee that #1 is a single token. It would instead be nice to insert something right before a single token we want to expand far in advance. A slightly more general problem is to scan tokens in the input stream while preserving spaces and grouping.

```
\let\xa\expandafter

\def\scan{\futurelet\foo\switch}
\def\switch{%
  \let\next\normal
  \ifcat\noexpand\foo\space \let\next\dospace\fi
  \ifcat\noexpand\foo\bgroup \let\next\trygroup\fi
  \ifcat\noexpand\foo\relax \try{&\meaning\foo}\fi
  \next}
\def\try#1{\ifcsname #1\endcsname\xa
           \let\xa\next \csname #1\endcsname\fi}
\def\dospace{\toks0\xa{\the\toks\xa0 \space}\xa\scan\unspace}

\xa\def\xa\unspace\space{}
\long\def\trygroup#1#{%
  \def\temp{#1}\xa\let\xa\next
  \ifx\temp\empty\recurse\else \normal\fi\next#1}
\long\def\recurse#1{%
  \begingroup\toks0{}\scan#1\END{}\xa\endgroup\xa
  \toks\xa0\xa\xa\xa{\xa\the\xa\toks\xa0\xa{\the\toks0}}\scan}
\long\def\normal#1{\toks0\xa{\the\toks0 #1}\scan}

\def\callback#1#2#{%
   \def#1{\noexpand#1}\xa\def\csname&\meaning#1\endcsname#2}
```

We can set up a few callbacks, e.g. `\END` to end scanning, and `\EXPAND` to expand the next token:

```
\callback\END#1{}
\callback\EXPAND#1{\expandafter\scan}
```

And now we can get arbitrary tokens from the input stream into `\toks0` using

```
\def\baz{!}
\scan foo {bar \EXPAND\baz} \baz \END
\message{\the\toks0} % foo\space {bar\space !}\space \baz
```

This can be made more general in several ways: if we don't check `\ifcat \noexpand\foo\relax` then we can execute callbacks on arbitrary tokens, including spaces and grouping symbols. Of course this slows things down quite a bit further, which brings me to the main disadvantage of this approach: it takes about 25 times as long as a simple string of `\expandafter`'s, and is therefore not suitable for inner loops. But the code it allows us to write, as long as efficiency isn't important, is much more readable.

Doe-het-zelf presentaties

Abstract
Dit artikel laat zien hoe je zonder een speciaal presentatie-pakket presentaties kunt maken en aan eigen wensen aanpassen.

Keywords
Presentaties geometry wallpaper fancyhdr

Met LaTeX kun je prima presentaties maken. Je kunt kiezen voor een kant-en-klaar pakket zoals Beamer, maar als je wensen eenvoudig zijn dan is een doe-het-zelf presentatie-stijl ook een optie. Het grote voordeel hierbij is dat je meer controle hebt over het eindresultaat.

Wat erbij komt kijken

Pagina definitie. Het enige absoluut noodzakelijke element van een doe-het-zelf presentatie stijl is een pagina-definitie.

De letters worden vanzelf groter naarmate de pagina kleiner wordt, omdat de pagina automatisch wordt vergroot tot schermgrootte. Dus is het meestal niet nodig lettergroottes te wijzigen.

De volgende aanroep van het geometry package definieert een aangepast pagina-formaat:

```
\usepackage[%
 paperwidth=108mm,
 paperheight=81mm,
 width=88mm,
 height=62mm,
 top=9mm,
 footskip=20pt]{geometry}
```

Let op de verhouding 4:3 van het ‘papier’-formaat. Zie de geometry handleiding voor details.

Typografie. Voor een presentatie is het meestal beter om niet in te springen, en in plaats daarvan alinea’s te scheiden met vertikale witruimte:

```
\setlength{\parskip}{6pt}
\setlength{\parindent}{0pt}
```

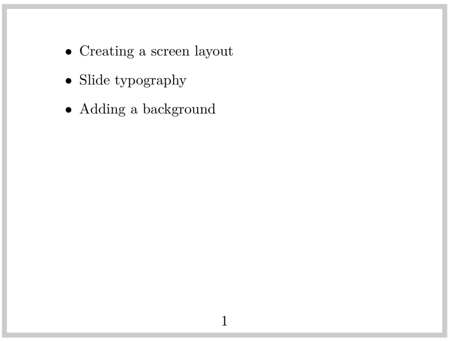
Slides. Je kunt slides simpelweg scheiden met \newpage. Maar als we er nu een environment voor definiëren dan kunnen we straks verfijningen aanbrengen zonder te hoeven ingrijpen in de tekst van het document:

```
\newenvironment{slide}{\newpage}{}
```

Voorbeeld. Met al deze commando’s in de preamble en met de volgende code voor een slide

```
\begin{slide}
\begin{itemize}
\item Creating a screen layout
\item Slide typography
\item Adding a background
\end{itemize}
\end{slide}
```

krijgen we de slide afgebeeld in figuur 1:



- Creating a screen layout
- Slide typography
- Adding a background

1

Figure 1. Article stijl met aangepaste pagina-definitie

Enkele verfijningen

Een classfile. We kunnen code in de preamble stoppen of in een afzonderlijke package, bijvoorbeeld `myslides.sty`. Maar we kunnen ook een echte classfile `myslides.cls` maken. De aanhef hiervan is dan:

```
\LoadClass{article}
\RequirePackage[%
 [...]
 footskip=20pt]{geometry}
```

Merk op dat we in classes en packages meestal \RequirePackage in plaats van \usepackage gebruiken. In ons LaTeX-bestand laden we deze classfile met

```
\documentclass{myslides}
```

Slide titels. We willen onze slides van titels kunnen voorzien. In de volgende definitie is de titel een optionele parameter van de slide omgeving:

```
\newenvironment{slide}[1][]%
 {\newpage {{\large\bfseries #1}}}{}
```

Vertikaal centreren. In de volgende definitie duwen de twee `\vfil`'s samen de inhoud in vertikale richting naar het midden van de pagina, maar laten wel de titel vast bovenaan staan:

```
\newenvironment{slide}[1][]%
 {\newpage {{\large\bfseries #1}}\null\vfil}%
 {\vfil\null}
```

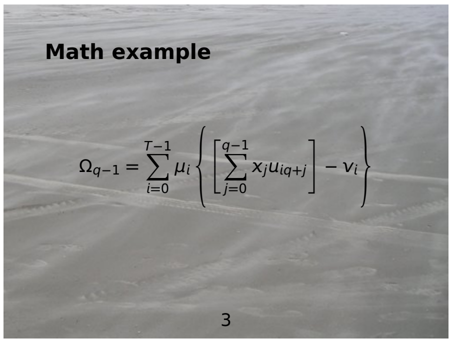
Lettertype. Computer Modern is niet het ideale lettertype voor slides. Een geschiktere keus is Bitstream Vera Sans. Dit schreefloze font omvat ook een goede collectie wiskundige symbolen. Laad hiervoor het Arev package, dat beschikbaar is voor zowel MikTeX als TeX Live:

```
\RequirePackage{arev}
```

Een achtergrond. Met de wallpaper package kunnen we makkelijk een achtergrond toevoegen:

```
\RequirePackage{wallpaper}
\LLCornerWallPaper{1}{zand}
```

Figuur 2 laat het resultaat zien:



Math example

$$\Omega_{q-1} = \sum_{i=0}^{T-1} \mu_i \left\{ \left[\sum_{j=0}^{q-1} x_j u_{iq+j} \right] - v_i \right\}$$

3

Figure 2. Ander lettertype, titel, vertikaal centreren, achtergrond

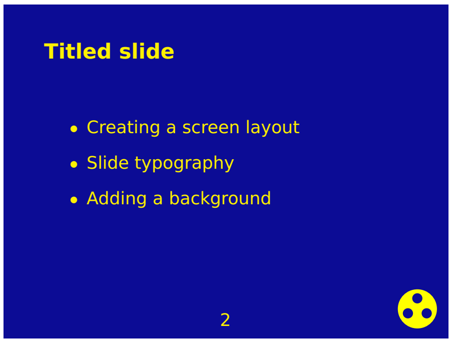
Afstemmen op de achtergrond

Met deze eerste achtergrond konden we de rest van de opmaak ongemoeid laten, maar dat is lang niet altijd het geval.

Lichte tekst op een donkere ondergrond. In het volgende voorbeeld (figuur 3) is alle tekst geel, om beter uit te komen tegen de donkerblauwe achtergrond. Dit doen we met de volgende extra regels in de classfile:

```
\RequirePackage{color}
\color{yellow}
```

Asymmetrische layout. De achtergrond van het volgende voorbeeld, figuur 4, noodzaakt een asymmetrische layout. Het geometry package voorziet hierin: je kunt inplaats van de tekstbreedte ook linker- en rechtermarges opgeven, en inplaats van de teksthoogte boven- en ondermarge:



Titled slide

- Creating a screen layout
- Slide typography
- Adding a background

2

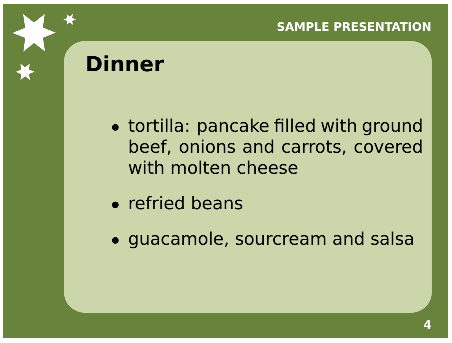
Figure 3. Lichte tekst op donkere achtergrond

```
\RequirePackage[%
 paperwidth=108mm,
 paperheight=81mm,
 vmargin={12mm,7mm},
 hmargin={20mm,6mm},
 headsep=13pt,
 footskip=7pt]{geometry}
```

Headers en footers. We kunnen met gangbare middelen headers en footers definiëren. Piet van Oostrum's package fancyhdr komt hierbij goed van pas:

```
\usepackage{fancyhdr}
\pagestyle{fancy}
% no header/footer rules
\renewcommand{\headrulewidth}{0pt}
\renewcommand{\footrulewidth}{0pt}
% reset contents
\fancyhead{}\fancyfoot{}
% extend into the margin
\fancyhfoffset{2mm}
\rhead{\color{white}\scriptsize\bfseries
 SAMPLE PRESENTATION}
\rfoot{\color{white}\scriptsize\bfseries
 \thepage}
```

Zie de fancyhdr handleiding voor details.



SAMPLE PRESENTATION

Dinner

- tortilla: pancake filled with ground beef, onions and carrots, covered with molten cheese
- refried beans
- guacamole, sourcream and salsa

4

Figure 4. Asymmetrische pagina, aangepaste header en footer

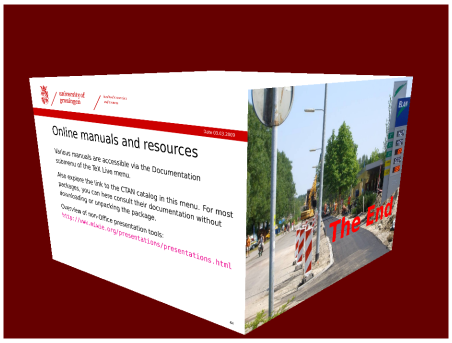
Dynamische effekten

Met het texpower package kun je slides in stappen zichtbaar maken in combinatie met eigen opmaak. Texpower biedt hiervoor een aantal technieken, die echter niet allemaal even fraai combineren met verticaal centreren. Texpower heeft uitgebreide documentatie.

Kunstjes met bestaande pdf-bestanden

In mijn Ubuntu Linux distributie ben ik ook een paar packages tegengekomen die met bestaande pdf-bestanden kunstjes kunnen uithalen.

Pdfcube. Als je een pdf presentatie met pdfcube afspeelt dan kun je, naast een normale pagina-overgang, ook 'roteren' van de ene pagina naar de volgende. Zie figuur 5.



Online manuals and resources

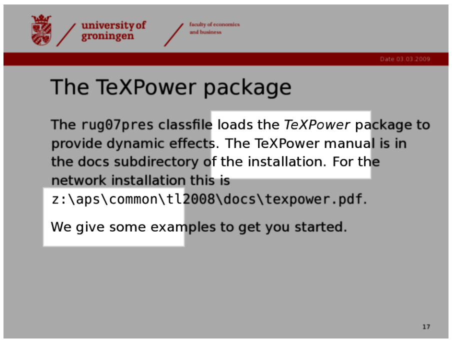
Various manuals are accessible via the Documentation submenu of the TeX Live menu.

Also explore the link to the CTAN catalog in this menu. For most packages, you can here consult their documentation without downloading or unpacking the package.

Overview of non-Office presentation tools:
http://www.miwie.org/presentations/presentations.html

Figure 5. Roterende kubus met pdfcube

Keyjnote. Hoewel keyjnote ook speciale effekten heeft voor pagina-overgangen, is het vooral interessant omdat je er delen van de pagina ermee uit kunt lichten (figuur 6) of met een schijnwerper belichten (figuur 7).

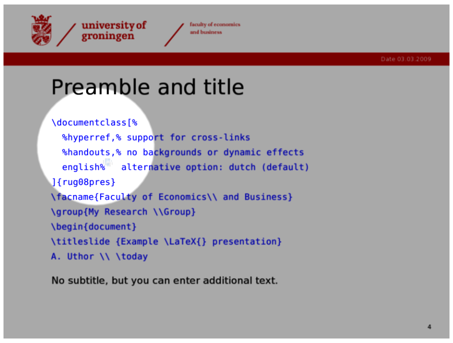


The TeXPower package

The rug07pres classfile loads the *TeXPower* package to provide dynamic effects. The TeXPower manual is in the docs subdirectory of the installation. For the network installation this is z:\aps\common\tl2008\docs\texpower.pdf.

We give some examples to get you started.

Figure 6. KeyJNote highlighten



Preamble and title

```
\documentclass[%
  %hyperref,% support for cross-links
  %handouts,% no backgrounds or dynamic effects
  english%   alternative option: dutch (default)
]{rug08pres}
\facname{Faculty of Economics\\ and Business}
\group{My Research \\Group}
\begin{document}
\titleslide {Example \LaTeX{} presentation}
A. Uthor \\ \today
```

No subtitle, but you can enter additional text.

Figure 7. KeyJNote schijnwerper

Een complete presentatie classfile

```
\LoadClass{article}
\RequirePackage[%
 paperwidth=108mm,
 paperheight=81mm,
 vmargin={12mm,7mm},
 hmargin={20mm,6mm},
 headsep=13pt,
 footskip=7pt]{geometry}

\usepackage{fancyhdr}
\pagestyle{fancy}
% no header/footer rules
\renewcommand{\headrulewidth}{0pt}
\renewcommand{\footrulewidth}{0pt}
% reset contents
\fancyhead{}\fancyfoot{}
% extend into the margin
\fancyhfoffset{2mm}
\rhead{\color{white}\scriptsize\bfseries SAMPLE PRESENTATION}
\rfoot{\color{white}\scriptsize\bfseries \thepage}

\setlength{\parskip}{6pt}
\setlength{\parindent}{0pt}

% Bitstream Vera Sans
\RequirePackage{arev}

\newenvironment{slide}[1][]%
 {\newpage {{\large\bfseries #1}}\null\vfil}{\vfil\null}

% background picture
\RequirePackage{wallpaper}
\LLCornerWallPaper{1}{starred}
```

Siep Kroonenberg
N.S.Kroonenberg at rug dot nl